Received December 13, 2019, accepted January 7, 2020, date of publication January 13, 2020, date of current version January 27, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.2965954

CrowdNet: Identifying Large-Scale Malicious Attacks Over Android Kernel Structures

XINNING WANG[1,2], CHONG LI[1], AND DALEI SONG[1]
[1]School of Engineering, Ocean University of China, Qingdao 266100, China
[2]Department of Computer Science and Software Engineering, Auburn University, Auburn, AL 36849, USA

Corresponding author: Dalei Song (songdalei@ouc.edu.cn)

This work was supported by Fundamental Research Funds for the Central Universities under Grant 201962012.

ABSTRACT While malicious attacks in Android devices are growing, machine learning-based malware prediction has become time-consuming and space-consuming. Open-source parallel frameworks for massive data processing can efficiently deal with iterative machine learning tasks based on their distributed computation and in-memory abstraction, but the performance of category validation actually degrades over Android kernel features in **task_struct**. In this paper, to thoroughly investigate Android kernel behaviors, we first present a kernel feature based framework, **CrowdNet**, for cloud computing platforms. CrowdNet includes an automatic data provider that collects footprints of kernel features and a parallel malware predictor that validates Android malicious behaviors. Then we calculate and select hidden centers by a heuristic approach for 12,750 Android applications to reduce the number of iterations and time complexity. Our experimental results show that CrowdNet protects large-scale data validation and speeds up the learning of kernel behaviors twofold. Further, identifying malicious attacks with CrowdNet improves the classification efficiency compared to traditional neural network and other machine learning techniques.

INDEX TERMS Android system, kernel feature, malware detection, machine learning, neural network, cloud computing.

I. INTRODUCTION

Due to the user-friendly and reliable operating system in Android devices, Android phones have become pervasive and ubiquitous in our daily life and 76.7% of them are captured by Android system [1]. The popularity of Android systems encourages developers and researchers to design and implement Android applications for satisfying different types of users. Unfortunately, the number of Android applications (*apps*) has a phenomenal growth accompanied by the rise of malicious attacks. According to a threat report from Kaspersky Lab in 2018 [2], there is a doubling of the number of attacks with malicious Android software: 116.5 million, against 66.4 million in 2017. These malicious attacks forcefully inflict different types of damages on normal Android devices, from the loss of important private user information to the disruption of system's performance. Consequently, a myriad of Android malware detection approaches [3]–[7] have been proposed to solve this issue and safeguard Android systems.

To comprehensively study the phenomenon of Android malicious attacks and properly detect the threats, kernel-based malware detection [8], [9] has been proposed and improved in 2011 and 2013 respectively. This technique audits all the applications of Android systems and obtains comprehensive log information from a Linux[1] kernel layer of Android systems. However, the practitioners in [9] only collected 32 kernel features from the Linux kernel of Android systems. In consideration of kernel-based malware detection, the number of kernel features is crucial to the correctness and scalability of Android malware detection [10]. Therefore, to leverage the kernel-based malware detection, extending multiple dimensions of all kernel features becomes indispensable and consequential to validate two categories of Android applications. In addition, due to continuous scanning for the entire kernel structures, the size of data collection increases exponentially, which typically leads to a poor classification performance and slow execution time. The massive data which cannot be processed by a single computer [11] elaborates useful information of Android malware detection. A cluster computing architecture with in-memory abstraction

The associate editor coordinating the review of this manuscript and approving it for publication was Junchi Yan.

[1]Linux 14.04.1-Ubuntu

such as Apache Spark [12] is likely to enable the time-efficient malware detection for massive Android kernel data analytics.

As the fundamental in-memory-oriented data structure, **R**esilient **D**istributed **D**ataset (**RDD**), Apache Spark caches massive data into shared memory and handle them efficiently. To eliminate the overhead of I/O communications, it iteratively exploits RDDs for parallel operations while dealing with large-scale data samples. But performing iterative scientific computation over Android kernel features incurs the decrements of classification performance in particular for neural network method. On the other hand, distributed iterative numerical calculation destroys the characteristics of malicious and benign behaviors that identify distinctive categories.

Detecting malicious threats with large-scale data samples in Android systems needs quantities of resources [13], e.g., a large memory, a high-speed CPU, because the scientific computation becomes more and more complicated along with the curse of dimensionality and data size. When dealing with the large-scale data on a general-purpose computer, there are a plethora of intermediate results saved in the memory or the disk, which introduces additional storage overhead for multi-dimensional data. As a fast analytics engine for massive data and machine learning, Spark is able to continuously cache data to shared memory and speed up large-scale data optimization. However, huge volumes of kernel feature datasets and a large number of machine learning iterations decrease classification performance of Android malware detection in Spark. To bridge the performance gap between massive data and machine learning, we propose a CrowdNet framework over kernel features and design several techniques in Spark.

The CrowdNet malware detection framework, consisting of two key components: an automatic kernel feature provider and a novel neural network-based predictor, is able to systematically analyze and evaluate kernel features and improves classification performance of Android malware prediction by dissecting different parts of kernel features. CrowdNet provides dedicated supports for gathering massive data and predicts malicious attacks based on their characteristics. Thereby it avoids unnecessary repetition of the analysis of malicious behaviors and makes up for the lost performance caused by machine learning iterations. Compared to the traditional neural network on parallel platform, CrowdNet calculates hidden centers by a heuristic approach in parallel and outperforms fine-grained operations. Furthermore, its properties of category identification best fit the demands of in-execution dynamic malware analysis and detection. In summary, our research makes the following contributions:

1) An automatic CrowdNet data provider is proposed to gather genetic footprints of Android kernel features in **task_struct**, which sequentially scans the installation files and automatically saves massive original data into the parallel database. Further, a large-scale dataset of Android kernel features, including above 191,250,000 data records, has been constructed for indicating a decent coverage of Android applications.
2) A novel CrowdNet predictor based on Android kernel feature datasets is designed and implemented to leverage the traditional neural network, which supports the efficient malware prediction and enhances the detection performance. Our optimized prediction technique as the best classifier between seven popular classifiers improves the performance by 12% compared to the traditional neural networks and only uses around 50% of original time to execute the prediction program.
3) We evaluate the performance of CrowdNet with a broad set of computing nodes and data volumes. Our results demonstrate that CrowdNet achieves the best performance than other techniques in terms of accuracy rates in parallel platforms. Moreover, CrowdNet reduces the time consumption caused by frequent iterations and I/O communications.
4) To the best of our knowledge, CrowdNet is the first system designed to automatically collect Android kernel features and concurrently predict Android malware on distributed platforms, which will serve as guidelines for researchers to explore the trends of Android kernel features and provide the support of validating Android malware detection models.

The rest of this paper is organized as follows. Section II introduces how to identify malicious attacks over kernel features in Android. Related work is shown in Section III. Sections IV and V present the design of CrowdNet data provider and CrowdNet predictor. Then, Section VI evaluates the benefits of the CrowdNet framework. Conclusion is discussed in Section VII.

II. IDENTIFY ANDROID MALWARES OVER KERNEL LAYER

This section introduces the background of Android malware detection, the challenges of Android malware detection and the CrowdNet model of how to identify Android malwares over kernel features.

A. ANDROID BASICS AND KERNEL LAYER

Figure 1 illustrates four main components of Android system [14], which is comprised of Applications, Application Framework, Libraries & Android Runtime, and Linux Kernel. The application layer is located on the top of Android system, with the responsibility for installation and operation of user software. The application framework contains high-level services in the form of java classes for communications between application layer and Android libraries. The Android libraries layer provides the resource access to system APIs from the second layer, in addition to those C/C++ based applications. The Android runtime encompasses two important components, core libraries for the standard java language and light-weight Android virtual machine. The bottom layer, Linux kernel, is the core of Android architecture, which handles the process scheduling, memory management, power management, communication between hardware and

TABLE 1. Feature categories and number signs (from number 3 to number 114, 112 features in total) in data structure task_struct: task_struct structure contains 6 task_state features, 48 mem_info features, 15 sche_info features, 30 signal_info features, and 13 others features.

