Context-Aware, Adaptive, and Scalable Android Malware Detection Through Online Learning

Annamalai Narayanan, *Student Member, IEEE*, Mahinthan Chandramohan, *Student Member, IEEE*, Lihui Chen, *Senior Member, IEEE*, and Yang Liu, *Senior Member, IEEE*

***Abstract*—It is well known that Android malware constantly evolves so as to evade detection. This causes the entire malware population to be nonstationary. Contrary to this fact, most of the prior works on machine learning based android malware detection have assumed that the distribution of the observed malware characteristics (i.e., features) does not change over time. In this paper, we address the problem of *malware population drift* and propose a novel online learning based framework to detect malware, named CASANDRA (Context-aware, Adaptive and Scalable ANDRoid mAlware detector). In order to perform accurate detection, a novel graph kernel that facilitates capturing apps security-sensitive behaviors along with their context information from dependence graphs is proposed. Besides being accurate and scalable, CASANDRA has specific advantages: first, being adaptive to the evolution in malware features over time; second, explaining the significant features that led to an apps classification as being malicious or benign. In a large-scale comparative analysis, CASANDRA outperforms two state-of-the-art techniques on a benchmark dataset achieving 99.23% F-measure. When evaluated with more than 87 000 apps collected in-the-wild, CASANDRA achieves 89.92% accuracy, outperforming existing techniques by more than 25% in their typical batch learning setting and more than 7% when they are continuously retained, while maintaining comparable efficiency.**

***Index Terms*—Concept drift, graph kernels, malware detection, online learning.**

I. Introduction

In recent times, malware detection for mobile platforms such as Android has evolved as one of the challenging problems in the field of cyber-security. The number of new Android malware applications (apps) have grown tremendously in recent years. For instance, Symantec reports [1] discovering 430 million new malware in 2015 which is a 36% increase over 2014. Also their capabilities have grown from simple phone cloning, sending premium-rated SMS to complex botnets, cryptolocker and ransomware [32]–[35]. Besides this, attackers continuously enhance the sophistication of malware to evade novel detection techniques. The sheer volume, growth rate and evolution of sophisticated Android malware highlights an imperative need for developing sound and scalable automated malware detection techniques [13], [14], [32], [34], [35].

Manuscript received November 13, 2016; revised April 4, 2017; accepted April 9, 2017. Date of current version May 23, 2017. *(Corresponding author: Annamalai Narayanan.)*

The authors are with Nanyang Technological University, Singapore 639798 (e-mail: annamala002@e.ntu.edu.sg; mahinthan@ntu.edu.sg; elhchen@ntu.edu.sg; yangliu@ntu.edu.sg).

Digital Object Identifier 10.1109/TETCI.2017.2699220

Machine Learning based malware detection: For over a decade, Machine Learning (ML) techniques have been predominantly used to perform malware detection in various platforms (such as Windows and Android) [13]–[18], [22], [32]–[35], [42], [51]. This is because, ML methods automatically learn the characteristics that distinguish malicious behavior, when trained using a collection of malware and benign samples making them amenable for automated detection. ML based approaches extract semantic features from apps behaviors and apply standard classification algorithms (e.g., Support Vector Machine (SVM), Random Forests (RFs), etc.) to perform binary classification. These approaches typically use features such as system calls/Application Programming Interfaces (APIs) invoked, resources and privileges used, control- and data-flows inside apps execution to detect malicious behavior patterns [14], [15], [17], [18], [22], [42]. These semantic features are extracted through static [13]–[16], [18], [42] and dynamic [32], [43] program analysis.

A. Malware Detection Using Graph Representations

Malware variants: A major reason for the 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 [35], [42], [44]. 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 [14]–[18]. In this work, we use a common term, Program Representation Graph (PRG) to refer to any of the 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 [15], [16], [18], [44]. Some methods such as [18], [22], [44] note that ML classifiers are readily applicable on data represented as vectors and attempt to encode PRGs as feature vectors.

Graph Mining and Kernels: Many existing works such as [20] and [21] use off-the-shelf graph mining algorithms on PRGs for malware detection. However, it has to be noted that 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., [44]) are NP hard and have severe scalability issues, making them impractical for real-world malware detection.

On the other hand, one of the increasingly popular approaches in ML for graph-structured data is the use of graph kernels. These graph kernels could be used together with a kernel classifier (e.g., SVM) to perform graph classification [60]. Recently, efficient and expressive graph kernels such as [60], [61] and [62] have been proposed and widely adopted in many application domains (e.g., computer vision [64], chemoinformatics [63], etc.). These kernels have been known to operate in linear-time and have produced accurate results in many real-world applications. Therefore, it just suffices to apply any of these kernels on suitable PRGs and we have an effective, scalable and ready-to-use malware detector. However, as we explain later, these general purpose graph kernels do not take many problem-specific constraints and hence yield suboptimal accuracies.

B. Challenges in ML Based Malware Detection

Almost all ML based Android malware detection techniques (incl. the aforementioned ones) operate in a *batch-learning* setting and use off-the-shelf batch learners like SVM or RFs. Meaning, the detection model is built using a batch of labeled benign and malware samples and is subsequently used to predict whether a given new sample is benign or malicious.

In general, these ML based approaches are typically plagued by four challenges that make them unsuitable for large-scale real-world malware detection:

(C1) Population drift: Though batch-learning based solutions are promising, their success is predicated on an important assumption that may not hold for the malware detection problem. Meaning, *they assume that the malware population (i.e., training data) used to build the detection model does not change over time*. However, malware does not fit this profile. The entire population of malware is constantly evolving due to various reasons such as exploiting new vulnerabilities and evading novel detection techniques. This evolution has a profound impact on malware characteristics and thereby on the features used by these ML models. This makes the collection of malware identified today unrepresentative of the ones generated in the future. This phenomenon is an epitome of *population drift* [34], [51].

(C2) Volume: Since the malware population grows at an alarming rate, a scalable classifier is of paramount importance for real-world malware detection. In order to keep abreast with drifting population, batch learners have to be frequently retrained with huge volumes of data. Hence they pose severe scalability issues when used in the Android malware detection context where we have thousands of apps streaming in every day. Retraining frequently with such a volume renders them computationally impractical.

(C3) Explainability: In general, these ML based solutions just predict the labels of a given sample without offering insights or explanations into how those predictions are arrived at. In other words, they act as *black-box* solutions. However, for malware detection models, understanding the reasons behind their predictions is important in assessing their trustworthiness. This is fundamental if one plans to take action such as deploying a new model or studying malware evolution based on these predictions.

(C4) Expressiveness: PRGs are known to be complex and expressive data structures that characterize topological relationships among program entities. Representing them as vectors or other formats amenable for applying ML algorithms is a non-trivial task [44]. In many cases such representations fail to capture all the vital information from PRGs, thus loosing their expressiveness. For instance, solutions like [17], [18] and [19] capture the topological neighborhood (i.e., structural) information from PRGs and detect security-sensitive behaviors through analyzing them. However, as revealed by recent works such as [15] and [16], an important contextual factor that distinguishes malice is whether or not the user is aware of such behaviors. Unfortunately, the above-mentioned methods which capture structural information well, fail to capture the contextual information and this leads to raising false alarms even when sensitive operations are performed with users' consent. The general purpose graph kernels such as [56]–[62] also suffer with the same drawback.

In summary, challenges C1, C2 and C3 impair all ML based approaches and C4 deters graph based learning approaches.

C. Our Approach

We take these four challenges into consideration and propose CASANDRA (Context-aware, Adaptive and Scalable ANDRoid mAlware detection) framework based on online learning, where we continuously retrain the model upon receiving each labeled sample and make prediction of a new sample using the updated model. CASANDRA is developed with the four following design goals:

1. Accuracy: Accuracy of CASANDRA, which is a PRG based approach depends on how well it retains PRG's expressiveness. In other words, it depends on how well the approach addresses challenge C4. To this end, we leverage on our previous work [11] and use the Contextual Weisfeiler-Lehman kernel (CWLK) that is specifically designed to capture both structural and contextual information from PRGs.

2. Efficiency: CASANDRA achieves its efficiency through the combined use of a scalable graph kernel (i.e., CWLK) and a state-of-the-art online classifier, namely, Confidence Weighted (CW) algorithm [50]. This addresses challenge C2.

3. Adaptiveness: CASANDRA automatically adapts to malware population drift through its use of online classifier and thus addressing challenge C1.

4. Explainability: Since CASANDRA uses a linear classifier along with CWLK which permits explicit feature vector representation of PRGs, we are able to categorically identify the PRG subgraph features which contribute to its predictions. This makes CASANDRA's predictions explainable thus addressing challenge C3.

Experiments: CASANDRA is evaluated through large-scale experiments both on benchmark and a recent real-world dataset of more than 87,000 apps. CASANDRA achieves 89.92% accuracy on the real-world dataset, outperforming two state-of-the-art techniques by more than 25% in their typical batch-learning setting and more than 7% when they are continuously retrained, while maintaining comparable efficiency. Through further experiments, we demonstrate CASANDRA's explainable detection process and its adaptiveness.

Contributions: On one hand, in our recent work [11], we developed CWLK, a graph kernel that is specifically designed to perform malware detection using PRGs.[1] CWLK captures both contextual and structural information, enabling it to achieve high accuracy in a batch learning setting. On the other hand, another recent work of ours, DroidOL [12], demonstrated that online learning based solutions are better suited for large-scale real-world automated malware detection than batch learning methods. However, DroidOL used a general purpose kernel which can only capture structural information from PRGs. In this work, we bring together the best of both the worlds. More specifically, *we combine a kernel which is specifically designed to cater the malware detection task (i.e., CWLK), with a learning paradigm that suits the task extremely well (i.e., online learning).* To the best of our knowledge CASANDRA is the first framework that leverages on both task-specific kernel and online learning to perform malware detection. Besides this, we have made the three following new contributions in CASANDRA:

1) Performing explainable malware detection is an unique feature of CASANDRA. As we demonstrate through our experiments, CASANDRA's explanations are more comprehensive and semantically closer to the malicious behaviors than those of state-of-the-art approaches. Both [11] and [12] were not demonstrated to possess this.
2) Adaptiveness is another important trait of CASANDRA which is not prevalent in any existing approach. We have designed new experiments to thoroughly demonstrate how CASANDRA adapts to malware population drift (see Section V-D). Though DroidOL [12] possessed this quality, it was neither experimentally verified nor demonstrated.
3) We also replaced the online learner used in DroidOL [12] with a more recent state-of-the-art online learner. This resulted in significantly better accuracies.[2]

Organization: The paper is organized as follows: we begin by introducing the background and motivations for our framework's design in Section II. The proposed CASANDRA framework is presented in Section III. The experimental design and implementation details are furnished in Section IV. CASANDRA's evaluation results and discussions are presented in section Section V. Related Work and Conclusions are discussed in Sections VI and VII, respectively.

II. BACKGROUND AND MOTIVATION

In this section, the background on Android malware detection and motivations for the two main components of the CASANDRA framework, namely, CWLK and online learning are presented. Specifically, we discuss the two following motivations: (1) why considering structural information alone from PRGs is insufficient to determine the maliciousness of a sample and how supplementing it with contextual information helps to increase the detection accuracy and (2) why using batch learning is impractical for building a real-world malware detector and how the use of online learning alleviates such practicality issues. For the preliminaries on kernel methods and graph kernels interested readers are referred to Appendix A.

A. Motivations for CWLK

To demonstrate the necessity of CWLK, we use a real-world Android malware from the *Geinimi* family which steals users private information and 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 users 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` APIs. Subsequently, it calls `leak_info_to_url` method to leak the location details (through `DataOutputStream.writeBytes`) to a specific server. The API dependency graph (ADG)[3] corresponding to the code snippet is shown in Fig. 1(b). The nodes in ADG are labeled with the sensitive APIs that they invoke and the edges denote the control-flow dependencies.

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

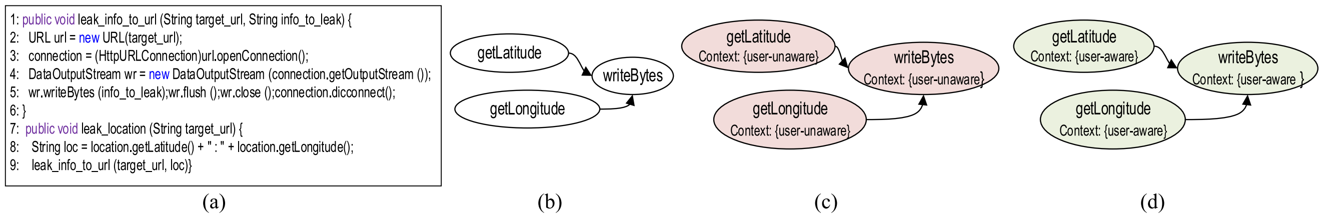
Contextual information: From the explanations above, it is clear that both the apps leak the same information in the same fashion. However, what makes *Geinimi* malicious is the fact that its leak happens without the users consent. In other words, unlike *Yahoo Weather*, *Geinimi* leaks private information through an event which is not triggered by users interaction. We refer to this as a leak happening in *user-unaware* context. On the same lines, we refer to *Yahoo Weathers* leak as happening in *user-aware* context. As explained in [15] and [16], 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 ADG node. This context annotated ADG of *Geinimi* and *Yahoo Weather* are shown in Fig. 1(c) and (d), respectively.