Categories (#.)	112 Android Kernel Feature Names & Their Number Signs
task_state (6)	[3]exit_state, [4]exit_code, [5]exit_signal, [6]pdeath_signal, [7]jobctl, [8]personality
mem_info (48)	[9]maj_flt, [10]min_flt, [11]arg_end, [12]arg_start, [13]end_brk, [14]start_brk, [15]cache_hole_size, [16]def_flags, [17]start_code, [18]end_code, [19]start_data, [20]end_data, [21]env_start, [22]env_end, [23]exec_vm, [24]faultstamp, [25]mm_flags, [26]free_area_cache, [27]hiwater_rss, [28]hiwater_vm, [29]last_interval, [30]locked_vm, [31]map_count, [32]mm_count, [33]mm_users, [34]mmap_vmoff, [35]mmap_base, [36]nr_ptes, [37]pinned_vm, [38]reserved_vm, [39]shared_vm, [40]stack_vm, [41]total_vm, [42]task_size, [43]token_priority, [44]nivcsw, [45]nvcsw, [46]start_stack, [47]rss_stat_events, [48]usage_counter, [49]nr_dirtied, [50]nr_dirtied_pause, [51]dirty_paused_when, [52]normal_prio, [53]utime, [54]stime, [55]utimescaled, [56]stimescaled
sche_info (15)	[57]last_queue, [58]pcount, [59]run_delay, [60]state, [61]on_cpu, [62]on_rq, [63]prio, [64]static_prio, [65]rt_priority, [66]policy, [67]rcu_read_lock_nesting, [68]stack_canary, [69]last_arrival, [70]flags, [71]ptrace
signal_info (30)	[72]group_exit, [73]signal_nr_threads, [74]signal_notify_count, [75]signal_flags, [76]signal_leader, [77]signal_utime, [78]signal_cutime, [79]signal_stime, [80]signal_cstime, [81]signal_gtime, [82]signal_cgtime, [83]signal_nvcsw, [84]signal_nivcsw, [85]signal_cnvcsw, [86]signal_cnivcsw, [87]signal_maj_flt, [88]signal_cmaj_flt, [89]signal_cmin_flt, [90]signal_inblock, [91]signal_oublock, [92]signal_cinblock, [93]signal_coublock, [94]signal_maxrss, [95]signal_cmaxrss, [96]signal_sum_sched_runtime, [97]signal_audit_tty, [98]signal_oom_score_adj, [99]signal_oom_score _adj_min, [100]sas_ss_sp, [101]sas_ss_size
others (13)	[102]gtime, [103]link_count, [104]total_link_count, [105]sessionid, [106]parent_exec_id, [107]self_exec_id, [108]ptrace_message, [109]timer_slack_ns, [110]default_timer_slack_ns, [111]curr_ret_stack, [112]trace, [113]trace_recursion, [114]plist_node_prio

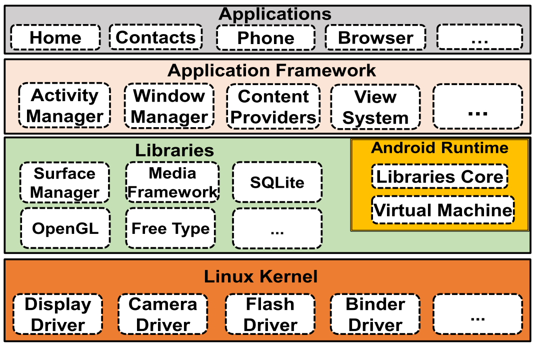


FIGURE 1. **Android system architecture.**

software, etc. In this open-source software platform, our CrowdNet mainly focuses on the Linux kernel layer.

B. ANDROID KERNEL STRUCTURES IN PCB

The data structure, **task_struct** [15] in Process Control Blocks (PCB), as a descriptor of process interaction, has 112 features to store the information of executing programs. It gives us an elaborate description of a running process, e.g., process state, process priority, scheduling policy, etc., after being allocated by the slab allocator. While measuring the variables constantly invoked by a malware process, kernel features can be used for delimiting the malicious behaviors. PCB can dynamically update and maintain the process identification data, process current state and process control information, so a method of mining PCB in the Linux operating system is proposed to predict malware applications in [9], [10], [16]–[20].

The 112 kernel features of Android **task_struct** [8] in Process Control Blocks (PCB) and their categories are shown in Table 1. **hash_key** & **classifier** mean unique nonmalware or malware software applications applied in smartphones. 6 **task_state** variables are defined to describe the exiting case of task execution. The traces of 48 **mem_info** memory usage features indicate resource demand and process interaction. When the ability of computation of an OS exceeds its threshold, a reasonable scheduling strategy in 15 **sche_info** features is introduced to increase the system's tolerance. The task structures containing 30 **signal_info** features reserve space for handling received signals for each process which applies or utilizes the limited resources to restrict or make excessive use of CPU, memory, or disk. The remaining 13 **others** kernel features are used to describe guest time, link number, session ID, etc. Note that the number signs of **hash_key** and **classifier**, Number 1 and Number 2, are not shown in Table 1.

C. APACHE SPARK FOR ITERATIVE MACHINE LEARNING

Figure 2 shows the brief framework of Spark on Mesos [21] and resource allocation example. Mesos Cluster communicates the Spark Scheduler and Data Nodes with Mesos Master and Mesos Slaves. Mesos Master, located between Spark Master and Data Nodes, is responsible for retrieving the usage information of resources from Mesos Slaves on Data Nodes and informing Spark Scheduler. When receiving the details from Mesos Master, Spark Scheduler decides the job with a higher priority should be launched at first and then sends the feedback signal to Mesos Cluster. Meanwhile, Spark loads the data samples into shared memory constituted by RDD from Hadoop Filesystem (HDFS), which translates the disk-based data to in-memory data. For example, **Node 1** (N1)

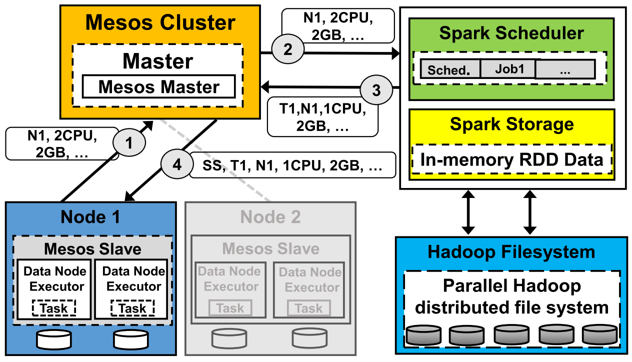


FIGURE 2. **Framework of spark on Mesos and example of resource allocation.**

in Figure 2 sends the information of free resources (2CPUs, 2GB memory) in ① to Mesos Cluster. Then Mesos Cluster sends the details in ② to Spark Scheduler and Task 1 (T1) is designated to utilize the resources (1CPU, 2GB memory) of Node 1 in ③ by Spark Scheduler. Finally, Mesos Cluster assigns T1 to N1 with 1CPU and 2GB memory via ④. Note that **N1**, **T1** and **SS** in ① ② ③ ④ represent Node 1, Task 1 and Spark Scheduler, respectively.

D. CHALLENGES OF IDENTIFYING MALICIOUS ATTACKS AT SCALE

Collecting large datasets is indispensable for training accurate malware detection models [22]. The difficulty of collecting large-scale datasets lies with Android platforms, in which there is no existing data collection and storage tools.

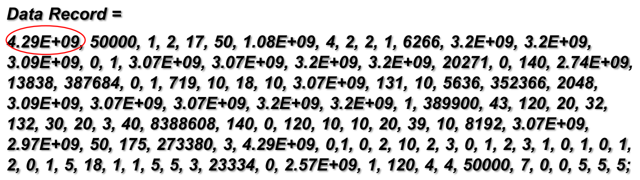
1) CHALLENGE 1: LARGE-SCALE DATA COLLECTION IN ANDROID PHONES

There are many approaches to gathering Android malware datasets, for instance, collecting crowdsourcing datasets, emulating user interactions and investigating user logs [23]. Due to the popularity of crowdsourcing data with its convenience and inexpensiveness, the datasets with kernel features have been used to detect Android malware. But the datasets including 112 kernel features from Android kernel layers have not been collected by software practitioners. To investigate Android kernel features, an automatic data provider must support massive data collection for specific information in Android kernel layers.

In order to profile malicious apps from Android devices, massive data storage has become a critical challenge on distributed systems. The strategy of large-scale data storage must be provided on parallel platforms because the storage space on Android phones is limited. However, in this work, our framework scans 750 records every second and completes data collection in 20 seconds. Finally 15,000 original data records can be collected for each app. But these massive data records cannot be saved into a small memory card in Android. So we design a data processing module in Section IV-D to transfer data, compress data and store data.

2) CHALLENGE 2: ACCURATELY NORMALIZING DATA RECORDS

The dimensionality of data records affects the efficiency of massive data analytics for Android apps [24]. To reflect the effect of massive data from Android devices, the elaborative analysis of a data record of Android app is a must. Figure 3 demonstrates a data record of an Android app containing 112 kernel features described in Table 1. The letter E in Figure 3 represents the exponent of ten. For instance, $4.29E+9$ marked by a red oval in Figure 3 stands for 4.29×10^9. It is observed that the value of each kernel feature is either too small (less than 10) or too large (equal to 4.29×10^9), which incurs underfitting of training the large-scale data [25]. To preserve high efficiency of training a best-fit model, normalizing data is required to reduce the time overhead of malware analysis. Hence how to accurately normalize massive data becomes more compelling in a single computer. It is obvious that the individual computer does not support the efficient operations at scale. Apache Spark, as a fast analytics engine, can speed up data normalization and learning rate in parallel. Another distributed framework, Message Passing Interface (MPI), provides standards and libraries for distributed computing systems [26], but compared to Spark it has a lower performance while tackling iterative machine learning algorithms. Thereby the distributed machine learning based framework, CrowdNet, is proposed to secure the accuracy rate of original large-scale data in Section V.