Requirements for effective detection: From the aforementioned example the two key requirements that make a malware detection process effective can be identified:

[1]To the best of our knowledge, CWLK was the first graph kernel specifically addressing a problem from the field of program analysis.

[2]DroidOL outperformed state-of-the-art batch-learning approaches by nearly 20% (without retraining) and 3% (with retraining), whereas, CASANDRA outperform the same by more than 25% (without retraining) and 7% (with retraining) accuracies, respectively. We experimentally verified, this is due to the changes in the choice of the kernel and the classifier.

[3]The detailed procedure for constructing the ADG is provided later in Section III-B.



```
public void leak_info_to_url (String target_url, String info_to_leak) {
  URL url = new URL(target_url);
  connection = (HttpURLConnection)url.openConnection();
  DataOutputStream wr = new DataOutputStream (connection.getOutputStream ());
  wr.writeBytes (info_to_leak);wr.flush ();wr.close ();connection.dicconnect();
}
public void leak_location (String target_url) {
   String loc = location.getLatitude() + " : " + location.getLongitude();
   leak_info_to_url (target_url, loc)}
```

Fig. 1. Location information being leaked in *Geinimi* (malware) and *Yahoo Weather* (benign) apps. (a) code snippet corresponding to leaking location information in both the apps. (b) ADG corresponding to the location leak. (c) *Geinimi's* CADG illustrating that it leaks information without the user's knowledge. (d) *Yahoo Weather's* CADG illustrating that it leaks information with the user's knowledge.

(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. For instance, in the case of *Geinimi*, the privacy leak attack spans across three ADG nodes. Therefore, 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 users' 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 it 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. In summary, this gives us a clear motivation to develop a new kernel that specifically addresses our two-fold requirement.

B. Motivations for Using Online Learning

As foreshadowed in Section I, the three main motivations for using online learning instead of batch learning in CASANDRA are:

(1) handling population drift, (2) handling large volumes of high dimensional data, and (3) performing detection on data that streams-in at real-time. Malware detection-specific justifications on how online learning helps to address these challenges are presented below.

Handling population drift: As reported in [54], the attack and the evasion strategies of malware have evolved significantly, ever since the inception release of Android. Also, one could observe the following pattern in this evolution: *attacks that were popular at some point in time could not sustain forever*. For instance, the *BaseBridge* family of malware leveraged on two root exploits, namely, *RATC* and *Zimperlich* to escalate its privileges [54]. Once the corresponding vulnerabilities were patched and the AV vendors became capable enough to detect such attacks, *BaseBridge's* propagation and sustenance were affected, ultimately resulting in its extinction. This leads to the following observation: *several malware families emerge, flourish and fade-away over time due to various domain-specific reasons*.

From an ML based malware detection viewpoint, this leads to emergence, dominance and disappearance of semantic features that characterize these attacks and evasion strategies. This results in the three following phenomena that happen over time: (1) new features emerge (2) the significance of features vary (3) the cumulative number of features keeps monotonically increasing. This causes the underlying distribution of malware samples to change over time leading to what is known as a *concept drift*. As a result of this drift, the predictions of a static model might become less accurate in due course of time. Therefore, detectors need to adapt to such changes quickly and accurately.

A straightforward solution to this problem is to detect the magnitude and direction of drift in the concept over time and retrain the batch models when the drift is significant [34]. However, this approach would be hampered if: (1) the data arrives in large volumes at a rate too fast to detect, or (2) retraining the models at regular intervals is too expensive. Unfortunately, both these conditions are true in the real-world malware detection setting, as we review in detail below. On the other hand, online learners which are trained on a stream of samples as they arrive are naturally adaptive to evolving distributions mainly due to their learning mechanisms that continuously and efficiently update the model with the most recent examples.

Handling large high-dimensional data: Particularly in the malware detection problem, data is large not only in sample size, but also in feature size. For instance, on the large-scale dataset used in our evaluation, even a light-weight method such as DREBIN [13] extracts and uses more than 1 million semantic features (see Section V-A2). In such big data contexts, the efficiency of a model in terms of both memory and time is of paramount importance in determining its practicality.

However, many batch learners (e.g., SVMs, RFs) typically take multiple passes over the training samples to optimize their parameters and yield the most accurate models within their hypothesis space. Often times, they are optimized to have long and computationally expensive training phases in order to perform accurate and faster during testing. Also, they might require large memory as they hold a batch of samples in memory facilitating them to take multiple passes. On the other hand, online learners take exactly one pass over training data. Meaning, they look at each sample that streams-in only once and learn from them, without storing or iterating over them. This makes the online learners highly efficient in terms of both memory and time. Therefore, online methods offer more practically tractable solutions for malware detection over their batch counterparts.

Handling real-time streaming: Android has a vibrant and growing app ecosystem with a considerable number of third-party markets. Google Play [2], the official market, host more than 2.4 million apps as of this writing. Meaning, nearly 1120

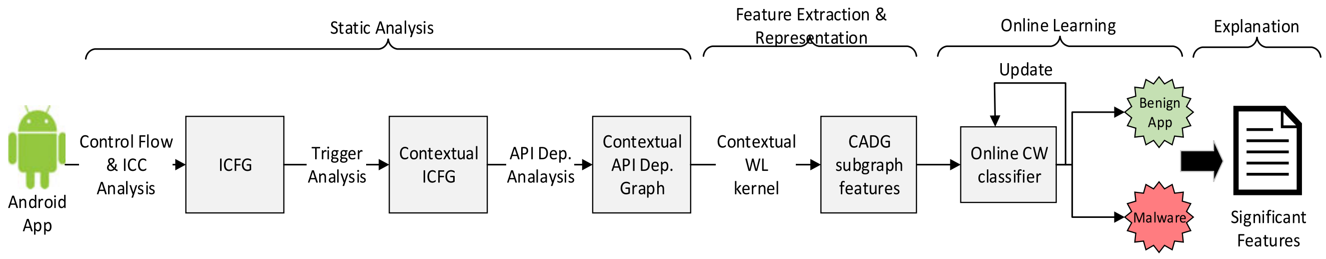


Fig. 2. CASANDRA: Framework overview.

new apps are submitted to Google Play, on average every day. Also, popular third-party markets (e.g., [9]) receive a similar number of new submissions on a daily basis.

Clearly, one important use-case of malware detection solutions like CASANDRA is to deploy them to perform automated vetting of apps submitted to these markets. Therefore, such solutions must be scalable, always up-to-date in terms of their knowledge on attack and evasion strategies and ready for predicting and offering insights into apps as and when they stream in. In other words, these model could not be trained with an outdated set of samples and could not take outages or downtime to retrain themselves. Unlike online methods, the batch learners are susceptible to both these disadvantages, making them unsuitable for handling classification over a continuous, fast and voluminous stream of apps.

III. METHODOLOGY

The methodology of CASANDRA framework designed to perform context-aware, scalable, adaptive and explainable malware detection is presented in this section. We begin by presenting the framework overview and subsequently, we explain each component of the framework.

A. *CASANDRA: Framework Overview*

Fig. 2 depicts CASANDRA framework's overview. The framework has four components as described below.

Static Analysis: Our framework considers subgraphs from Contextual API Dependency Graphs (CADGs) as semantic features to perform malware detection. To this end, we first perform static program analysis to transform the given set of apps into their corresponding CADG representations. This procedure is explained in detail in section Section III-B.

Feature Extraction & Representation: Once the CAGD of the apps in the dataset are constructed, those subgraphs which represent the security-sensitive events that happen in every app along with their context are extracted as using CWLK. We follow a Bag-of-Features (BoF) model [60] to construct the feature vector of individual apps. The detailed procedure is explained in Section III-C.

Online Learning: Once the feature vectors of all the apps in the training set are built, we train a CW classifier with them to detect malware. CW classifiers training and update procedures are explained in detail in Section III-D. Subsequent to this training CASANDRA is ready to perform malware detection at scale. It is important to note that the classifier is trained in an online fashion, meaning whenever a sample is presented, the classifier not only predicts its label but also updates the model based on the actual label of the sample.

Explanation: ML based malware detection solutions are usually *black-box* methods as they do not explain why a particular sample is detected as malicious or benign. In CASANDRA, we address this shortcoming as follows: besides detection CASANDRA reports significant CADG subgraphs of an app that contribute to a detection. In most cases, these significant subgraphs reveal the app's characteristics related to malicious or benign behaviors. The detailed procedure is presented in Section III-E.

B. *Static Analysis*

CASANDRA considers CADG subgraphs as features which encompasses both structural and contextual information characterizing security sensitive operations from apps. We perform a comprehensive static analysis on a given app to construct its CADG. Our static analysis and CADG construction process is similar to that of DroidSIFT [15] and AppContext [16].

The CADG construction workflow as depicted in Fig. 2 involves three steps. Each of them are described below.

Step 1: Inter-procedural Control Flow Graph (ICFG) Construction: The procedure mentioned in [16] is followed to construct the ICFG of a given app. Intuitively, the nodes of ICFG are the instructions in every method. The ICFG edges are the intra- and inter-procedural control flows among these instructions. There two types of interprocedural invocations in Android apps: (1) method invocations and (2) Inter Component Communications (ICCs). Method invocations could be directly inferred through control flow analysis. However, ICCs are not straightforward as these invocations are facilitated by the underlying Android OS framework. In order to model such invocations, we leverage on the IC3 tool [41] and perform a precise ICC analysis.

Step 2: Contextual ICFG (CICFG) Construction: During the course of execution of the program, the ICFG nodes could be reached by actions triggered by entities such as user or system. Trigger events are the external events such as users' interaction with app's user-interface (UI) and system changes such as receipt of SMS that trigger invocation of security-sensitive APIs. Trigger events connect security-sensitive behaviors to their "initiator" in the external environment (e.g., users or system). DroidSIFT [15] proposed a method to analyze the entry point nodes (i.e., PRG nodes that do not have any incoming edge) to identify and categorize the trigger events into UI and NON-UI triggers. We extend this procedure and categorize trigger events of ICFG nodes into the three following types:

1) UI triggers: events triggered by interactions on app's UI (e.g., Clicking UI buttons to make calls, etc.).
2) NON-UI triggers: events initiated by the system state changes (e.g., receipt of SMS, change of signal strength), and events initiated by hardware related actions (e.g., pressing the HOME or BACK button).
3) UNRESOLVED triggers: Entry points of certain sensitive methods are not traceable by our static analysis. For instance, *DroidKungFu*, a popular malware family has its malicious payload triggered by dynamically loaded code. However, similar to [15] and [16], our static analysis cannot model dynamic code loading and reflection based triggers. Therefore, we consider them as behaviors with UNRESOLVED trigger events rather than ignoring them.

Understandably, each ICFG node is triggered by one or more of these triggers. We consider ICFG nodes that are reachable through UI and NON-UI triggers to be in the *user-aware* and *user-unaware* contexts, respectively. Nodes that are reached through UNRESOLVED triggers are considered to be in the '*unresolved*' context. The reachability context(s) of every node is added as a node attribute. This transforms ICFG to CICFG. To formally define CICFG, we adopt and extend the definition of an ICFG from Harrold *et al.* [69] as follows.

Definition 1 (CICFG): $CICFG = (N_i, E_i, \xi)$ of an app a is a directed graph in which each node $n \in N_i$ denotes an instruction in every method of a, and each edge $e(n_1, n_2) \in E_i$ denotes either an intra-procedural control-flow dependence from n_1 to n_2 or a calling relationship from n_1 to n_2. ξ is a set of contexts though which every node $n \in N$ could be reached.

Step 3: CADG construction: The CICFG captures the complete control-flow across every instruction in an app, along with context information. However, only a selected minority of these CICFG nodes will be security-sensitive (i.e., will invoke sensitive APIs). Therefore, once the CICFG of an app is constructed, we abstract it to obtain CADG and narrow down our focus only to its sensitive behaviors. Intuitively, CADG of an app is obtained from its CICFG by considering only the nodes that access security-sensitive APIs[4] along with their context information. All other nodes are removed and paths that exists between such nodes in the CICFG become edges in ADG, if they satisfy certain conditions as described below.

Definition 2 (CADG): A CADG can be represented as a 4-tuple $CADG = (N, E, \lambda, \xi)$, where N is a finite set of nodes and $n \in N$ is an instruction of a method that corresponds to invoking a security-sensitive API and therefore $N \subseteq N_i$. $E \subseteq N \times N$ is a set of directed edges where an edge from $e(n_1, n_2) \in E$ exists iff there exists a path $p(n_1, n_2)$ between these two nodes in the CICFG and $method(n_1) = method(n_2)$, where $method(n)$ denotes the method that encompasses instruction n.[5] λ is the set of labels representing the security-sensitive APIs and $\ell : N \rightarrow \lambda$ is a labeling function which assigns a label to each node. ξ is a set of triggers though which every node in the CADG could be reached and $\mathcal{C} \rightarrow \xi$ is a function which assigns the context to each node.

A portion of *Geinimi* and *Yahoo Weather* examples' CADGs are shown in Fig. 1(c) and (d), respectively. In both these apps, all three CICFG nodes invoke security-sensitive APIs and they are retained in CADGs. All of these nodes belong to the same method (i.e., `leak_info_to_url`) and hence the path that existed among them in CICFG translate to edges in CADG.

C. Feature Extraction & Representation Using CWLK

Once the CADGs are constructed, our next task is to extract semantic features that characterize sensitive behaviors of apps from them. To this end, we use CWLK, a graph kernel we developed in our previous work [11] which is specifically designed to capture both structural and contextual information from PRGs and perform accurate malware detection. This directly addresses the requirements R1 and R2 stated in Section II-A.

We furnish the details on how CWLK supports CASANDRA in extracting CADG subgraph features to perform malware detection, in this subsection. For detailed discussions on CWLK use cases with other PRGs (e.g., ICFG), comparison with WLK and other graph kernels, we refer the reader to [11].

We begin by explaining the regular WLK, then introduce CWLK and finally discuss how WLK falls short when performing malware detection using CADGs and how CWLK rectifies the same.

Weisfeiler-Lehman Kernel: WLK [60], works by decomposing graphs into subgraphs in such a way that a kernel function for a pair of graphs can be defined as a convolution of kernel functions defined over their subgraphs.

WLK computes the similarities between a given pair of graphs $G = (N, E, \lambda)$ and $G' = (N', E', \lambda)$ based on the 1-dimensional WL test of graph isomorphism [60]. The algorithm works by iteratively 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 encompasses the ith degree neighborhood around n. $\lambda_i(n)$ could be optionally compressed using a hash function $f_c : \Sigma^* \rightarrow \Sigma$ such that $f_c(\lambda_i(n)) = f_c(\lambda_i(n'))$, iff $\lambda_i(n) = \lambda_i(n')$. For graph G, this relabeling process yields a WL graph at height i, denoted as $\mathcal{G}_i$. Thus for any given graph G, we could obtain a sequence of WL graphs as defined below.

Definition 3 (WL sequence): Define the WL graph at height i of the graph $G = (N, E, \lambda)$ as the graph $\mathscr{G}_i = (N, E, \lambda_i)$.

[4]Two existing works, namely, PScout [37] and SUSI [38] list commonly known security-sensitive Android APIs. These two lists have been used to perform security and malware analysis by many existing approaches such as [16], [39], [42]. Following these studies, CASANDRA uses these two lists to identify security-sensitive APIs.

[5]This follows from the observation that in most malware the malicious code portion is closely-knit i.e., spanning only to a few methods. We also attempted two other variants of CADG. We reduce the path in CICFG to edges in CADG (i) only if the calling and called nodes belong to the same package and (ii) only if the calling and called nodes belong to the same class. Both these variants contained much larger number of edges in CAGD and also failed to capture the attacks as effectively as the CADG defined above (experimentally verified).

Algorithm 1: CWLK - Contextual relabeling.

```
input  : $G = G_0 = (N, E, \lambda, \xi)$: PRG with set of nodes ($N$),
         set of edges ($E$) and set of node labels ($\lambda$) and
         contexts ($\xi$)
         $h$: number of iterations
output: $\{\mathcal{G}_0, \mathcal{G}_1, ..., \mathcal{G}_h\}$: contextual WL sequence of height $h$
1  begin
2      for $i = 0$ to $h$ do
3          for $n \in N$ do
4              $\sigma_i(n) = \{\}$
5              if $i > 0$ then
                   // neighbourhood (i.e., degree 1 neighbors) of n
6                  $\mathcal{N}(n) = \{m \mid (n, m) \in E\}$
7                  $M_i(n) = \{\lambda_{i-1}(m) \mid m \in \mathcal{N}(n)\}$
                   // neighbourhood label
8                  $\lambda_i(n) = \lambda_{i-1}(n) \oplus sort(M_i(n))$
               // adding context to the neighborhood label
9              for $c \in \xi(n)$ do
10                 $\sigma_i(n) = \sigma_i(n) \cup c \oplus \lambda_i(n)$
               // contextual neighbourhood label
11             $\sigma_i(n) = join(\sigma_i(n))$
               // optional step: label compression
12             $\gamma_i(n) = f_c(\sigma_i(n))$
           // CWL graph at height i
13         $\mathcal{G}_i = (N, E, \gamma_i)$
14     return $\{\mathcal{G}_0, \mathcal{G}_1, ..., \mathcal{G}_h\}$
```

The sequence of graphs

$$\mathscr{G}_0, \mathscr{G}_1, \ldots, \mathscr{G}_h = (N, E, \lambda_0), (N, E, \lambda_1), \ldots, (N, E, \lambda_h) \quad (1)$$

is called the WL sequence up to height h of G, where $\mathscr{G}_0 = G$ (i.e., $\lambda_0 = \lambda$) is the original graph and $\mathscr{G}_1 = r(\mathscr{G}_0)$ is the graph resulting from the first relabeling, and so on.

The WL kernel over graphs G and G' leverages on their respective WL sequences and is defined as follows.

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

$$k^{(h)}_{WL}(G, G') = k(\mathscr{G}_0, \mathscr{G}'_0) + \ldots + k(\mathscr{G}_h, \mathscr{G}'_h) \quad (2)$$

where h is the number of WL iterations and $\mathscr{G}_0, \mathscr{G}_1, \ldots, \mathscr{G}_h$ and $\mathscr{G}'_0, \mathscr{G}'_1, \ldots, \mathscr{G}'_h$ 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 $k^{(h)}_{WL}(G, G') = |(\lambda_i(n), \lambda_i(n'))|$, for $i \in \{0, \ldots, h\}, \forall n \in N, \forall n' \in N'$. Clearly, this enables WLK to capture structural information from CADGs. However, there is no scope for capturing the reachability context information in WLK. This is exactly what we address through our CWLK.

Contextual Weisfeiler-Lehman graph Kernel: The goal of CWLK is to capture both structural and contextual information from PRGs. To this end, we modify the re-labeling step of WLK so as to accommodate the context of every neighborhood. We refer to this process as *contextual-relabeling* and the sequence of graphs thus obtained as *contextual WL sequence*.

Contextual re-labeling: Specifically, CWLK performs one additional step in the re-labeling process which is to attach the contexts of every node to its neighborhood label in every iteration. This in effect, indicates the contexts under which a particular neighborhood is reachable. The label thus obtained is referred to as *contextual neighborhood label*. The contextual relabeling process is presented in detail in Algorithm 1.

Kernel	Geinimi	Yahoo Weather
WLK Feature (h = 0)	{ getLatitude } (a)	{ getLatitude } (b)
WLKFeature (h = 1)	{ getLatitude + writeBytes} (c)	{ getLatitude + writeBytes} (d)
CWLK Feature (h = 0)	user-unaware { getLatitude } (e)	user-aware { getLatitude } (f)
CWLK Feature (h = 1)	user-unaware { getLatitude + writeBytes} (g)	user-aware { getLatitude + writeBytes } (h)

Fig. 3. WLK and CWLK neighborhood labels for the node `getLatitude` from *Geinimi* and *Yahoo Weather* apps.

The inputs to the algorithm are PRG, G and the degree of neighbourhoods to be considered for re-labeling, h. The output is the sequence of contextual WL graphs, $\{\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h\} = \{(N, E, \gamma_0), (N, E, \gamma_1), \ldots, (N, E, \gamma_h)\}$, where $\gamma_0, \ldots, \gamma_h$ are constructed using the contextual relabeling procedure.

For the initial iteration $i = 0$, no neighborhood information needs to be considered. Hence the contextual neighborhood label $\gamma_0(n)$ for all nodes $n \in N$ is obtained by justing prefixing the contexts to the original node labels and optionally compressing the same (lines 9-12). For $i > 0$, the following procedure is used for contextual re-labeling. Firstly, for a node $n \in N$, all of its neighboring nodes are obtained and stored in $\mathcal{N}(n)$ (line 6). For each neighbor node $m \in \mathcal{N}(n)$ the neighborhood label up to degree $i - 1$ is obtained and stored in multiset $M_i(n)$ (line 7). $\lambda_{i-1}(n)$, the neighborhood label of n till degree $i-1$ is concatenated to the sorted value of $M_i(n)$ to obtain the current neighborhood label, $\lambda_i(n)$ (line 8). Finally, $\lambda_i(n)$ is prefixed with the contexts of node n to obtain $\sigma_i(n)$, which denotes the uncompressed contextual neighborhood label (lines 9-11). $\sigma_i(n)$ is then optionally compressed using the function f_c to obtain the contextual neighborhood label, $\gamma_i(n)$ (line 12). For every iteration i, this process of contextual relabeling yields CWL graph at height i, $\mathcal{G}_i$ (line 13). Finally, the CWL sequence comprising CWL graphs from height 0 to h are returned (line 14).

Definition 5 (CWL kernel): Given a valid kernel $k(.,.)$ and the CWL sequence of graph of a pair of graphs G and G', the contextual WL graph kernel with h iterations is defined as

$$k^{(h)}_{WL}(G, G') = k(\mathcal{G}_0, \mathcal{G}'_0) + \ldots + k(\mathcal{G}_h, \mathcal{G}'_h) \quad (3)$$

where h is the number of CWL iterations and $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_h$ and $\mathcal{G}'_0, \mathcal{G}'_1, \ldots, \mathcal{G}'_h$ are the CWL sequences of G and G', respectively.

Intuitively, CWLK counts the common contextual neighborhood labels in two graphs. Hence we have $k^{(h)}_{CWL}(G, G') = |(\gamma_i(n), \gamma_i(n'))|$, for $i \in \{0, \ldots, h\}, \forall n \in N, \forall n' \in N'$.

Illustrating WLK's shortcoming and CWLK's efficacy: Having presented the formulations for both kernels, we now illustrate WLK's shortcomings and explain how CWLK addresses the same, with an example. For the ease of illustration, the label compression step is avoided. Applying WLK on the CADGs of *Geinimi* and *Yahoo Weather* examples (see Fig. 1(c) and (d)), for the node `getLatitude`, for heights $h = 0$, 1, we get the neighborhood labels as shown in Fig. 3(a)–(d). In both cases, the node `getLatitude` has only one degree-1 neighbor, `writeBytes` and this fact is reflected in the neighborhood label. Clearly, WLK captures the structural information around the node `getLatitude`, incrementally in every iteration of h. In fact, neighborhood label for $h = 1$ captures that a sensitive node, `writeBytes` lies in the neighborhood of `getLatitude`, which highlights a possible privacy leak. However, WLK does not capture whether the neighborhood involved in this leak is reached in *user-aware* or *unaware* context. Hence, whether or not the leak is malicious still remains inconclusive. This is precisely what CWLK addresses. On applying CWLK for the same node in both apps, we obtain the contextual neighborhood labels as shown in Fig. 3(e)–(h). Clearly, the contextual neighborhood labels of *Geinimi* reveal that the sensitive operations are performed in the *user-unaware* context, whereas, the same for *Yahoo Weather* happen in the *user-aware* context. Hence, it is evident that the CWLKs contextual relabeling provides a means to clearly distinguish malicious CADG neighborhoods from the benign ones. Therefore, unlike WLK, CWLK based classification does not detect *Yahoo Weather* as a false positive. This example clearly establishes the suitability of CWLK for the malware detection task.

CWLK validity, complexity and representation: CWLK's positive definiteness is asserted in Appendix B. The runtime complexity of CWLK with h iterations on a graph with n nodes and e edges is $O(he)$ which is same as that of WLK. For derivation and discussions on time complexity, see Appendix C. Similar to WLK, CWLK also supports explicit feature vector representation of PRGs following the Bag-of-Features (BoF) model. The process of obtaining such representations is explained in Appendix D.

D. Online Learning

Once the feature vectors of all the apps in the training-set are built using CWLK, we train an online CW [50] classifier with them to detect malware. CW classifier's training and update procedures are as explained below with relevant notations.

Denote the features of an app (both benign and malware) as a vector $x = [x^{(1)}, x^{(2)}, \ldots, x^{(d)}]^T$, and its label as $y \in \{-1, +1\}$, where -1 indicates benign and $+1$ indicates malicious apps. The CW classifier receives a number of samples, x_i, and their labels, y_i, and trains using this labeled data. Given a new unseen sample, x, the goal of CW classifier is to predict the label, y, of this new sample based on its trained model.