Data Record =

4.29E+09, 50000, 1, 2, 17, 50, 1.08E+09, 4, 2, 2, 1, 6266, 3.2E+09, 3.2E+09, 3.09E+09, 0, 1, 3.07E+09, 3.07E+09, 3.2E+09, 3.2E+09, 20271, 0, 140, 2.74E+09, 13838, 387684, 0, 1, 719, 10, 18, 10, 3.07E+09, 131, 10, 5636, 352366, 2048, 3.09E+09, 3.07E+09, 3.07E+09, 3.2E+09, 3.2E+09, 1, 389900, 43, 120, 20, 32, 132, 30, 20, 3, 40, 8388608, 140, 0, 120, 10, 10, 20, 39, 10, 8192, 3.07E+09, 2.97E+09, 50, 175, 273380, 3, 4.29E+09, 0,1, 0, 2, 10, 2, 3, 0, 1, 2, 3, 1, 0, 1, 0, 1, 2, 0, 1, 5, 18, 1, 1, 5, 5, 3, 23334, 0, 2.57E+09, 1, 120, 4, 4, 50000, 7, 0, 0, 5, 5, 5;

FIGURE 3. A data record of 112 kernel features.

3) CHALLENGE 3: PERFORMANCE LOSS AND DEGRADATION FOR MASSIVE DATA

Area Under an ROC Curve [27] intuitively tells how much prediction models are capable for validating categories. To help investigators to analyze Android kernel features based on machine learning, a robust analysis of how machine learning techniques affect malware prediction is allowed and shown in Figure 4. It illustrates the ROC space of the six methods, **N**aive **B**ayes (**NB**), **N**eural **N**etwork (**NN**), **C**onvolutional **N**eural **N**etwork (**CNN**), **D**ecision **T**ree (**DT**), **S**upport **V**ector **M**achine (**SVM**) and **L**ogistic **R**egression (**LR**). When applying these methods to our massive dataset, they generate four separate confusion matrices that in turn correspond to ROC points [28]. The X-axis denotes the false positive rate which equals to the number of negatives incorrectly classified divides by the number of total negatives.

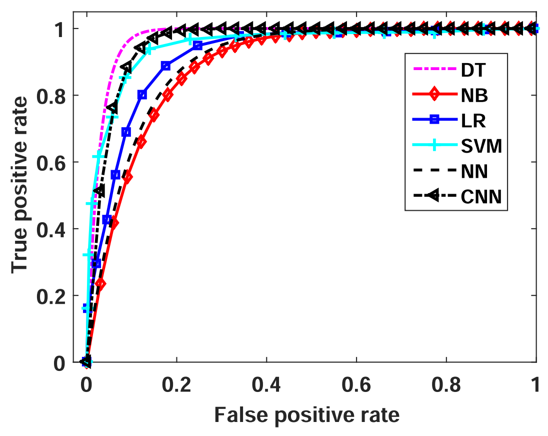


FIGURE 4. ROC curves of six machine learning methods in massive data.

The Y-axis denotes the true positive rate which equals to the number of positives correctly classified divides by the number of total positives.

As we can see in Figure 4, in contrast with DT, CNN, SVM, and LR, NN achieves a lower performance to offer low availability and security for massive kernel data. Furthermore, the ROC curve of NN increases slowly against with false positive rate with a smaller value of AUC. NN, as an efficient computational model, should have provided the best solutions to classification and regression according to characteristics and qualities of data. But the large amounts of Android data samples and the characteristics of massive data incur the lower performance specifically for NN method. Actually the number of Android samples greatly exceeds the dimension of Android kernel features, which is the main reason of performance loss and degradation for NN. Processing the large volumes of data cannot only depend on the original implementation of NN and has to devolve on a novel integrated neural network for performance improvement. Furthermore, while the size of Android dataset is growing exponentially, massive numerical computation of iterative machine learning also leads to extra memory and CPU overhead for NN. Hence we optimize the massive dataset and improve the classification performance by our new CrowdNet framework in the following sections.

E. ATTACK MODEL

In this section we outline the attack model that CrowdNet assumes and protects against. Our work mainly focuses on providing a data-efficient framework to analyze in-execution kernel features in Android, so we assume the client device is compromised by attackers that get access to the kernel data in Android kernel layer of Figure 1. On the other hand, we assume the adversaries control the entire kernel layer, which enables them to modify any kernel feature while processes are performing. Hence the dynamic attacks leave enough evidences in proc file which can be traced and predicted by our CrowdNet provider. In line with previous studies, CrowdNet can leverage well-known techniques for tracing footprints of in-execution Android kernel features such as kernel based behavior analysis [8], behavioral malware detection framework [29], [30], and multi-level anomaly detector [18]. We assume the customers are perceived to install an unknown Android application which attempts to be granted permission of the important information. Moreover, we assume the untrustworthy application such as Trojan horse masquerading as the legitimate controls the fine-grained dynamic data access and operation in proc system file that acts as a dynamic interface to map kernel information. The virtual file system, proc file, is generated when system boots and is dissolved at time of system shutdown.

We assume the Android permission system does not deny access to user sensitive data, including SMS, business (trade secrets, contracts, or call information), etc. In relation to the running processes in Android, we assume adversaries do not incur severe damage in Android system and they only steal the user private information, obtain the administrator privilege, or misuse resources. We also make the standard assumptions about Internet connection, e.g., that users can use Wi-Fi to connect to Internet, that users can use unlimited cellular data of 4G LTE (Long-Term Evolution), so Android malicious applications can be downloaded and installed from the online market and then massive data of malicious attacks can be transferred and stored to distributed platforms. Finally, we assume malicious attacks leave evidences in proc file between the scan interval and the evidence in Android kernel layer is not erased before uninstalling the application. To achieve the correct footprints, we also assume the attacks are computationally bound and therefore cannot easily run the brute force attacks.

F. SYSTEM ARCHITECTURE

Our CrowdNet Architecture is shown in Figure 5, including two main modules: CrowdNet Data Provider and CrowdNet Predictor. On the top of Figure 5 is the CrowdNet data provider which is in charge of collecting massive kernel data from Android devices. On the bottom of Figure 5 is the CrowdNet predictor, in which massive data is saved into distributed database and analyzed to detect Android malware. To enhance the performance and availability, CrowdNet also provides the support of malware detection for massive data. CrowdNet is deployed on Apache Spark [31] with Apache Cassandra [32] and Mesos [21]. The CrowdNet data provider can dynamically trace the footprints of kernel features while the process is running and then transfers data to parallel storage system on computer cluster. Overall, it constantly works on client side and transparently communicates with administrators on parallel platforms. The CrowdNet predictor stores massive data from CrowdNet data provider and reduces the size of the large-scale data with standardized function.

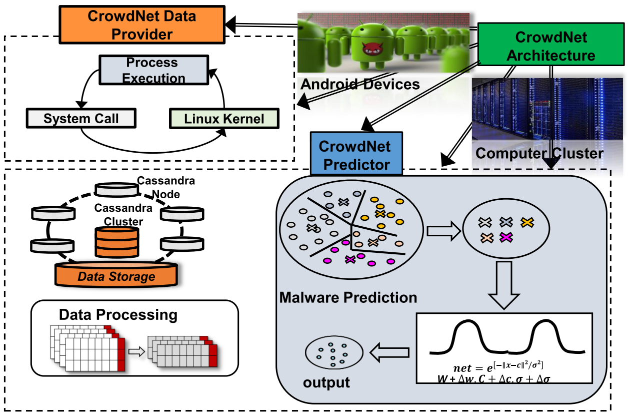


FIGURE 5. CrowdNet architecture with data provider and predictor.

From the above discussion, the communication between the two major components is orchestrated by UDP services and HTTP services with several important programming tools. The CrowdNet provider is responsible for data transferring, data compressing and message communication. When it receives data instances and finishes data compressing, UDP and HTTP services will send them to computer side with a USB cable or over a Wi-Fi connection for distributed storage. Additionally, CrowdNet predicts Android malicious attacks associated with provider and predictor on distributed platforms.