CW classifier fits a linear decision boundary (i.e., hyperplane) between the positive and negative class samples. That is, the model is a weight vector, $w = [w^{(1)}, w^{(2)}, \ldots, w^{(d)}]^T$ which indicates the weight (i.e., relative importance) of each of the features used to predict the output label y. The predicted label, $\hat{y}$, is the sign of the inner product between x and w:

$$\hat{y} = sign(x \cdot w) \tag{4}$$

CW incrementally builds the model in (4) in rounds. In round t, it receives a sample, x_t and predicts its label $\hat{y_t}$ using the current model; it then receives y_t, the true label of x_t and updates its model based on the sample-label pair: (x_t, y_t). A Gaussian distribution over the weights with mean μ and covariance matrix Σ is maintained by the CW algorithm. The value $\mu^{(f)}$ represents mean weight of feature f (i.e., mean of $w^{(f)}$), and the value $\Sigma_{f,f}$ captures the confidence in f's weight. While classifying a new sample x, the weight w is drawn from $N(\mu, \Sigma)$. In practice, one could choose $w = \mu$, the average weight vector and use (4) to arrive at the predicted label. CW updates the model, (i.e., μ and Σ), continuously on every labeled sample instead of only when committing misclassifications. The rationale is that *making correct prediction also suggests that the model should increase its confidence of the current weights*. CW's update rule is presented below:

$$(\mu_{t+1}, \Sigma_{t+1}) = arg \min_{\mu, \Sigma} D_{KL}(\mathcal{N}(\mu, \Sigma) || \mathcal{N}(\mu_t, \Sigma_t)), \tag{5}$$

$$\text{s.t. } Pr_{w \sim N(\mu, \Sigma)}[y_t(w \cdot x_t)] \geq \eta. \tag{6}$$

Equation (5) determines that the new distribution characterized by new Σ and μ should be close to the old distribution as much as possible. The KL divergence (D_{KL}) provides the measure of distance between the two distributions. Equation (6) determines that the update should ensure that the probability of making correct prediction if the same sample, x_t is seen in the next round must be bigger than η, where η is a configurable parameter (usually set bigger than 50%). The computational complexity of the update is linear in the number of non-zero features in x_t.

The strength of CW lies in its notion of modeling confidence on features' weights. Evidently, if the weight of a feature does not fluctuate very much over time, one could confidently believe that this weight is what it should be. CW achieves the two following desirable characteristics through this confidence notion, which are not exhibited by other online learners (e.g., Online Perceptron (OP) [47], Passive Aggressive (PA) [49]):

1. The weights of more confident features are updated less aggressively. For instance, using `sendSms` API in the *user-unaware* context is a good indicator of an app's maliciousness; consequently, its weight does not get updated abruptly over time, thereby instigating high confidence on this feature. Therefore, CW ensures that the weight will not change much even when it receives an instance of benign app using *user-unaware*{`sendSms`}, which possibly could be a case of incorrect labeling. This makes CW naturally robust to labeling noise.
2. The model is updated just enough to adapt to the changing trends in the data while refraining from changing too much. The rationale is that the previous model carries valuable information about the data and should not be modified too abruptly.

Overall, compared to primitive online learner such as PA, the CW classifier makes a fine-grained distinction between each features weight confidence. Hence, it is specifically more suited to detect malware apps, as our data stream continually introduces a dynamic mix of new and recurring CADG subgraph features. Once the CW classifier in CASANDRA is trained with all the samples it is ready to perform malware detection at scale.

E. Explaining CASANDRA's Predictions

Once CASANDRA classifies a sample as malicious or benign, the next step is to offer explanations of these predictions by reporting significant CADG subgraphs of an app that contribute to prediction. These explanations help to understand whether the rationale behind the model's predictions are reasonable, thus ensuring users' trust on the model. Besides, they could help human analysts in several tasks such as studying malware attacks and creating malware signatures.

Explaining the predictions of linear classifiers is a well-studied problem [13], [18], [68]. Following them, we are able to determine the contribution of each CADG subgraphs feature to the final class prediction as described below.

Based on (4) in section Section III-D, for a given sample x, during the prediction of $\hat{y}$, the largest ν weights $w^{(s)}$ which are significantly important for placing the sample on the malicious (or benign) side of the decision boundary are identified (i.e., a point-wise multiplication $wx = w \times x$ is performed and ν largest values are obtained from the resulting vector wx). Since each weight $w^{(s)}$ is assigned to a certain feature $x^{(s)}$, it is then possible to explain why an app has been classified as malicious (or benign). After extracting the top ν CADG subgraph features, CASANDRA reports them explaining the significant semantic characteristics that make a particular sample malicious (or benign). As we could observe from our evaluations, the frequently observed malware features include CADG subgraphs representing the use of reflection, dynamic and native code, accessing private information, sending of SMS and communication over internet predominantly in the *user-unaware* or *unresolved* context.

IV. Experimental Design & Implementation

We conducted several large-scale experiments to evaluate CASANDRA's accuracy, efficiency, adaptiveness and explainability which are its primary design goals. We also compare its performance against two state-of-the-art malware detection approaches. In this section, experimental design aspects such as research questions (RQs) addressed, datasets used, evaluation setup and metrics are presented along with implementation details.

A. Research Questions

We intend to address the following RQs through our evaluations:

(RQ1 Accuracy) How accurate is CASANDRA *in detecting unseen malware from both benchmark and real-world datasets?*

The impact of the two most important factors responsible for CASANDRA's accuracy are investigated separately through two following sub-RQs.

(RQ1.1) What impact does capturing contextual information through CWLK have on CASANDRA's accuracy?

(RQ1.2) Does CASANDRA's online learning provide any benefit over batch learning in terms of accuracy?

(RQ2 Efficiency): How efficient is CASANDRA over batch learning based solutions in terms of overall training and prediction time?

(RQ3 Explainability): How explainable are CASANDRA s predictions?

(RQ4 Adaptiveness): How adaptive is CASANDRA when performing malware detection in the real-world setting? In other words, does it adapt to malware population drift seamlessly?

TABLE I
IN-THE-WILD DATASET WITH APPS DATED FROM JAN. 1 2014 TO AUG. 13 2014

Market	# of Benign Apps	# of Malware
Google Play [2]	39156	26178
Anzhi [3]	2957	12260
AppChina [4]	1845	4154
SlideMe [5]	289	132
HiApk [6]	65	157
FDroid [7]	29	2
Angeeks [8]	6	27

B. Datasets

Many existing solutions such as [13], [15], [16], [42] are evaluated only using benchmark datasets. We observe that malware in benchmark datasets are much easier to detect compared to the ones found in-the-wild (as reported in [14], [36]). Hence we evaluate CASANDRA on both benchmark and a large collection of recent real-world malware collected in-the-wild, so as to exhibit its potential as a practical real-world malware detection solution. The details of these two datasets are presented below.

Benchmark (BM) dataset: DREBIN [13] provides a popular benchmark dataset comprising 5560 malware samples belonging to 179 families. This dataset is used in our evaluation. The date of creation of these apps fall in the range: from Mar'09 to Oct'12. The DREBIN collection forms the malware portion of the dataset and for the benign portion, we used 5000 randomly chosen benign apps from Google Play [2] that are created in the same period.

In-the-wild (ITW) dataset: We collected a large dataset of 87,257 apps from seven different markets in 2014. Following the common practice in software security research [13], [14], [18], we leveraged on the VirusTotal web portal[6] which hosts malware detection services from more than 40 Anti-virus (AV) scanners to determine the ground-truth labels of these apps. To infer whether an app is malicious or benign, we upload it to the VirusTotal service and inspect the output of all its AV scanners (Kaspersky, AVG, etc.). We consider an app as malicious, if at least two scanners flags it as such. Apps detected as adware are removed, as they could not be entirely considered benign or malicious [13]. Through this labeling process, we infer that the ITW dataset contained 44,347 benign and 42,910 malware apps. Table I provides the information on the distribution of apps in this dataset along with names of the markets where they have been crawled from. The date of creation of these apps fall in a span of 224 days starting from 1 Jan'14 to 13 Aug'14.

C. Experimental Setup

All the experiments were conducted on a server with 36 cores of Intel(R) Xeon CPU E5-2699 v3 @ 2.30 GHz and 32 GB RAM running Ubuntu 14.04.

D. Implementation and Comparative Analysis

CASANDRA is implemented in approximately 17200 lines of Python and Java code. IccTA [39] an Android static analysis

[6]https://www.virustotal.com

workbench based on Soot [10] is used for building CADGs from apps.

Comparison with state-of-the-art solutions: CASANDRA is compared against two state-of-the-art Android malware detection solutions, namely, DREBIN [13] and Allix *et al.* [14]. To this end, we re-implemented these two approaches. Since the malware detection accuracy of these solutions predominantly depend on the features they use, we briefly introduce them here.

Drebin [13] is well-known for its scalable and explainable detection. It extracts light-weight semantic features such as APIs and permissions used, URLs accessed, names of components from apps and subsequently, trains a linear SVM classifier to distinguish malware from benign apps.

Allix *et al.* [14] recently proposed another scalable approach using structural features, namely Control Flow Graph (CFG) signatures. Therefore, we refer to this technique as CFG-Signature Based Detection (CSBD) in the remainder of the paper. CSBD constructs CFGs of individual methods and encodes them as text-signatures. Subsequently, a RF classifier is trained with these signatures to detect malware.

E. Evaluation Metrics

Standard evaluation metrics such as Precision, Recall, F-measure and cumulative error rates are used to determine the effectiveness of malware detection. Efficiency is determined in terms of time required to train and test the classifiers.

V. EVALUATION

The evaluations, results and discussions pertaining to each of the RQs are presented in this section.

A. RQ1: Accuracy

The results for CWLK's and online learning's impact on CASANDRA's accuracy are presented in the two following subsections.

1) (RQ1.1) Impact of CWLK: In order to determine whether incorporating both structural and contextual information improves CASANDRA's accuracy, we study the two following scenarios: (1) use only the structural information from CADGs, (2) use both structural and contextual information from CADGs to perform malware detection. Understandably, the former scenario is realized through using WLK and the latter is realized through CWLK. Hence in this experiment, we compare the accuracies of these two kernels on our task.

We experimented with different kernel heights, $h = 0$ to 5 for CWLK and WLK (see (2) and (3)). The average number of (contextual) neighborhood features are found to increase exponentially with increasing values of h as long as $h \leq 2$. However, this number does not increase significantly after $h = 2$. This is because we have removed nodes that do not access sensitive APIs, which affects the connectivity and restricts CADG neighborhood sizes. Hence we restrict the height h to be 0, 1 and 2 for both the kernels. Thus, in our experiments, for each kernel, we have three malware detection models (one for each value of h).

Dataset & Experiments: The BM dataset is used in this experiment. 70% of the samples (chosen at random) are used to train and the remaining 30% samples are used to test the models' 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. Since we wish to exclude the impact of online learning on the models' accuracy, we use a canonical batch kernel classifier, namely SVM with both the kernels. The models are referred as CWLK+SVM and WLK+SVM, denoting the kernel and the classifier, respectively. In order to study how CWLK features compare to state-of-the-art solutions, DREBIN [13] and CSBD [14] are included in this comparative analysis. The precision, recall and F-measures of these models are compared.

Results & Discussion: These results for the 8 malware detection models under comparison are presented in Table II. The following inferences are drawn from the table:

1) At the outset, for both CWLK and WLK, 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 both the 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.
2) It is clear that CWLK outperforms WLK for significant values of h (i.e., 1 and 2), in terms of F-measure. *Since the only difference between WLK and CWLK is the latters capability to capture the context information, evidently, this is the reason for CWLK's superior performance*.
3) Also, CWLK achieves better Precision than WLK for all values of h, consistently. This indicates that CWLK suffers lesser false positives than WLK. This reduction is a direct result of capturing context information which helps to precisely distinguish malicious CADG neighborhoods from the benign ones.
4) Since CWLK uses both contextual and structural information, it is important to analyze the contribution of each of these types of information to its performance. Capturing only structural information is equivalent to using WLK. Hence from columns 1 to 3 of Table II, it is evident that structural information alone could provide a minimum of 96.18% and an average of 97.54% F-measure. Similarly, the contribution of contextual information alone is ascertained using CWLK and setting $h = 0$ to be 95.73% F-measure. Finally, the effect of using both types of information is ascertained by using CWLK and setting $h > 0$ to be a minimum of 98.47% and an average of 98.85% F-measure. This clearly conveys that structural information is primary for performing effective malware detection and contextual information complements it, thereby helping to improve the accuracy.
5) When comparing CWLK to the state-of-the-art solutions, CWLK (with $h = 2$) outperforms all the compared solutions in terms of F-measure and Precision. In particular,

TABLE II
COMPARISON OF CWLK AGAINST STATE-OF-THE-ART KERNEL AND MALWARE DETECTION MODELS - AVG. (± STD) OVER 5 RUNS

	WLK + SVM (h = 0)	WLK + SVM (h = 1)	WLK + SVM (h = 2)	CWLK + SVM (h = 0)	CWLK + SVM (h = 1)	CWLK + SVM (h = 2)	Drebin [13]	CSBD [14]
P	97.11(±0.35)	98.50(±0.22)	99.21(±0.08)	98.86(±0.26)	99.07(±0.09)	**99.56**(±0.04)	99.01(±0.20)	98.61(±0.46)
R	95.27(±1.21)	97.11(±0.51)	98.08(±0.32)	92.79(±0.98)	97.88(±0.47)	98.90(±0.28)	**99.15**(±0.08)	99.13(±0.12)
F	96.18(±0.63)	97.80(±0.16)	98.64(±0.29)	95.73(±1.05)	98.47(±0.17)	**99.23**(±0.11)	99.08(±0.09)	98.87(±0.25)

it outperforms the second best performing technique (i.e., DREBIN) by 0.15% F-measure. In terms of Recall, it outperforms CSBD and is comparable to DREBIN.

The results and discussions above demonstrate CWLK's suitability for the malware detection problem. Hence, we consider using CWLK in all the remaining experiments unless mentioned otherwise.

2) *(RQ1.2) Impact of Online learning:* With the inference on CWLK's impact arrived at, we now intend to evaluate the benefit of using online over batch learning for malware detection.

Dataset & Experiment: In this experiment, we use the ITW dataset as it is collected in the wild and larger, thus offering more scope for illustrating malware population drift. We divide the ITW dataset apps (both benign and malware) into batches according to their date of creation. The resulting time-line based distribution of the two datasets are presented in Fig. 7 (in Appendix E). We have 224 batches, one for each day of the ITW collection.

Specifically, in this experiment, we intend to illustrate the benefits that CASANDRA attains due to its use of online learning. To study this impact, we replaced the online learner in CASANDRA with a canonical batch learner, namely, SVM and compare the same with CASANDRA under different experimental settings (which involve periodic retraining). The above-mentioned SVM variants use the same features as CASANDRA and hence it is sufficient to compare online and batch learning paradigms. Furthermore, in order to compare CASANDRA to state-of-the-art solutions, like in the previous RQ, we include DREBIN and CSBD (which use different set of features) in this comparative analysis.

Batch Learning Configurations: For batch-learning solutions, the classifiers are trained/retrained in a sliding window fashion over the 224 batches as explained below. For SVM, we experiment with the following variants: SVM-Once, SVM-Daily, SVM-MultiOnce, and SVM-MultiDaily. For SVM-Once, SVM classifier is trained only once on the apps from Day 1 (1 Jan14). This model is tested on all other days without retaining. For SVM-Daily, the classifier is retrained after every day; however, only one previous days samples are used for every retraining e.g., 11 Jan14 results reflect training on the apps created on 10 Jan14, and testing on 11 Jan14 apps. SVM-MultiOnce is similar to SVM-Once and SVM-MultiDaily is similar to SVM-Daily, however, with the size of the batch for training and retraining covers 10 days instead of 1. In summary, for Once and Multi-Once variants, the model is never retrained and for Daily and Multi-Daily, the model is retrained in a sliding window fashion over the batches of data. The size of MultiDaily training sets is determined to be 10-day batches based on the data that our evaluation machine can handle.

Casandra's online training regimen: Since CASANDRAs feature extraction using CWLK is based on BoF model, its number of features grows as the samples stream in. Fig. 8 (in Appendix F) shows the cumulative number of features for each day of the evaluation in ITW dataset, representing the feature space growth. Each days total includes new features introduced that day and all the old features from previous days. We obtained a total of 15,171 features from the samples encountered on Day 1 (1 Jan'14). The dimensionality grows quickly as we extract new subgraph features from samples encountered every day and while reaching the final day (13 Aug'14), we have accumulated 2,114,050 features. This phenomenon of feature space growth is common across many techniques including DREBIN and CSBD. This is because apps evolve over time for various reasons such as capability enhancements, bug fixes and adapting to changes in Android framework APIs [13], [34], [36]. This evolution results in newly observed characteristics which translate into new features from an ML view-point.

Now, leveraging on the inferences from our previous work [12], we refrain from using only a subset of these features (e.g., using only the 15,171 features encounter on Day 1). Instead, we allow the dimensionality of our CW classifier to grow with the number of new features encountered; on the last day (13 Aug'14), for instance, we classify with more than 2.1 million features. Implicitly, samples that were introduced before a feature *i* was first encountered will have value 0 for feature *i*. This training regimens helps our CW classifier stay abreast of changing trends in malware and benign apps features.

Results & Discussions: Fig. 4(a)–(c) shows the cumulative error rates of CASANDRA in comparison to the aforementioned variants of SVM, DREBIN and CSBD. The following observations are made from the figure:

1) Updating the detection models over time is essential to detect new malware as shown by SVM-MultiDaily and SVM-Daily outperforming SVM-MultiOnce and SVM-Once, respectively.
2) Training on significantly more data improves the performance, as illustrated by SVM-MultiDaily and SVM-MultiOnce outperforming SVM-Daily and SVM-Once, respectively. However, it is noted that there is a fundamental limit on the amount of data a batch-learning technique could train on because of the storage, memory and time requirements. Thus, among the SVM variants we have considered, SVM-MultiDaily achieves the best accuracy, as it has both the advantages of being trained frequently and trained with large volumes of data.
3) From Fig. 4(a), one could see that CASANDRA consistently outperforms all the batch-learning SVM variants. In

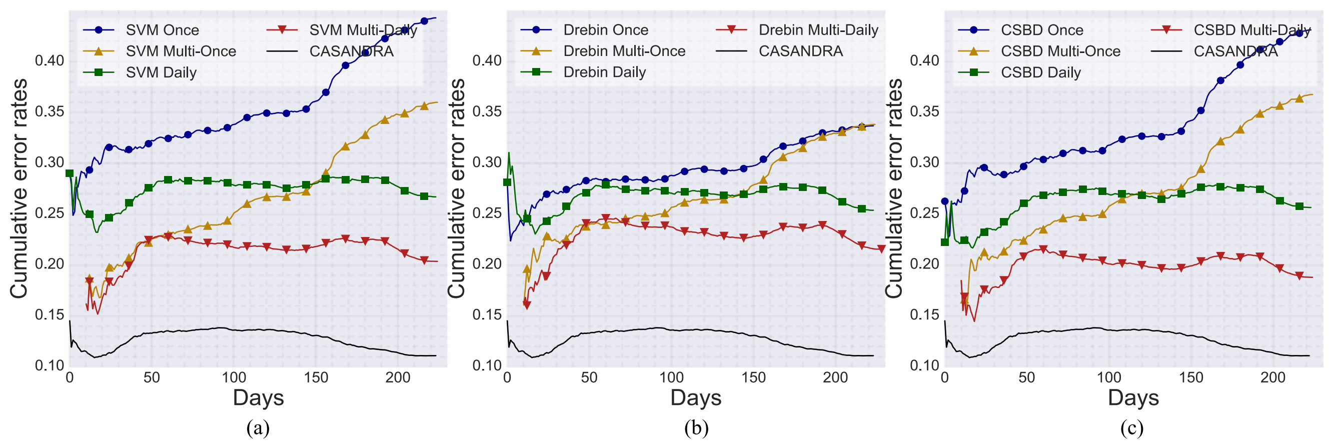


Fig. 4. CASANDRA: : Cumulative error rates for CASANDRA vs. batch learning algorithms (a) CASANDRA Vs. four variants of SVM, (b) CASANDRA Vs. four variants of DREBIN, (c) CASANDRA Vs. four variants of CSBD.

particular, when the SVM models are not retrained, CASANDRA outperforms SVM-Once and SVM-MultiOnce by more than 32% and 25%, respectively. When the SVM models are retrained on a daily basis, CASANDRA still outperforms SVM-Daily and SVM-MultiDaily by more than 16% and 9%, respectively. This is because CASANDRA is able to adapt to the changes in the malware characteristics instantaneously as well as retain significant useful information from the past.

4) In the case of comparison with DREBIN and CSBD, the trends in the performance of all four variants of these methods are similar to those of the SVM variants. Hence the observations on frequently updating the models and training with more data, hold.
5) From Fig. 4(b) and (c), one could see CASANDRA consistently outperforming the best performing variants of state-of-the-art methods. Particularly, it outperforms Drebin-MultiDaily by 10.61% and CSBD-MultiDaily by 7.89%. This reaffirms the suitability of online learning solutions over retrained batch learners for practical large-scale malware detection.

B. RQ2: Efficiency

In this RQ, we investigate the efficiency of CASANDRA in terms of training and testing durations. Specifically, we study whether CASANDRA achieves superior accuracy at the cost of very high (practically inviable) training or test durations.

Dataset & Experiment: The ITW dataset which involves several thousands apps, instigating a vocabulary of over 2 million features for CASANDRA, 1 million features for DREBIN, offers enough opportunities to challenge these techniques in terms of efficiency. Hence we consider ITW dataset for this RQ. The experimental setting mentioned in RQ1.2 (Section V-A2) are reused in this RQ as well.

A typical batch learning test cycle will involve only feature extraction, representation and prediction on each test set sample. However, in the online learning setting the model will predict and at the same time learn from samples that stream-in. The initial online model is often built with a batch of trivial number of samples and keeps updating itself from the samples that stream-in. We emulate this scenario, as we use the batch of samples that stream-in on the first day of ITW dataset (i.e., 1 Jan'14) to build the model. The samples that arrive thereafter are considered as stream of samples on which CASANDRA predicts the label and also learns from.

TABLE III
COMPARING CASANDRA'S EFFICIENCY AGAINST THAT OF STATE-OF-THE-ART METHODS

Method	Training Duration (avg. ± std.) in sec.	Testing Duration (avg. ± std.) in sec.
DrebinOnce	**0.0096**	**0.0004** (± 0.00)
DrebinMultiOnce	0.0352	0.0006 (± 0.00)
DrebinDaily	0.4493 (± 0.08)	0.0010 (± 0.00)
DrebinMultiDaily	0.5873 (± 0.27)	0.0010 (± 0.00)
CSBDOnce	0.1849	0.0354 (± 0.01)
CSBDMultiOnce	0.5858	0.0594 (± 0.03)
CSBDDaily	11.0698 (± 5.38)	0.0605 (± 0.02)
CSBDMultiDaily	14.5157 (± 11.40)	0.0641 (± 0.03)
CASANDRA	0.0131	0.0011 (± 0.00)

Results & Discussions: The training and the testing durations of CASANDRA and the 4 variants of state-of-the-art solutions across all the 224 evaluation days in the ITW dataset are presented in Table III. In the case of batch learning models, the training durations of variants that are retrained every day (i.e., Daily and MultiDaily) are distributions of values (one for each day). For the variants that are not retrained (i.e., Once and MultiOnce) the training durations are single values, as the training happens only once. In the case of testing durations, each model undergoes testing cycle every day and hence this produces a series of testing durations.

The following observations are made from Table III:

1) Understandably, the batch learning variants that are trained with smaller sets consume lesser training duration than their counterparts trained with larger sets. For instance, the training durations of Drebin-Once and CSBD-Once are more than 3 times shorter than those of their respective MultiOnce counterparts. Similar trend is observed when we compare the variants that are retrained.

That is, on average, both Drebin-Daily and CSBD-Daily are trained 1.3 times faster than their respective MultiDaily counterparts. This shows that the training duration of the batch learning models increase exponentially with the training set size. For obvious reasons, the testing durations of Once and Daily variants are similar to those of their Multi counterparts for both techniques.

2) It is well-known the choice of the classifier plays a pivotal in determining the training and testing durations of the models. Linear models like Linear SVM and CW are simpler and could be trained much faster than quasi-linear models like RFs. This fact is evident from our findings in the aforementioned figure. All the variants of DREBIN and CASANDRA (which use linear models) are significantly faster than their CSBD counterparts (which use RF classifiers) in terms of both training and testing durations.

3) Finally, we intend to compare CASANDRA's efficiency with those of best performing variants of the state-of-the-art methods (i.e., Drebin-MultiDaily and CSDB-MultiDaily). The training duration of CASANDRA is 44 and 1108 times lesser than those of DREBIN and CSBD MultiDaily variants, respectively. *This clearly illustrates that retraining the models at specific intervals are impractical and much less efficient than online learning in the real-world setting*. In terms of testing durations, CASANDRA is nearly 58 times faster than CSBD-MultiDaily and as fast as Drebin-MultiDaily. In summary, on the ITW dataset, CASANDRA takes 28.23 microseconds on average to predict the class of a given sample.

C. RQ3: Explainablity

Apart from its detection performance the strength of CASANDRA lies in its ability to offer interpretations of the obtained results. We verify this quality of CASANDRA in this RQ.

Dataset & Experiment: In this experiment, we intend to rank the features from a given malware sample that maximally influence CASANDRA's classification decision and investigate if they reflect the malicious behavior of the sample. If they do so, we can conclude that those features explain the sample's behavior the best. Understandably, the ground truth on the malware behavior could be ascertained by knowing the family to which it belongs. For instance, a sample belonging to the *FakeInstaller* family is expected to send premium-rated SMS as this behavior is a part of its attack vector. Therefore, clearly, the availability of malware family labels is indispensable for this experiment. Out of our two datasets, only the BM dataset contains malware family labels. Hence in this RQ, we experiment with two popular malware families from the BM dataset and analyze how CASANDRA features with high weights allow conclusions to be drawn about their behavior.