III. RELATED WORK

A. ANDROID MALWARE DETECTION OVER KERNEL FEATURES

Kernel-based malware detection has been proposed to dynamically analyze the behaviors of Android malicious apps. For instance, H. Alptekin et al. theoretically propose a malware detection method, named TRAPDROID, from Linux kernel perspective in [20], where they dynamically capture unified behavior profiles, such as maj_flt, min_flt, stime, utime, etc., to demonstrate the importance of kernel features for Android malware detection. In [10], the "Andromaly" framework is also proposed to collect Android kernel features. The researchers utilize machine learning algorithms to find the best combination for their own datasets. Ham et al. [33] have collected 32 resource features of network, SMS, CPU, power, memory, virtual memory and process. They also use different machine learning algorithms to train suitable models for Android systems. In [9], F. Shahzad et al. provide a TstructDroid framework to discriminate Android benign apps and Android malicious apps. They gather a small dataset consisting of 110 malicious apps and 110 benign apps with 32 Android kernel features. Isohara et al. [8] design an audit application called logcat on virtual machine to monitor the application behaviors and propose a kernel-based behavior analysis method.

B. LARGE-SCALE STUDY ON ANDROID MALWARE DETECTION

Due to the growing popularity of high performance computing, many researchers provide parallel frameworks to analyze their large-scale datasets. Yuan et al. [34] propose a new framework, named Lshand, to discover unknown Android malware and perform further analysis on Android apps. Y. Zhang et al. design a novel ANDroid Hybrid REpresentation Learning (ANDRE) method to cluster weakly-labeled Android malware [35] and a machine learning based malware detection system [36] is proposed to impove the detecting accuracy. V. Afonso et al. use a large-scale analysis for Android apps to create native code policy in [37]. Mojica et al. [38] analyze the software reuse on hundreds of thousands of Android apps across 30 categories. SMV-HUNTER [39] combines static and dynamic analysis for large-scale identification of vulnerabilities in 23,418 apps. An offloading algorithm [40] based on Q-learning is proposed for smartphones to accurately detect Android malware with unknown features. Huang et al. [41] provide an insight on Android malware development by a system called AMDHunter for revealing new Android threats. Paranthaman et al. [42] also utilize Apache Spark to detect Android malware in 2169 software samples. To precisely detect Android malware, J. DeLoach et al. leverage a modified Logistic Regression classifier in [43]. DroidRA [44] is used to extract the target object values of reflective methods. R. Goyal et al. present a distributed service called SafeDroid in [45] to detect malicious apps on Android platforms.

IV. CROWDNET DATA PROVIDER

From the above disscusion, we identified the major challenges of Android malware detection. Therefore, in this section, we systematically interpret how CrowdNet data provider gathers the 112 fields of Android task structure and analyzes Android apps installed into a real Android device.

A. MALWARE & BENIGN DATASETS

As shown in Tables 2 and 3, we gather 6375 representative Android malicious apps by VirusTotal [46] from 38 Android malware families and 6375 Android benign apps in 24 popular benign categories from Google Play Store [47] where there is an APK downloading mirror [48]. Our data collection is described in the following sections. To trace footprints of 112 kernel features in **task_struct** our CrowdNet provider monitors and retrieves 750 records per second while Android programs in Tables 2, 3 are executing. For each Android app, we finally obtain 15,000 data records in 20 seconds. After collecting data records of 12,750 Android apps, we construct

TABLE 2. Android malware families.

Malware	Samples	Malware	Samples
ADRD	261	IconoSys	246
Adwo	140	jSMSHider	161
Agent	120	KMin	124
AnserverBot	120	LoveTrap	112
Asroot	110	NickySpy	119
BaseBridge	140	Plankton	136
BigServ	109	Pjapps	110
Boxer	110	SMSKey	107
Dowgin	200	SMSreg	131
DroidDreamLight	256	SMSReplicator	156
Droidkungfu 1-4	688	SndApps	134
FakeDoc	111	OpFake	130
FakeInst	132	TapSnake	126
FakePlayer	210	Waspx	114
Geinimi	281	YZHC	113
GingerMaster	210	Youmi	116
GoldDream	207	Zhash	111
Gone60	206	ZitMo	107
HippoSMS	204	Zsone	207

TABLE 3. Android benign categories.

Benign	Samples	Benign	Samples
AR Apps	260	Music	260
Books	260	Navigation	260
Business	260	News	260
Education	260	Photo & Video	260
Entertainment	260	Productivity	260
Finance	260	Reference	260
Food & Drink	260	Shopping	260
Health & Fitness	260	Social Networking	260
Kids	260	Sports	300
Lifestyle	260	Travel	300
Magazines & Newspapers	260	Utilities	300
Medical	260	Weather	275

two massive datasets. Our datasets contains 191,250,000 data records in total which indicates a decent coverage of Android apps.

B. READ AND WRITE OF KERNEL MODULE IN ANDROID

To efficiently achieve the massive data from Android phones, a new module with read and write operations must be constructed to record the changes of system resources. Thereby we directly build a new component for virtual file management to monitor the changes of kernel features. If this module detects the variation of system resources, a new variable for the current task with a unique process ID is generated and sent to the hard disk via socket connection. Here, the new variable only needs 4 bytes to store in Android, which does not cause extra storage and schedule overhead. Meanwhile, we can achieve the changed variables of kernel features and transfer them to other storage devices.

C. ANALYSIS SEQUENCE OF APK FILES

To accurately analyze the sequence of APK files, we design a simple state machine in Android. Initially, we install APK files into a physical device with a Node.js program when the Android system is available. But if the device is busy with waiting or recharging, we do not launch the analysis program because the entire system is not stable and safe to run additional programs. Until the device becomes ready to analyze APK files, our analysis program automatically installs APK files to Android device.

We define three states for the analysis sequence of APK files:

1) Ready: this state shows the next APK is ready to install to the device or analyze via **/proc** file.
2) Waiting: this state shows multiple APKs are waiting for installing or analyzing when the system is busy with analyzing other APKs.
3) Recharging: this state shows the device is busy with charging without installing or analyzing any APKs.

D. ARCHITECTURE OF CROWDNET DATA PROVIDER

The multiple dimensional kernel feature's provider shown in Figure 6, working both on Android devices and storage servers, is mainly composed of three components: (1) The scheduling mechanism of Android APK repository, (2)&(3) message (package) communication in local computer, and (4)&(5) data processing of compression, transformation and storage via several User Datagram Protocol (UDP) services of lightweight data package transmission and Hypertext Transfer Protocol (HTTP) of the request-response module. In order to efficiently scan the detailed information of kernel features, the scheduling mechanism is implemented to dispatch the setup of Android apps concurrently. Therefore, we design a lightweight scheduler for the scheduling component (1) to reduce the storage overhead and assign scanning tasks.

As shown in Figure 6 (2)&(3), the message communication component on computer side with /proc file generates intermediate results to read the information of all kernel features (**task_struct**). Components (2) and (3) in Figure 6 are the same module to illustrate the implementation of message communication. After loading this module into physical devices, multiple communication assignments, such as memory allocation, read/write operations, scheduling strategies, etc., can be executed coordinating with the scheduling part. Furthermore, how to monitor the utilization efficiency is indispensable for system's maintenance and succinctness.

In Figure 6 (4)&(5), we introduce scalable data processing component which converts data formats, compresses data

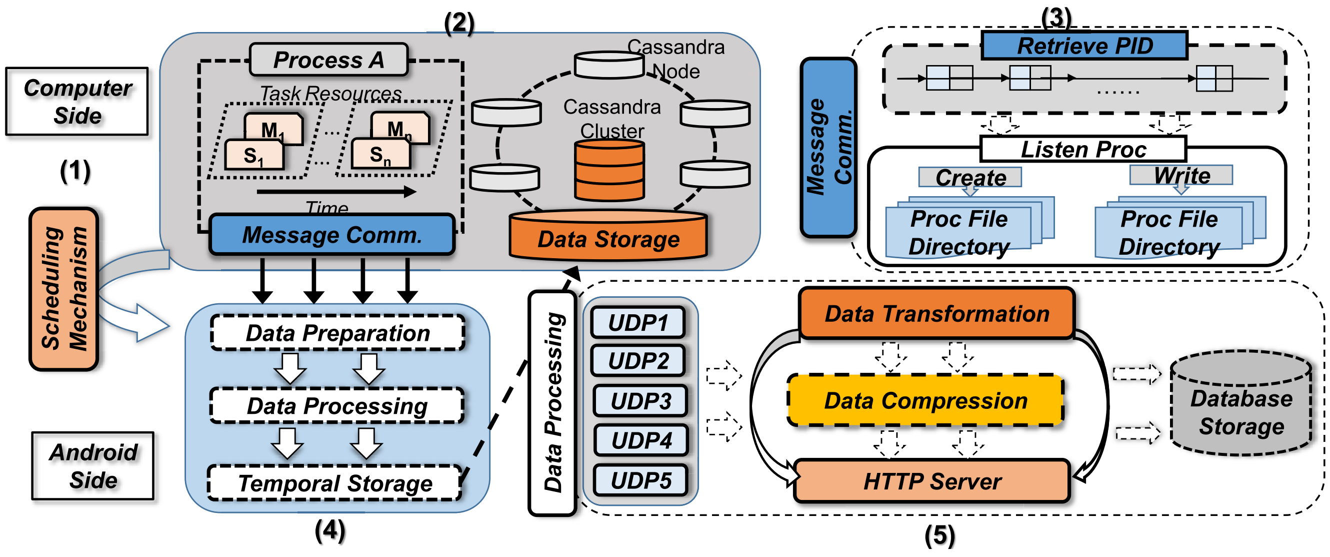


FIGURE 6. Overview of multiple dimensional kernel feature's (Raw Data) provider. In (2)&(3), message communication module in local computer, where (2) and (3) are the same component. In (4)&(5), data processing module in android kernel, similarly, where (4) and (5) are the same component.

volumes and transfers data. Components (4) and (5) working on Android side are also the same module divided into two parts: UDP and HTTP. To convert binary data to string data, UDP services need support numerical calculation and transformation. Moreover, the data must be compressed to the format with less bytes to reduce the storage overhead in Android. But when data conversion and data compression are finished, the new data will be sent to a local database from this temporal data pool.

E. IMPLEMENTATION OF CROWDNET DATA PROVIDER

Algorithm 1 illustrates the implementation of our automatic CrowdNet data provider in section II-F. We first manage APK files on computer side in Lines 1-2. Line 1 shows the downloaded APK files are copied to the specified folder and then they are sorted by Line 2 according to their last modified date. The scheduling mechanism in Figure 6 is simply implemented in Lines 1-2. As shown from Line 3 to Line 8 in Algorithm 1, we build the message communication component of Figure 6. After orchestrating Android apps on computer side, HTTP server is started for listening to /proc file and relaying Android data records in Line 3. Line 4 illustrates the process of collecting data from Android is monitored by port 8080. Lines 5-6 show a tool called Android Debug Bridge (ADB) inserts a new kernel module into Android device. And then ADB triggers installing APKs by Line 7 and achieves the current process ID to verify different APKs by Line 8. The data processing component in Figure 6 is implemented from Line 9 to Line 18 in Algorithm 1. Line 9 launches UDP service to send Android data records from Android side to computer side and Lines 10-11 trigger data collection on Android side. Lines 12-15 show the iterative data collection and Line 16 is the data compression. The data is sent to computer side via HTTP server in Line 17 and stored into the parallel database in Line 18.

Algorithm 1 CrowdNet Data Provider

```
1: cp − r /home/Android_apps/ ∗ /home/Android_installed
2: sort_apps(path)
3: httpServer(relay_data)
4: listenPort(8080)
5: adb push tstruct_mod.ko /sdcard/
6: adb shell insmod /sdcard/tstruct_mod.ko
7: adb shell am start activity_name
8: PID ← ps activity_name
9: udpServer(PID)
10: adb shell echo PID → /proc/getMalwareData
11: adb shell sh start_collect(PID)
12: while (i ≠ 15000) do
13:     sample ← cat /proc/getDataInstance
14:     i ← i + 1
15: end while
16: compressdata(sample)
17: switch2httpServer(PID)
18: savedata(sample)
```

V. CROWDNET PREDICTOR

To mitigate the gap between massive data and I/O communication, we also design a novel CrowdNet predictor for large-scale datasets. Hence, we introduce the implementation of CrowdNet predictor and the strategies of how CrowdNet predictor to deal with massive data and improve the performance in parallel.

A. DATA PROCESSING

To maintain the massive data integrity, all kernel features within the parallel database are preserved over continuous time. Initially, we have to look up the metadata in distributed system to find out the data instances with inaccurate values. But if some missed or corrupted data instances are observed, these unexpected fields must invoke one or more data management operations for data consistence. In this work, we remove the inaccurate fields, replace the missed or corrupted fields with their expected values. With the frequent analysis of CrowdNet provider, CrowdNet predictor needs to cope with redundant data instances that affect the classification accuracy and the execution time. Therefore, to retrieve a consistent effect, CrowdNet predictor cleans up the original data over different rows of datasets for the high-quality data management.

In CrowdNet predictor, we also utilize standard procedures to improve the quality of large-scale data. Each classification model needs different types of data instances for achieving a better classification performance. In order to generally satisfy the requirements of most classification techniques, all data instances are converted into numerical data format with a [0,1] or [−1,1] range by normalized functions, improving the efficiency of complicated model training. There are two popular methods to standardize numerical data instances [49] as shown as below, 0-1 scaling of Equation (1) and Z-score scaling of Equation (2),

$$X_i' = (X_i - X_{min})/(X_{max} - X_{min}), \quad X' \in [0, 1] \tag{1}$$
$$X_i'' = (X_i - \mu_s)/\sigma_s \tag{2}$$

In Equation (1), X_i', X_i, X_{min}, X_{max} represent each new value after data transformation, each original value of a column, the minimal value between all the data instances in the same column with each original value, the maximal value between all the data instances in the same column with each original value, respectively. X_i'', X_i, μ_s, σ_s of Equation (2) stand for each new value, each original value of a column, the expected value in the same column with each original value, and the standard deviation value in the same column with each original value.

Since this method in Equation (1) guarantees that all the new features will be not more than one and not less than zero, it is applied to standardize our streaming data instances from Android phones. And A. Kusiak has proved that the classification accuracy of massive data can be improved with the specific feature bundles [50].

B. BASIC OVERVIEW OF CROWDNET METHODOLOGY

Radial basis function networks [51] are proposed to solve the issues of nonlinear classifications or nonlinear approximation. In terms of the traditional NN algorithm that attempts to reduce the global error rate through less iterative calculations, it does not fit the exascale computation of millions of data instances in Section II-D3. Thereby CrowdNet replaces the kernel of NN method with the radial basis function to consolidate multiple techniques to a combined parallel method. CrowdNet also utilizes a Gaussian kernel to finish the nonlinear transformation of massive data instances, which improves the training performance at scale.

In addition, CrowdNet is composed of three similar layers compared to NN, input layer, hidden layer and output layer. Like the traditional NN method, its input layers deal with the original or reduced data for next layers, but its hidden layers perform nonlinear transformation and data mapping in a new space with the following Equation (3):

$$net_j = exp(-\left\|X - C_j\right\|^2 / \sigma_j^2) \tag{3}$$

where net_j, X, C_j, σ represent the j-th neuron's net value, the input vector, the j-th neuron's center position, and the j-th neuron's standard deviation, respectively. $\|.\|$ denotes the Euclidean norm and $\left\|X - C_j\right\|$ stands for Euclidean distance between the pattern and the center. CrowdNet utilizes center's values C and its standard deviation between input layers and hidden layers. However, NN attempts to train the weights W_1.

The output layers of CrowdNet need to combine each output values from hidden layers according to Equation (4):

$$o = \sum_{j=1}^{n} W \times net_j \tag{4}$$

where o, j, W, net_j represent the output value of output layers, the number of neurons from 1 to n, the vector of the weights and the j-th neuron's net value, respectively. CrowdNet includes three steps: clustering centers, calculating net values and standardizing outputs. Actually, the first step demonstrates the method of how to cluster training centers and select the closest center (normalizing the output) for an unknown value. This step need to be finished by the clustering algorithms before deploying CrowdNet transformation. Then, the second step given by Equation (3) is able to calculate net values and according to the output function (4) the third step with standardizing outputs also can be accomplished.

C. CALCULATE CROWDNET CENTERS

As discussed in Section V-B, a clustering algorithm is required to divide massive data into several regions. For brevity, K-means algorithm is used to separate a clustering of data instances into K regions in Algorithm 2.

Algorithm 2 K-means Clustering for CrowdNet Centers

```
1: Input: Training dataset $D$, number of clusters $k$
2: Initialize $k$ clusters randomly or Read $k$ clusters $c_j$
3: Set $sum_j = 0$ and $n_j = 0$ for $j = \{1, \ldots, k\}$
4: while TRUE do
5:     for $x_i \in D$ do
6:         for $j \in \{1, \ldots, k\}$ do
7:             $j_{min} = \arg\min \left\|x_i - c_j\right\|$
8:             $sum_{j_{min}} = sum_{j_{min}} + x_i$
9:             $n_{j_{min}} = n_{j_{min}} + 1$
10:            $D_j \leftarrow x_i$
11:        end for
12:    end for
13:    for $j \in \{1, \ldots, k\}$ do
14:        $c_j = sum_j / n_j$
15:    end for
16: end while
```

Algorithm 2 explains how K-means finds a suboptimal partition for unknown data instances. In Algorithm 2, Line 2 randomly chooses the data instances from large-scale datasets. In line 3, sum_j is the summation of all data points belonging to the j-th center, n_j denotes the total number of all data points belonging to the j-th center. During iterative computation of k clustering regions, additional variables, $sum_{j_{min}}$ and n_j, are used to temporally store intermediate results in Line 8. Here, $sum_{j_{min}}$ denotes the minimum summation for x_i. And then K-means assigns the present data point x_i to the region D_j which is the closest centroid to x_i. Meanwhile, it calculates the relevant cluster statistics in Lines 5-12. Lines 13-15 update centroids of the existing k clusters with the mean of current dataset. Until all the centroids of k clusters rarely changes, the program will be terminated normally.