As in the previous RQs, we intend to compare the explainability of CASANDRA with that of DREBIN and CSBD. However, CSBD uses features (i.e., CFG signatures) that are humanly uninterpretable (see [14, Sec. 4.1] for details) and hence we refrain from including it in this RQ's comparative analysis.

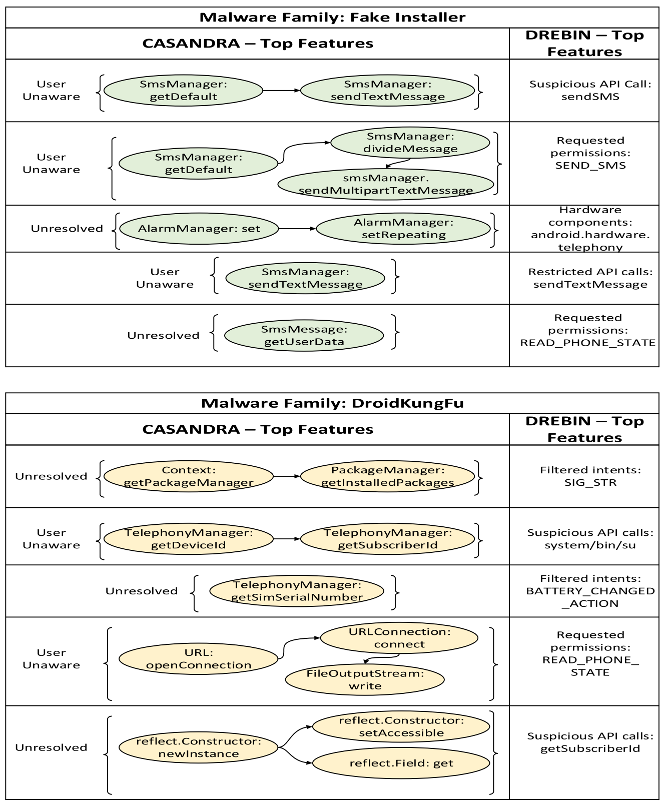

Malware Family: Fake Installer
CASANDRA – Top Features
DREBIN – Top Features
User Unaware
SmsManager: getDefault
SmsManager: sendTextMessage
Suspicious API Call: sendSMS
User Unaware
SmsManager: getDefault
SmsManager: divideMessage
smsManager. sendMultipartTextMessage
Requested permissions: SEND_SMS
Unresolved
AlarmManager: set
AlarmManager: setRepeating
Hardware components: android.hardware. telephony
User Unaware
SmsManager: sendTextMessage
Restricted API calls: sendTextMessage
Unresolved
SmsMessage: getUserData
Requested permissions: READ_PHONE_STATE
Malware Family: DroidKungFu
CASANDRA – Top Features
DREBIN – Top Features
Unresolved
Context: getPackageManager
PackageManager: getInstalledPackages
Filtered intents: SIG_STR
User Unaware
TelephonyManager: getDeviceId
TelephonyManager: getSubscriberId
Suspicious API calls: system/bin/su
Unresolved
TelephonyManager: getSimSerialNumber
Filtered intents: BATTERY_CHANGED _ACTION
User Unaware
URL: openConnection
URLConnection: connect
FileOutputStream: write
Requested permissions: READ_PHONE_ STATE
Unresolved
reflect.Constructor: newInstance
reflect.Constructor: setAccessible
reflect.Field: get
Suspicious API calls: getSubscriberId

Fig. 5. CASANDRA Vs. DREBIN: Comparing explainability on two malware families from the BM dataset.

Results & Discussion: To study the most significant features that influence the predictions of the CASANDRA and DREBIN, we consider two well-known malware families, namely *FakeInstaller* and *DroidKungFu* from the BM dataset. For each sample of these families we determine the features with the highest contribution to the classification decision and average the results over all members of a family. The resulting top five features for each malware family for both the techniques are shown in Fig. 5.

From the figures the following inferences are drawn for each family:

FakeInstaller malware samples send premium-rated SMS to specific numbers without users' consent. It is evident that features identified by CASANDRA not only highlight neighborhoods with operations related to SMS, but also reflect that they are triggered predominantly in the *user-unaware* or *unresolved* context. These operations include getting the default SMS manager, sending normal and multi-part SMS. Also, from the third top feature, one could see that *FakeInstaller* triggers these operations using `AlarmManager` APIs which are invoked in the background without the users' knowledge. On the other hand, DREBIN's explanations are simpler and naive. For instance, as DREBIN claims simply sending SMS does not make an app malicious. However, doing the same without the users' consent does. This distinction is not evident from DREBIN's explanations.

DroidKungFu is a sophisticated command-and-control (C&C) based family of malware capable of exploiting several vulnerabilities in earlier version of Android to gain root access and steal sensitive data from the device. Its intention to leak a variety of private information such as unique identifiers (e.g., IMEI, IMSI) and contents from content provider (e.g., contacts) to the C&C server over internet is adequately revealed by CASANDRA's features. Also, this malware reads system state changes such as `SIG_STR` through respective intents and use them to trigger malicious services in the background. Again, compared to DREBIN, CASANDRA's explanations are closer and more descriptive of the secretive operations of *DroidKungFu*. Interestingly, the fact that *DroidKungFu* leverages on *Java reflection* to obfuscate its attack is evident from CASANDRA's explanations.

D. RQ4: Adaptiveness

In this RQ, we intend to study how CASANDRA adapts itself to malware evolution and population drift. Specifically, we intend to explore how online learning helps CASANDRA to learn fresh patterns and unlearn obsolete patterns of malicious behavior over time. As stated earlier, all the existing Android malware detection approaches (incl. DREBIN [13] and CSBD [14]) are based on batch learning and are not capable of adapting themselves to population drift. Adaptiveness is CASANDRAs unique feature and hence we could not compare this with any existing technique. Hence, we just illustrate how CASANDRA achieves adaptiveness in this subsection.

Dataset & Experiment: In this RQ we prefer the BM dataset for three reasons: (1) it hosts malware collection over a longer period which is ideal to study malware evolution (2) its heterogeneity it contains malware with attacks as simple as sending premium-rated SMS to complex botnets, (3) availability of accurate malware family labels.

In this experiment, the apps in BM dataset are sorted according to their month of creation. We observe that several malware families in this dataset emerge, flourish and fade-away over time due to various domain-specific reasons. For example, *DroidKungFu* family first emerged in May11 and reached its peak spread during Sep-Oct11 and gradually disappeared by May12.

Any good malware detector must adapt to such evolution. To examine whether CASANDRA exhibits this adaptiveness, we train it in an online fashion on the BM dataset following a procedure similar to RQ1.2 (Section V-A2). The training and prediction quantum changes from days in RQ1.2 to months in this RQ.

Once the prediction and learning for all the samples in a month m is over, we record the weights of all the features. Since BM dataset hosts malware collection over 44 months, the weights of each feature are recorded at 44 points in time. This gives us an opportunity to monitor and track the fluctuations in the importances of individual features over time. In general, features that characterize typical malicious and benign behaviors will be assigned large positive and negative weights, respectively. Features that do not reveal anything about malice or benignity will be assigned near-zero weights.

For a detector that adapts well to the evolving trend in malware population, the feature weights should follow the pattern of population drift. More specifically, when a particular family of malware $\mathcal{F}$ gains prominence over a period, say from month m to m', the weights of features that characterize the attacks and evasion strategies of $\mathcal{F}$ should increase or remain high. Once the samples from family $\mathcal{F}$ start to dwindle, the weights of those features should decrease or remain low.

Results & Discussion: In order to study this behavior, we chose three malware families that perform privacy leak attacks, namely, *DroidKungFu*, *Ginmaster*, and *ADRD*. The behaviors of *DroidKungFu* and *Ginmaster* are explained in the previous RQ. *ADRD* also secretively leaks users' private data in a similar fashion. As shown in Fig. 6, in the BM dataset, *ADRD* first emerged in May'10 and gained prominence from Oct'10 to Dec'10. It finally vanished from Mar'11. A very small number of *ADRD* samples resurfaced in Jun'11, Nov'11, and Jul'12. The evolution of *DroidKungFu* and *Ginmaster* populations could be inferred in a similar fashion from Fig. 6. Now, leveraging on CASANDRA's explainability, we choose 10 features that best explain the privacy leaks from these families and study fluctuations in their weights over time. These fluctuations are also shown in Fig. 6.

One could clearly see that the feature weight fluctuations follow the evolution of the corresponding families. For instance, from Mar'09 to Apr'10 many of these features had no significant positive weights as none of these populous privacy leak families emerged. Some features had near-zero weights indicating that they do not characterize any malice at that point in time. After May'10, a small number of features that characterize *ADRD* start to gain positive weights. After the voluminous influx of *DroidKungFu* in May'11, almost all the features started to gain larger positive weights. In particular, one could see the following interesting patterns:

1) Feature f_2 typically characterizes the *ADRD* family. The weights of this feature surge rapidly during *ADRD*'s peak spread period (i.e., Oct-Dec'10).
2) Features related to reflection (i.e., f_4 and f_8) and inferring installed apps (i.e., f_3 and f_9) are common for both *DroidKungFu* and *Ginmaster*. These features receive significant increase in their weights once *DroidKungFu* emerges in Jun'11. Even after *DroidKungFu* vanishes, these features continue to accumulate more weights as *Ginmaster* samples keep streaming till Sep'12.
3) The weights of features that characterize common malicious behaviors across many families such as URL features (i.e., f_1 and f_{10}) remain predominantly high throughout the entire duration.
4) Similar subgraph features exhibit very close or exactly same weight fluctuation patterns. For instance, the two `PackageManager` features (i.e., f_3 and f_9) and two URL features (i.e., f_1 and f_{10}) have exactly same pattern over the entire duration. This reflects the fact that these features characterize either same or very similar attacks.
5) Finally, when we reach the end of the dataset collection period in Sep'12, we could see CASANDRA has assigned large positive weights to all the 10 features. Meaning, it has learned that all of them characterize malicious behaviors. In particular, CASANDRA has learnt that features such as `reflection` and `PackageManager` APIs (f_3 and f_9) characterize strong malicious behaviors than other

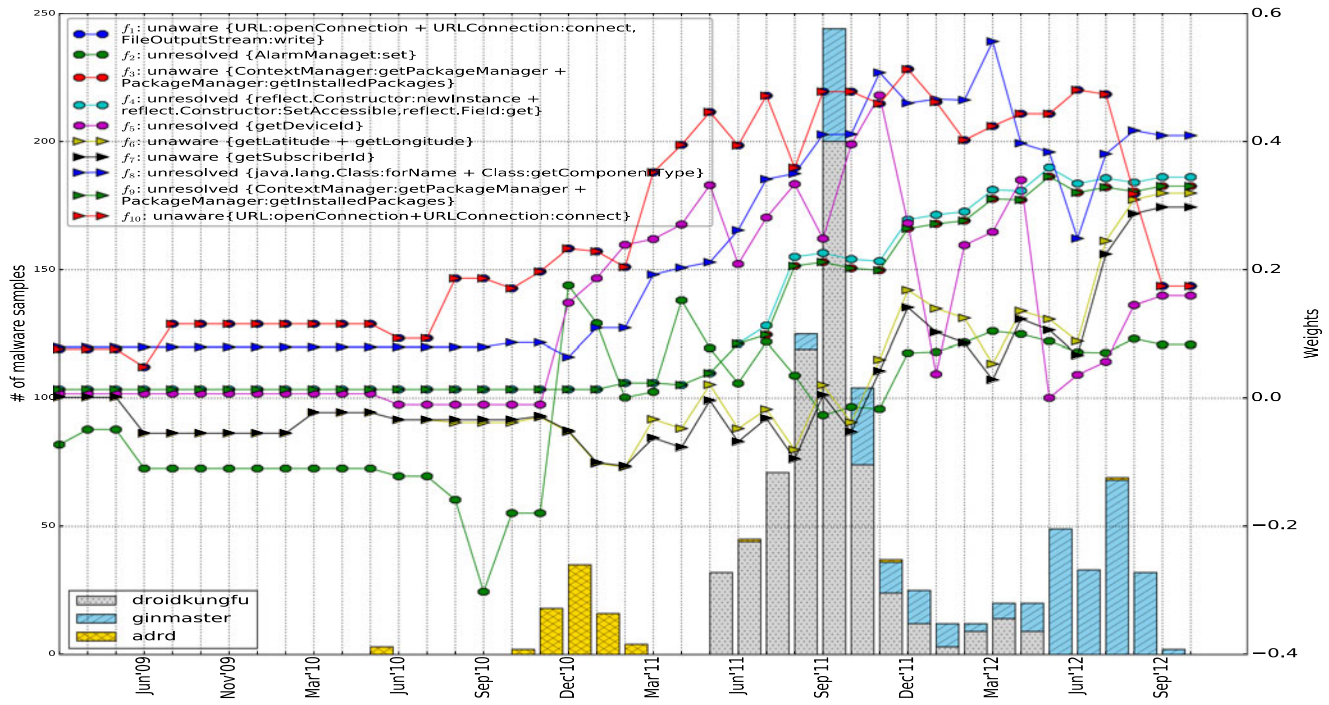


Fig. 6. Illustration of how CASANDRA adapts to the malware evolution and population drift in the BM dataset. The X axis shows the distribution of family-specific malware samples. Y axis (left) shows the number of samples and Y axis (right) shows the weights of individual features. The legend contains CWLK subgraph features (format: < feature id >: < context > {< root-node > + < neighbor-list >}).

features such as reading device ID (f_5), which may be common in benign apps as well. This is reflected from the fact that it has assigned larger positive weights to the former set of features than the latter.