D. SELECT THE KERNEL WIDTHS (σ)

To optimize the activation function of CrowdNet, we need the centroid c_j and the standard deviation σ_j to decide the curve of the Gaussian Function. As discussed in Section V-C, we have introduced how to calculate the centroid c_j and will explain how to effectively select the kernel width σ_j. Since a very large or small σ_j, the kernel width [52], can lead to numerical issues with gradient descent algorithms, we adjust the kernel width dynamically based on different results of Gaussian basis function.

We can achieve the kernel width by different setting schemes [53]. In this study, to investigate the benefit from traditional techniques, we utilize K-means method to calculate the centroid c_j. Hence, the kernel width σ_j can be set to the mean of Euclidean distances between data points and their cluster centroids according to Equation (5):

$$\sigma_j = \frac{1}{n_j} \sum_{x \in D_j} \left\|x - c_j\right\| = \frac{1}{n_j} \sum_{x \in D_j} (x_i - c_{ij})^2. \tag{5}$$

In Equation (5), the values of the parameters n_j, D_j, c_j represent the number of data points belonging to the j-th cluster, the data collection of the j-th cluster, and the j-th clustering center, respectively, which are retrieved from the Algorithm 2.

E. GRADIENT DESCENT TO REDUCE ERROR RATE

CrowdNet iteratively reduces the error rate by gradient descent [54] to obtain the minimal error in Equation (6)

$$TE = \sum_{i=1}^{n}\sum_{j=1}^{k}\left(t_{i,j} - o_{i,j}\right)^2 \tag{6}$$

where $t_{i,j}$ is the target response of the i-th output from the j-th neurons and $o_{i,j}$ is the actual response of the i-th output from the j-th neuron. Actually, the value of $t_{i,j}$ is known and the value of $o_{i,j}$ is achieved by Equation 4. The minimal error is that the derivatives of clustering center c_j, kernel width σ_j and the output weight w_j vanish. Therefore, an iterative computation of gradient descent with the direction of the negative gradient $-\frac{\partial TE}{\partial w}$, $-\frac{\partial TE}{\partial c}$, $-\frac{\partial TE}{\partial \sigma}$ can solve this issue.

To further eliminate the errors with Gaussian basis function, the following updating rules are designed and implemented in CrowdNet. The details of updating rules are shown in Equations (7), (8), and (9):

$$\Delta w_j = -\alpha \sum_{i=1}^{n} net_j(x_i)(t_{i,j} - o_{i,j}) \tag{7}$$

$$\Delta c_j = -\alpha \sum_{i=1}^{n} net_j(x_i)\frac{x_i - c_j}{\sigma_j^2}\sum_{j=1}^{k} w_j(t_{i,j} - o_{i,j}) \tag{8}$$

$$\Delta c_j = -\alpha \sum_{i=1}^{n} net_j(x_i)\frac{(x_i - c_j)^2}{\sigma_j^3}\sum_{j=1}^{k} w_j(t_{i,j} - o_{i,j}) \tag{9}$$

where α is the learning rate constant which is significant to reach convergence [55]. Here we set the learning rate to a small constant value to simplify the training program and avoid overshooting the minimal errors. Algorithm 3 shows the implementation of gradient descent with the constant learning rate. The input values in Line 1 are achieved from Algorithm 2. With the iterative computation, the three values, Δw, Δc, $\Delta \sigma$, are utilized to update the previous values of W, C, σ in Lines 6-9. And then the new errors will be obtained from Lines 10-13.

VI. PERFORMANCE EVALUATION

In this section, we evaluate the accuracy of CrowdNet framework on the Spark system and the performance of classifications with multiple computing nodes.

A. EXPERIMENT CONFIGURATION

Our experiments are executed on Apache Hadoop v2.6.0 and Apache Spark v1.6.0. The configurations of Spark are shown in Table 4. In addition, the Apache Mesos v0.27.1 [56] is used for managing Spark running and dispatching resources.

Algorithm 3 Gradient Descent with Constant Learning Rate

```
1:  Input: Training dataset $D$, $\alpha$, $TE_{min}$, clustering centers
    set $C$, kernel width set $\sigma$
2:  Randomly choose the weight vector $W$, initialize the target output vector $TP$ and the input vector $X$ with dataset $D$
3:  while $TE > TE_{min}$ do
4:      $NET = EXP(-||X - C||^2/\sigma^2)$
5:      $OP = W * NET$
6:      $\Delta w = -\alpha * NET * (TP - OP)$
7:      $\Delta c = -\alpha * NET * (X - C)/\sigma^2 * W * (TP - OP)$
8:      $\Delta \sigma = -\alpha * NET * (X - C)/\sigma^3 * W * (TP - OP)$
9:      $W = W + \Delta w, C = C + \Delta c, \sigma = \sigma + \Delta\sigma$
        Compute the new total errors $TE$
10:     $OP2 = W * NET$
11:     $ERR = TP - OP2$
12:     $TE = sum(sum(ERR \cdot ERR))$
13: end while
```

TABLE 4. Apache spark configurations.

Parameter Name	Value
spark.master	spark://gpu-0-1:7077
spark.eventLog.enabled	true
spark.driver.memory	20 GB
spark.executor.uri	hdfs:/spark-1.6.0-bin-hadoop2.6.tgz
MESOS_NATIVE_LIBRARY	/local//lib/libmesos.so

B. WEIGHT DISTRIBUTION OF KERNEL FEATURES

Figure 7 shows the weight distribution of 54 kernel features from Number 3 to Number 56 in Table 1. Six **task_state** features achieve small weight values which are less than 0.05. The results of CrowdNet are gathered from 16 computing nodes that are operated by Apache Mesos, which indicates the six **task_state** features are less relevant to malware detection than others. For **mem_info** features, we can see according to our CrowdNet method, 16 weight values are between 0.5 and 1.0 and 24 values are between 0 and 0.5. It can be observed that the benefit of **mem_info** features is more significant than other features in malware detection experiments. To investigate 112 features' support, we have also collected the weight distribution of other kernel features. Figure 8 describes the weight distribution of the remaining kernel features from Number 57 to Number 114 shown in Table 1, **sche_info**, **signal_info** and **others**. For **sche_info**, CrowdNet achieves 4 values between 0.5 and 1.0 and 7 values between 0 and 0.5. Compared to **sche_info** features, CrowdNet obtains 8 values between 0.5 and 1.0 and 12 weight values between 0 and 0.5 for **signal_info** features. There are 2 values between 0 and 0.5 for **others** features. The non-zero weight values are principal unit of neural networks. They decide how much influence of input values works on output values. Therefore, the weight values of all kernel features show the relative importance to input values and classification performance.

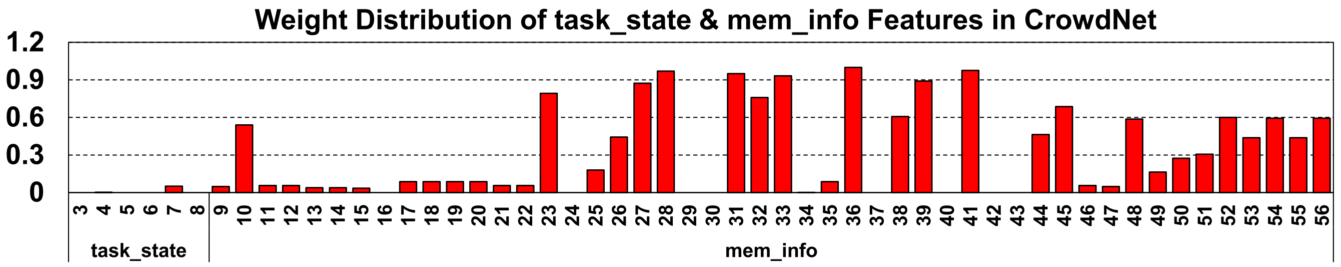


FIGURE 7. Weight distribution of task_state and mem_info features: x-axis denotes the number and category of kernel features in Table 1 from number 3 to number 56 and y-axis denotes the weight value of these kernel features.

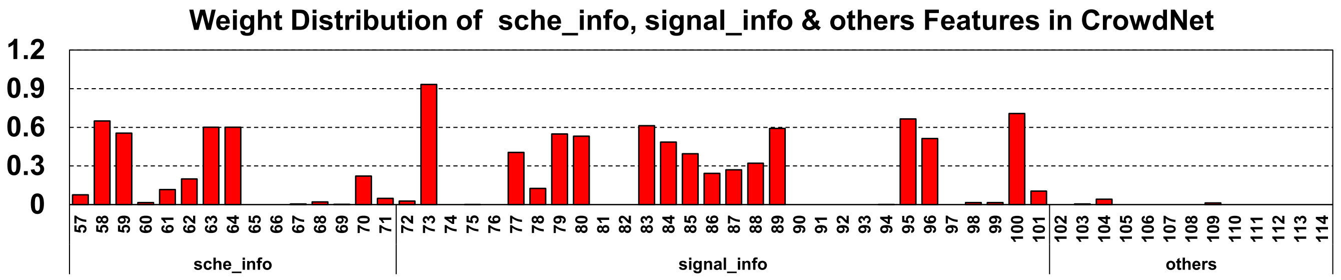


FIGURE 8. Weight distribution of sche_info, signal_info and others features: x-axis denotes the number and category of kernel features in table 1 from number 57 to number 114 and y-axis denotes the weight value of these kernel features.

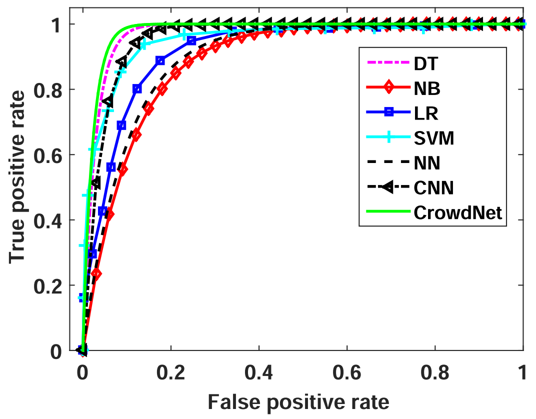


FIGURE 9. ROC curves of seven methods.

C. RECEIVER OPERATING CHARACTERISTIC

Figure 9 illustrates the ROC space of seven methods, DT, NB, LR, SVM, NN, CNN and CrowdNet. Overall, AUC area of all seven methods initially increases with the more false positive rates, then cease to increase at the largest false positive rate. The method owning the largest AUC area in Figure 9 theoretically makes the best classification performance. Here, CrowdNet can be perceived as the first classifier and DT can be the second classifier based on the results of Figure 9. In contrast with previous results in Figure 4, CrowdNet preserves the classification performance with the largest AUC area. We also observe that the ROC curve of DT technique as a reliable classifier increases more sharply and smoothly. Compared to DT, CNN, SVM and LR methods, CrowdNet has a larger AUC area. The traditional NN method only achieves a lower discrimination with a smaller AUC value and NB classifier has the smallest AUC area than other methods. Hence, the benefit of CrowdNet becomes more and more obvious with different true positive rates and false positive rates. To reflect the effect of CrowdNet, we experimentally compare the execution time and classification performance in Sections VI-D and VI-F.

D. EXECUTION TIME OF SEVEN METHODS

To better understand the performance of the seven popular methods, we analyze the major overhead dominating the execution time. The execution time of each method generally changes with different sizes of datasets and different numbers of computing nodes. Since the execution time is sensitive to the size of data samples and the characterization of parallel platforms, we have from 1GB to 8GB datasets concurrently running on from 1 node to 16 nodes and separately compare their results which can serve as guidelines for security practitioners to select the one that best fit their system requirements (8GB is the maximum size for our 12,750 apps and 16 is the maximum number of computing nodes allocated to our distributed tasks).

We first investigate DT method on from 1 computing node to 16 computing nodes in Figure 10. Its execution time with

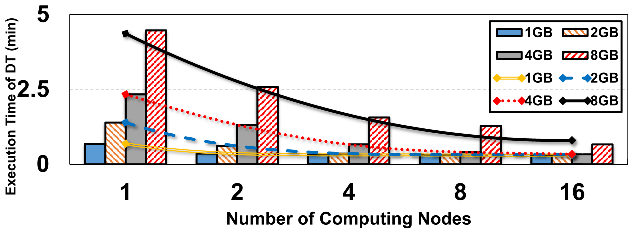


FIGURE 10. Execution time (min) of DT method.

1GB dataset constantly stays at 0.4 minute while increasing the number of computing nodes (workers) from 1 to 16. As expected, training 2GB dataset consumes a double time over training 1GB dataset. This is because more data samples are chopped in the same segments and these segments are assigned to the same computing workers. Since more computing resources are allocated to each computing node for the processing of 2GB dataset, the execution time over multiple nodes is significantly shortened by faster CPUs and larger memory. On the other hand, when the size of dataset increases to 8GB, the execution time of CrowdNet in Figure 16 is comparable to that of DT between 1 and 2 minutes. Figure 11 compares the execution time of LR method with different sizes of datasets and different numbers of computing nodes. Each node processes at least 125MB data when LR method is working on the cluster concurrently. As we can see from Figure 11, the execution time of LR method increases at a much faster rate compared with CrowdNet method in Figure 16. Since it takes a long time to eventually reach convergence for our large-scale data, the execution time of LR is significantly prolonged by the slow processes. To further investigate the main factors of other methods, we systematically analyze the execution time of SVM in Figure 12. The execution time of SVM is comparable to that of LR, but is much longer than our CrowdNet. This is because both techniques perform a probabilistic model and minimize time cost based on likelihood ratio. Its execution time with 1GB and 2GB datasets varies slightly, but for 4GB and 8GB datasets, the number of computing nodes as a potential factor affects its execution time.

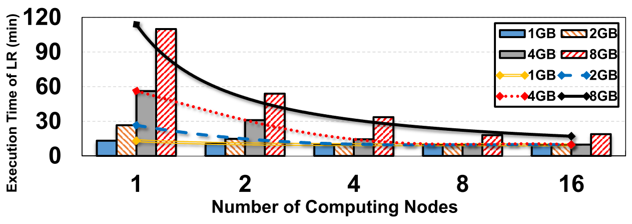


FIGURE 11. Execution time (min) of LR method.

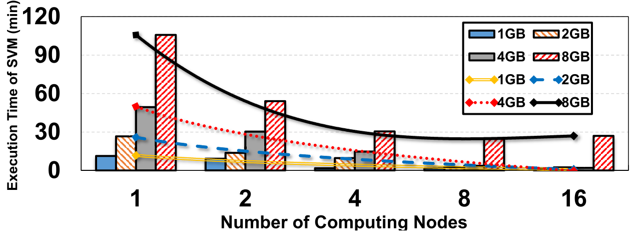


FIGURE 12. Execution time (min) of SVM method.

Figure 13 reveals the execution time of NB for large-scale data. NB consumes the least execution time between the seven methods, but it delivers a poor accuracy performance. This is because NB only needs to compute the frequency of each feature instead of going through all the data samples. However, for quantitative data from Android apps NB

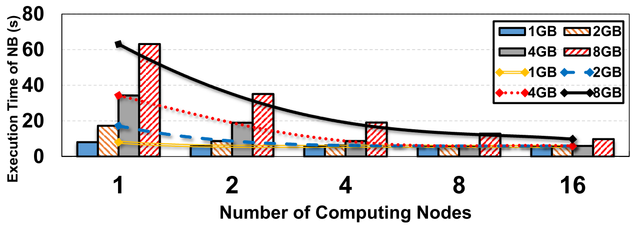


FIGURE 13. Execution time (s) of NB method.

does not incorporate feature characterizations. Meanwhile, we have dissected the execution time of NN and CNN methods, shown in Figures 14 and 15 respectively. CNN and NN outperform computing jobs in a comparable time period. Compared to NN that accomplishes 8GB data processing tasks of varying complexity with 16 nodes in 6.5 minutes, CrowdNet strategically delivers a better performance, for example, 1.7 minutes with 16 nodes. Like NN, CNN also delivers a little lower execution time by 3.4 minutes with 16 nodes. Figure 16 shows the execution time of our CrowdNet with an increasing number of computing nodes. For 8GB dataset, CrowdNet just needs at most 20 minutes to complete the entire task by processing and combining metadata

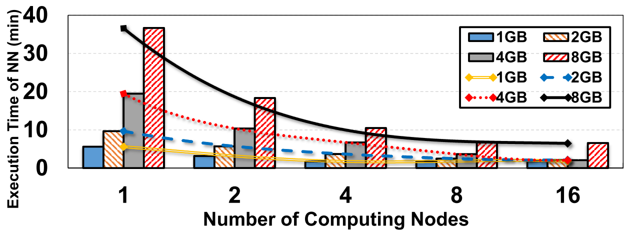


FIGURE 14. Execution time (min) of NN method.

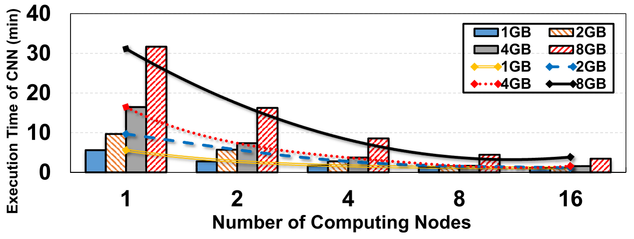


FIGURE 15. Execution time (min) of CNN method.