In summary, we could clearly see that CASANDRA when trained in an online fashion, adapted well to the population drift and evolution in malware. So far, none of the existing techniques (incl. DREBIN and CSBD) demonstrated this capability. We believe adaptiveness is the cornerstone for the practical success of ML based malware detector.

VI. RELATED WORK

Many well-known malware detection solutions have been reviewed in the previous sections. We throw light on remaining works and contrast them from CASANDRA in this subsection. Features used for detection is the most important aspect of ML based malware detectors. Hence, we discuss the related work with respect to this aspect.

Crowdroid [43] leverages on dynamic analysis and uses Linux system-call sequences as features. DroidAPIMiner [24] considers a hand-crafted set of sensitive APIs (along with their parameters) and package level information as features. CHABADA [26] leverages on text-mining techniques to study the relevance between apps behavior and their description and henceforth detect suspiciously behaving apps. Sahs and Khan [22] extract a variety of features including permissions and CFG signatures and perform early-fusion of those features. With these features, they take an anomaly detection approach using a One-Class SVM to detect malware. ADAGIO [18] constructs apps' call-graphs (CGs) and uses byte-code instructions to assign labels to nodes. Subsequently, it captures structural information from CGs using NHGK and uses kernel SVM to detect malice.

Recently, several techniques such as MARVIN [27], HADM [30] and StormDroid [31] adopt a hybrid approach by combining both static and dynamic analysis features and demonstrated achieving better accuracies than solutions leveraging on one of the analysis paradigms.

However, all these solutions are based on batch-learning and they neither account for concept drift in Android malware nor exhibit adaptiveness. On the contrary, CASANDRA uses online learning and is naturally adaptive to concept drift. Moreover, none of the above-mentioned techniques exhibit all the four qualities that CASANDRA possess (i.e., accurate context-aware detection, efficiency, explainability and adaptiveness). The only other work that discusses concept drift in Android malware is Prescience [34]. It detects drifts in malware concept using Venn-Abers predictors and retrains computationally heavy models such as XGBoot and ExtraTrees, periodically. Though, this retraining approach addresses concept drift, as we showed in Section V-B it is significantly less efficient than using online learning.

VII. CONCLUSION & FUTURE WORK

In this paper, we present CASANDRA, an online learning based Android malware detection framework. CWLK, a novel graph kernel that facilitates capturing apps' security-sensitive behaviors along with their context information from dependency graphs is also proposed. CWLK supports explicit feature vector representation of apps' dependency graphs using which an online classifier is trained to detect malware. Our large-scale evaluations on both recent real-world and benchmark datasets demonstrate that CASANDRA outperforms two state-of-the-art techniques. CASANDRA achieves 89.92% accuracy on a real-world dataset with more than 87,000 apps outperforming state-of-the-art techniques by more than 25% in their typical batch-learning setting. On average, CASANDRA takes 28.23 microseconds to predict the label of a given sample in our large-scale experiment, which is comparable to state-of-the-art techniques, making it scalable enough to perform market-scale analysis. This superior performance and scalability make CASANDRA, in particular, and online learning based solutions, in general, better candidates for the malware detection task.

Future Work: We plan to investigate three specific directions in our future work:

1) Taking into account the recent developments in the area of Deep Graph Kernels (e.g., [65], [66]), which show potentials to learn latent sub-structures from graphs to achieve better accuracy, we intend to explore on the deep learning variant of CWLK in our future work.
2) We plan to incorporate contextual information in other sub-structure based graph kernels such as NHGK [61] and NSPDK [62] and subsequently, investigate their suitability for malware detection.
3) API dependencies used in CASANDRA represent only one perspective (i.e., view) of the apps. However, as revealed by recent works such as MADAM [28] and RevealDroid [29] capturing other views such as *information flows*, and *permission dependencies* leads to more comprehensive detection. Inspired by this inference, we intend to extract multiple features sets from PRGs and systematically integrate them using Online Multiple Kernel Learning methods [71], for performing a more comprehensive malware detection.

Dataset Release: To allow easy reproduction and verification of our work, we provide all the datasets used within this work for download at: https://sites.google.com/view/casandrantu.

APPENDIX A
PRIMER ON KERNEL METHODS & GRAPH KERNELS

Kernel methods have been highly successful in solving a specific class of problems where feature vector representations of samples are not readily obtainable. Malware detection using PRGs is one of such problems. For many well-known classifiers, the data samples have to be explicitly represented as feature vectors through a user-specified feature map ϕ. In contrast, kernel methods require only a user-specified kernel k, i.e., a similarity function over pairs of samples in their native representations. Kernel methods work by mapping the samples into a feature space, implicitly and finding an appropriate decision boundary in the new feature space. Here, feature map $\phi(\cdot)$ is realized through the kernel function k, which facilitates computing inner products in the feature space using the samples in their native representation, i.e., $k(x_i, x_j) = \langle \phi(x_i), \phi(x_j) \rangle$.

Graph Kernels: Formally, a graph kernel $k : \mathbb{G} \times \mathbb{G} \rightarrow R$ is a kernel function defined on a domain of graphs, $\mathbb{G}$. Hence, given a labeled graph dataset $D_g = (g_1, y_1), \ldots, (g_n, y_n)$ and a graph kernel k, a kernel classifier (e.g., SVM) can be directly used to perform graph classification. Graph kernels usually belong to the family of *R convolution kernels*. These kernels decompose graphs into sub-structures such as paths, walks etc. The comparison of two graphs is then based on the similarity between all pairs of such sub-structures. Several graph kernels have been proposed based on this idea [56]–[58], [60]–[62].

Explicit vs. Implicit Feature mapping: Existing graph kernels can be classified into approaches that use explicit feature mapping (ϕ) and those that directly compute a kernel function (i.e., ϕ is not necessarily known and may be of infinite dimension). Examples of former category include WLK [60] and NHGK [61], and that of latter category are RW [56] and SP [58] kernels. Kernels that support explicit feature mapping have two advantages that make them particularly suitable for the malware detection problem:

1) Scalability: If explicit representations are manageable, these approaches usually outperform other kernels regarding runtime on large datasets, since the number of vector representations scales linear with the dataset size.

2) Explainability: These kernels support extracting sub-structures of graphs as features and building a vocabulary of such features. This facilitates building explicit feature vector representation of individual graphs. This aspect makes this category of kernels amenable for performing explainable malware detection.

Two explicit feature mapping kernels, namely, WLK [60] and NHGK [61] have been successfully used for Android malware detection in [22] and [18] respectively. Moreover, [18] performed explainable detection leveraging on the explicit feature map produced by NHGK.

APPENDIX B
CWLK POSITIVE SEMI-DEFINITENESS

Theorem 1: CWLK is positive definite.

Proof: Let us define a mapping ϕ that counts the occurrences of a particular contextual neighborhood label sequence σ[7] in G (generated in h iterations of Algorithm 1). Let $\phi_\sigma^{(h)}(G)$ denote the number of occurrences of σ in G, and analogously $\phi_\sigma^{(h)}(G')$ for G'. Then,

$$k_\sigma^{(h)}(G, G') = \phi_\sigma^{(h)}(G), \phi_\sigma^{(h)}(G') = |\{(\sigma_i(n), \sigma_i(n'))| \\ \sigma_i(n) = \sigma_i(n'),\ i \in \{0, \ldots, h\},\ n \in N,\ n' \in N'\}| \tag{7}$$

Summing over all σ from the vocabulary Σ^*, we get

$$\begin{aligned} k_{CWL}^{(h)}(G, G') &= \sum_{\sigma \in \Sigma^*} k_\sigma^{(h)}(G, G') = \sum_{\sigma \in \Sigma^*} \phi_\sigma^{(h)}(G) \phi_\sigma^{(h)}(G') \\ &= |\{(\sigma_i(n), \sigma_i(n')) \mid \sigma_i(n) = \sigma_i(n'), \\ &\quad i \in \{0, \ldots, h\},\ n \in N,\ n' \in N'\}| \\ &= |\{(\sigma_i(n), \sigma_i(n')) \mid f_c(\sigma_i(n)) = f_c(\sigma_i(n')), \\ &\quad i \in \{0, \ldots, h\},\ n \in N,\ n' \in N'\}| \end{aligned} \tag{8}$$

[7]The label compression step (to obtain γ from σ) is avoided for the sake for clarity.

where the last equality follows from the fact that f_c is injective.

As $f_c(\sigma) \neq f_c(\sigma')$ if $\sigma \neq \sigma'$, the string σ corresponds to exactly one contextual neighborhood label and $k_{CWL}^{(h)}$ defines a kernel with corresponding feature map $\phi_{CWL}^{(h)}$, such that

$$\phi_{CWL}^{(h)} = (\phi_{\sigma}^{(h)}(G))_{\sigma \in \Sigma^*} \tag{9}$$

Appendix C CWLK Time 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 6-8 of Algorithm 1) take $O(e)$ time for one iteration and the same for h iterations take $O(he)$. The inclusion of context (lines 9 and 10), does not incur additional overhead as $e >> |\xi|$. 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 [60].

Appendix D Explicit Feature Vector Representations of PRGs Using CWLK

When computing CWLK on K graphs to obtain $K \times K$ kernel matrix, a nave approach would involve K^2 comparisons, resulting a time complexity of $O(K^2 he)$. However, as mentioned in [60], a Bag-of-Features (BoF) model based optimization could be performed to arrive the kernel matrix in $O(Khe + K^2 hn)$ time. This optimized computation involves the following steps:

1) A vocabulary Σ of all the contextual neighbourhood 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 $|\Sigma|$ dimensions.
2) Subsequently, $K \times K$ kernel matrix can be computed by multiplying these vectors in $O(K^2 hn)$ time.

In summary, while CWLK has the same efficiency as WLK, it supports semantically richer feature vector representations of PRGs.

Appendix E Timeline Based Distribution of Apps in ITW Dataset

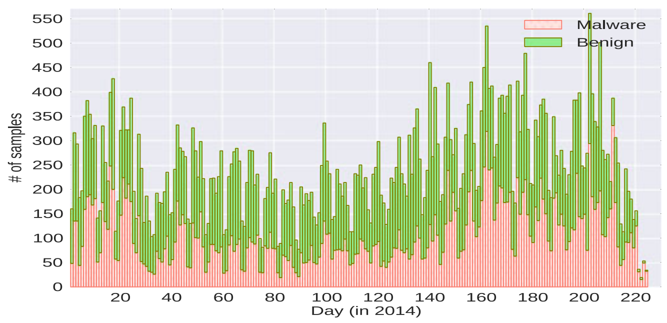


Fig. 7. Timeline based distribution of apps in our large-scale ITW dataset (with malware and benign apps proportions).

Appendix F Feature Space Growth

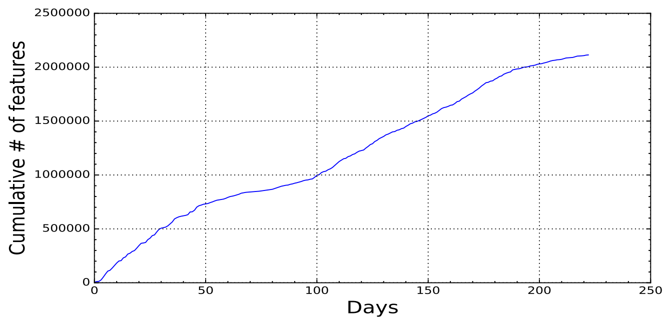


Fig. 8. Cumulative number of features observed over time for our ITW dataset.

References

[1] *Symantec 2016 Threat Report*. [Online]. Available: https://www.symantec.com/content/dam/symantec/docs/reports/istr-21–2016-en.pdf
[2] *Google Play*. [Online]. Available: https://play.google.com/store
[3] *Anzhi third-party market*. [Online]. Available: www.anzhi.com
[4] *AppChina third-party market*. [Online]. Available: www.appchina.com
[5] *SlideMe third-party market*. [Online]. Available: www.SlideME.org
[6] *HiApk third-party market*. [Online]. Available: www.hiapk.com
[7] *FDroid third-party market*. [Online]. Available: www.fdroid.org
[8] *Angeeks third-party market*. [Online]. Available: http://apk.angeeks.com
[9] *Amazon app store URL*. [Online]. Available: https://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011
[10] *Soot framework*. [Online]. Available: http://sable.github.io/soot
[11] A. Narayanan *et al.*, "Contextual Weisfeiler-Lehman graph kernel for malware detection," in *Proc. Int. Joint Conf. Neural Netw.*, 2016, pp. 4701–4708.
[12] A. Narayanan *et al.*, "Adaptive and scalable android malware detection through online learning," in *Proc. Int. Joint Conf. Neural Netw.*, 2016, pp. 2484–2491.
[13] D. Arp *et al.*, "Drebin: Effective and explainable detection of android malware in your pocket," in *Proc. Annu. Symp. Netw. Distrib. Syst. Secur.*, 2014.
[14] K. Allix *et al.*, "Empirical assessment of machine learning-based malware detectors for Android," *Empirical Softw. Eng.*, vol. 21, no. 1, pp. 183–211, 2016.
[15] M. Zhang *et al.*, "Semantics-aware android Malware classification using weighted contextual API dependency graphs," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2014, pp. 1105–1116.
[16] W. Yang *et al.*, "Appcontext: Differentiating malicious and benign mobile app behaviors using context," in *Proc. 37th Int. Conf. Softw. Eng.*, 2015, pp. 303–313.
[17] C. Yang *et al.*, "Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2014, 163–182.
[18] H. Gascon *et al.*, "Structural detection of android malware using embedded call graphs," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 45–54.
[19] K. Chen *et al.*, "Finding unknown malice in 10 seconds: Mass vetting for new threats at the Google-Play scale," in *Proc. 24th USENIX Conf. Secur. Symp.*, 2015, pp. 659–674.
[20] W. Peng *et al.*, "ACTS: Extracting android App topological signature through graphlet sampling," in *Proc. IEEE Conf. Commun. Netw. Secur.*, 2016, pp. 37–45.
[21] F. Martinelli *et al.*, "Classifying android malware through subgraph mining," in *Data Privacy Manage. Auton. Spontaneous Secur.*, 2014, pp. 268–283.
[22] J. Sahs and L. Khan, "A machine learning approach to android malware detection," in *Proc. Eur. Intell. Secur. Informat. Conf.*, 2012, pp. 141–147.
[23] S. Chakradeo *et al.*, "MAST: Triage for market-scale mobile malware analysis," in *Proc. 6th ACM Conf. Secur. Privacy Wireless Mobile Netw.*, 2013, pp. 13–24.

[24] Y. Aafer *et al.*, “DroidAPIMiner: Mining API-level features for robust malware detection in Android,” in *Proc. Int. Conf. Secur. Privacy Commun. Syst.*, 2013, pp. 86–103.
[25] N. Peiravian and X. Zhu, “Machine learning for android malware detection using permission and API calls,” in *Proc. IEEE 25th Int. Conf. Tools Artif. Intell.*, 2013, pp. 300–305.
[26] A. Gorla *et al.*, “Checking app behavior against app descriptions,” in *Proc. 36th Int. Conf. Softw. Eng.*, 2014, pp. 1025–1035.
[27] M. Lindorfer *et al.*, “MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis,” in *Proc. IEEE 39th Annu. Comput. Softw. Appl. Conf.*, 2015, vol. 2, pp. 422–433.
[28] A. Saracino *et al.*, “MADAM: Effective and efficient behavior-based android malware detection and prevention,” *IEEE Trans. Depend. Sec. Comput.*, vol. PP, no. 99, p. 1, 2016.
[29] J. Garcia *et al.*, “Obfuscation-resilient, efficient, and accurate detection and family identification of android malware,” Department of Computer Science, , George Mason University, Dept. Comput. Sci., , George Mason Univ., Fairfax, VA, USA, Tech. Rep., 2015.
[30] L. Xu *et al.*, “HADM: Hybrid analysis for detection of malware,” SAI Intelligent Systems Conference (IntelliSys). London, UK. 2016.
[31] S. Chen *et al.*, “StormDroid: A streaminglized machine learning-based system for detecting android malware,” in *Proc. 11th ACM Asia Conf. Comput. Commun. Secur.*, 2016, pp. 377–388.
[32] S. K. Dash *et al.*, “DroidScribe: Classifying android malware based on runtime behavior,” *Mobile Secur. Technol.*, vol. 7148, pp. 1–12, 2016.
[33] G. Suarez-Tangil *et al.*, “DroidSieve: Fast and accurate classification of obfuscated android malware,” in *Proc. 7th ACM Conf. Data Appl. Secur. Privacy*, 2017, pp. 309–320.
[34] A. Deo *et al.*, “Prescience: Probabilistic guidance on the retraining conundrum for malware detection” in *Proc. ACM Workshop Artif. Intell. Secur.*, 2016, pp. 71–82.
[35] M. Xu *et al.*, “Toward engineering a secure android ecosystem: A survey of existing techniques,” *ACM Comput. Surveys*, vol. 49.2, 2016, Art. no. 38.
[36] S. Roy *et al.*, “Experimental study with real-world data for android app security analysis using machine learning,” in *Proc. 31st Annu. Comput. Secur. Appl. Conf.*, 2015, pp. 81–90.
[37] K. W. Y. Au *et al.*, “PScout: Analyzing the android permission specification,” in *Proc. ACM Conf. Comput. Commun. Secur.*, 2012, pp. 217–228.
[38] S. Arzt *et al.*, “SuSi: A tool for the fully automated classification and categorization of android sources and sinks,” in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014.
[39] L. Li *et al.*, “IccTa: Detecting inter-component privacy leaks in android apps,” in *Proc. 37th Int. Conf. Softw. Eng.-Vol. 1*, 2015, pp. 280–291.
[40] D. Octeau *et al.*, “Effective inter-component communication mapping in android: An essential step towards holistic security analysis,” in *Proc. 22nd USENIX Secur. Symp.*, 2013, pp. 543–558.
[41] D. Octeau *et al.*, “Composite constant propagation: Application to android inter-component communication analysis,” in *Proc. 37th Int. Conf. Softw. Eng.-Vol. 1*, 2015, pp. 77–88.
[42] V. Avdiienko *et al.*, “Mining apps for abnormal usage of sensitive data,” in *Proc. IEEE/ACM 37th IEEE Int. Conf. Softw. Eng.*, 2015, pp. 426–436.
[43] I. Burguera *et al.*, “Crowdroid: Behavior-based malware detection system for android,” in *Proc. 1st ACM Workshop Secur. Privacy Smartphones Mobile Devices*, 2011, pp. 15–26.
[44] M. Fredrikson *et al.*, “Synthesizing near-optimal malware specifications from suspicious behaviors,” in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 45–60.
[45] A. Blum, *On-Line Algorithms in Machine Learning*. Berlin, Germany: Springer, 1998.
[46] J. Ma *et al.*, “Identifying suspicious URLs: An application of large-scale online learning,” in *Proc. 26th Annu. Int. Conf. Mach. Learning*, 2009, pp. 681–688.
[47] F. Rosenblatt, “The perceptron: A probabilistic model for information storage and organization in the brain,” *Psychological Rev.*, vol. 65, no. 6, pp. 386–408, 1958.
[48] T. Zhang, “Solving large scale linear prediction problems using stochastic gradient descent algorithms,” in *Proc. 21st Int. Conf. Mach. Learning*, 2004, p. 116.
[49] K. Crammer *et al.*, “Online passive-aggressive algorithms,” *J. Mach. Learn. Res.*, vol. 7, pp. 551–585, 2006.
[50] M. Dredze *et al.*, “Confidence-weighted linear classification,” in *Proc. 25th Int. Conf. Mach. Learn.*, 2008, pp. 264–271.
[51] A. Singh *et al.*, “Tracking concept drift in malware families,” in *Proc. 5th ACM Workshop Secur. Artif. Intell.*, 2012, pp. 81–92.
[52] A. Kantchelian *et al.*, “Approaches to adversarial drift,” in *Proc. ACM Workshop Artif. Intell. Secur.*, 2013, pp. 99–110.
[53] M. M. Masud *et al.*, “Cloud-based malware detection for evolving data streams,” *ACM Trans. Manage. Inf. Syst.*, vol. 2, no. 3, 2011, Art. no. 16.
[54] Y. Zhou and X. Jiang, “Dissecting android malware: Characterization and evolution,” in *Proc. IEEE Symp. Secur. Privacy*, 2012, pp. 95–109.
[55] S. V. N. Vishwanathan *et al.*, “Graph kernels,” *J. Mach. Learn. Res.*, vol. 11, pp. 1201–1242, Apr. 2010.
[56] T. Grtner *et al.*, “On graph kernels: Hardness results and efficient alternatives,” in *Learning Theory and Kernel Machines*. Berlin, Germany: Springer, 2003, pp. 129–143.
[57] U. Kang *et al.*, “Fast random walk graph kernel,” in *Proc. 2012 SIAM Int. Conf. Data Mining*, 2012, pp. 828–838.
[58] K. M. Borgwardt and H.-P. Kriegel, “Shortest-path kernels on graphs,” in *Proc. 5th IEEE Int. Conf. Data Mining*, 2005, pp. 74–81.
[59] N. Shervashidze *et al.*, “Efficient graphlet kernels for large graph comparison,” in *Proc. 12th Int. Conf. Artif. Intell. Statist.*, 2009, vol. 5, pp. 488–495.
[60] N. Shervashidze *et al.*, “Weisfeiler-Lehman graph kernels,” *J. Mach. Learn. Res.*, vol. 12, pp. 2539–2561, 2011.
[61] S. Hido and H. Kashima, “A linear-time graph kernel,” in *Proc. 9th IEEE Int. Conf. Data Mining*, 2009, pp. 179–188.
[62] F. Costa and K. De Grave, “Fast neighborhood subgraph pairwise distance kernel,” in *Proc. 26th Int. Conf. Mach. Learn.*, 2010, pp. 255–262.
[63] P. Mah *et al.*, “Extensions of marginalized graph kernels,” in *Proc. 21st Int. Conf. Mach. Learn.*, 2004, p. 70.
[64] Z. Harchaoui and F. Bach, “Image classification with segmentation graph kernels,” in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit.*, 2007, pp. 1–8.
[65] P. Yanardag and S. Vishwanathan, “Deep graph kernels,” in *Proc. 21th ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2015, pp. 1365–1374.
[66] A. Narayanan *et al.*, “subgraph2vec: Learning distributed representations of rooted sub-graphs from large graphs,” in *Proc. Workshop Mining Learn. Graphs*, 2016.
[67] J. Friedman, T. Hastie, and R. Tibshirani, *The Elements of Statistical Learning* (Springer Series in Statistics), vol. 1. Berlin, Germany: Springer, 2001.
[68] M. T. Ribeiro *et al.*, “Why should i trust you?: Explaining the predictions of any classifier,” in *Proc. 22nd ACM SIGKDD Int. Conf. Knowl. Discovery Data Mining*, 2016, pp. 1135–1144.
[69] M. J. Harrold *et al.*, “Computation of interprocedural control dependence,” *ACM SIGSOFT Softw. Eng. Notes*, vol. 23, no. 2, pp. 11–20, 1998.
[70] B. Biggio *et al.*, “Poisoning behavioral malware clustering,” in *Proc. Workshop Artif. Intell. Secur. Workshop*, 2014, pp. 27–36.
[71] S. C. H. Hoi *et al.*, “Online multiple kernel classification,” *Mach. Learn.*, vol. 90, no. 2, pp. 289–316, 2013.

Annamalai Narayanan received the B. Tech. degree in information technology from Anna University, Chennai, India, in 2008 and the M.Sc. degree in communication software and networks from Nanyang Technological University (NTU), Singapore, in 2011. He is currently working toward the Ph.D. degree with the School of Electrical and Electronic Engineering, NTU, Singapore. His current research interests include software security and machine learning.

Mahinthan Chandramohan received the B.Eng. degree in computer engineering from Nanyang Technological University (NTU), Singapore, in 2011. He is currently working toward the Ph.D. degree with the School of Computer Science and Engineering, NTU, Singapore. His research interests include program analysis, malware analysis, vulnerability detection, and machine learning with applications in software security.

Lihui Chen (SM'07) received the B.Eng. degree in computer science and engineering from Zhejiang University, Hangzhou, China, and the Ph.D. degree in computational science from the University of St Andrews, St Andrews, U.K. She is currently an Associate Professor with the Centre for Infocomm Technology, School of Electrical and Electronic Engineering, Nanyang Technological University, Singapore. She has authored more than 100 referred papers in international journals and conferences in her research areas. Her current research interests include machine learning algorithms and applications, data mining, and data analytics.

Yang Liu received the Ph.D. degree in computer science from the National University of Singapore (NUS), Singapore, in 2010 and continued with his postdoctoral work in NUS. Since 2012, he joined Nanyang Technological University as an Assistant Professor. His research focuses on software engineering, formal methods and security. Particularly, he specialises in software verification using model checking techniques. This work led to the development of a state-of-the-art model checker, Process Analysis Toolkit.