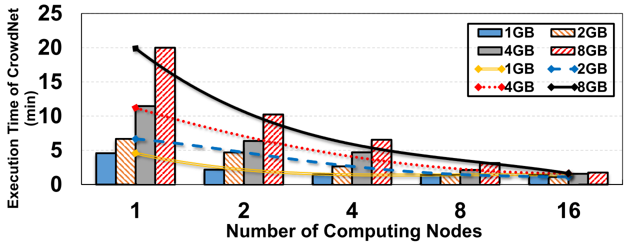


FIGURE 16. Execution time (min) of CrowdNet method.

concurrently. Although CrowdNet initially consumes a little longer execution time with 1 computing node, it eventually alleviates contentions between resources and achieves even more compelling execution time than other methods.

In summary, the execution time of the seven methods with a small dataset (1GB Dataset) does not change much while increasing the number of computing nodes (workers), since only a computing node with 12 cores and 20GB memory can process 1GB data computation. Obviously, 1 computing node increments gradually the overhead time with the increase of the size of datasets from 1GB to 8GB. From our experimental results and spark configuration, we can see 1 computing node (worker) can accomplish the computation of 1GB dataset. Therefore, 2GB dataset can be processed by 2 computing nodes, denoted as: 2GB $\Rightarrow$ 2 workers, similarly, 4GB $\Rightarrow$ 4 workers, 8GB $\Rightarrow$ 8 workers. SVM and LR in Figures 11 and 12 are of the same order of magnitude for execution time due to iterations of linear computation. We can see that NN and CNN deliver longer execution time than CrowdNet in Figures 14 and 15. NB in Figure 13 performs the poor prediction performance with the least execution time. In contrast, CrowdNet and DT in Figures 16 and 10 preserve the higher performance with less execution time.

E. OVERHEAD OF CROWDNET METHOD

Figure 17 shows the memory usage of CrowdNet method. We can see that the CrowdNet method introduces additional overhead of memory storage from 200s to 600s because of loading massive data into memory. However, the extra storage cost does not claim resources for training and testing of our massive dataset in Figure 18. In 600s CrowdNet requests 5% memory resource to load our massive data to the parallel memory from the disk. After 600s, CrowdNet continues to calculate the cluster centers of massive data and select the kernel width with about 2000% CPU resources and 15% memory resources. In our experiments, we launch multiple parallel tasks on different numbers of CPUs. Hence the highest CPU usage can run at 100% $\times$ #. of CPUs. To enhance the reliability and accuracy of massively parallel processing, CrowdNet iteratively trains the classification model in memory and efficiently gathers the information of input parameters. Due to numerical computation of massive data, CrowdNet leads to 15% memory in use between 700s and 1100s in Figure 17. But when finishing this task, the memory usage reduces to 12% for learning the classification model. Meanwhile, to enhance parallelism and reduce execution time, CrowdNet consistently causes high CPU usage between 600s and 1300s in Figure 18. After CrowdNet finally completes computational processes in 1300s, it frees up CPU usage and avoids low computational efficiency. However, we can find the memory usage in Figure 17 is not reduced to 0% because CrowdNet initially generates parallel in-memory data instances. Thereby we have to fully clean up the memory space by terminating the parallel computing processes after 1500s. CrowdNet briefly results in a little higher memory usage and a little more complicated CPU usage.

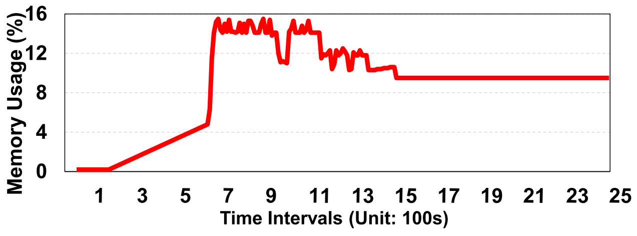


FIGURE 17. Memory usage of CrowdNet.

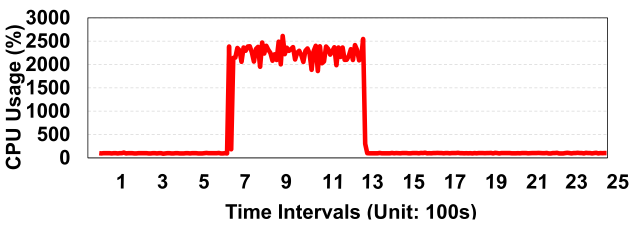


FIGURE 18. CPU usage of CrowdNet.

F. CLASSIFICATION PRECISION

To achieve a better classification precision, we train and test the prediction models with 8GB dataset and 16 computing nodes. With a small number of iterative calculations, the seven classification methods offer the comparable precision results based on different numbers of computing nodes. Figure 19 reveals the precision of the seven techniques. On average, DT, NB, LR, SVM, NN, CNN and CrowdNet preserve the accuracy rates of 91%, 81%, 89%, 88%, 82%, 90% and 94%, respectively. It is obvious that CrowdNet achieves a better precision with less computation overhead compared to other six classifiers. It is because unlike the traditional methods that deal with the entire dataset, CrowdNet strives to aggregate massive data samples and determine few CrowdNet centers. By parallelizing the prediction jobs, all tasks are able to concurrently work on different computing nodes to reduce the I/O overhead. Although the massive parallelism shuffles each task to its corresponding computing node, the precision performance of each method is less likely to be impacted by the characterization of distributed computing. In contrast, CrowdNet associates discrete data samples with each center that can save the intermediate

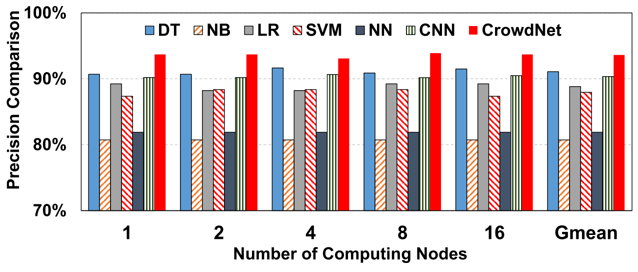


FIGURE 19. Precision comparison of DT, NB, LR, SVM, NN, CNN and CrowdNet.

results, thereby improving the efficiency at scale. Meanwhile, while CrowdNet iteratively reduces the mean square errors with gradient descent in parallel, it is likely to deliver suboptimal performance in aggregation operations because it avoids overshooting the minimum. Therefore, it can be perceived that CrowdNet is a good choice for anomaly detection in Android.

VII. CONCLUSION

In this paper, we propose a CrowdNet framework to accurately predict massive Android malware. It elaboratively implements the collection, storage, and transferring of the large-scale dataset and then efficiently deals with the original data instances from the data provider and precisely predicts malicious behaviors with multiple techniques. To the end, this paper demonstrates the sensitiveness of DT, SVM, LR, NB, NN, CNN and CrowdNet, of which CrowdNet can preserve the best precision and eliminate the execution cost. Moreover, our CrowdNet technique improves the classification performance when the data size dramatically increases and reduces the time consumption caused by frequent I/O communications.

REFERENCES

[1] R. Simpson. (2019). *Mobile Operating System Market Share Worldwide*. [Online]. Available: http://gs.statcounter.com/os-market-share/mobile

[2] M. Garnaeva, J. Wiel, D. Makrushin, and Y. N. A. Ivanov. (2018). *Kaspersky Security Bulletin 2018*. [Online]. Available: https://go.kaspersky.com/rs/802-IJN-240/images/KSB_statistics_2018_eng_final.pdf

[3] H. Cai, N. Meng, B. Ryder, and D. Yao, "DroidCat: Effective Android malware detection and categorization via app-level profiling," *IEEE Trans. Inf. Forensics Security*, vol. 14, no. 6, pp. 1455–1470, Jun. 2019.

[4] J. Allen, M. Landen, S. Chaba, Y. Ji, S. P. H. Chung, and W. Lee, "Improving accuracy of Android malware detection with lightweight contextual awareness," in *Proc. 34th Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2018, pp. 210–221.

[5] D. Li, Z. Wang, and Y. Xue, "Fine-grained Android malware detection based on Deep Learning," in *Proc. IEEE Conf. Commun. Netw. Secur. (CNS)*, May 2018, pp. 1–2.

[6] W. Y. Lee, J. Saxe, and R. Harang, "SeqDroid: Obfuscated Android malware detection using stacked convolutional and recurrent neural networks," in *Deep Learning Applications for Cyber Security*. New York, NY, USA: Springer, 2019, pp. 197–210.

[7] W. Wang, M. Zhao, and J. Wang, "Effective Android malware detection with a hybrid model based on deep autoencoder and convolutional neural network," *J. Ambient Intell. Hum. Comput.*, vol. 10, no. 8, pp. 3035–3043, Aug. 2019.

[8] T. Isohara, K. Takemori, and A. Kubota, "Kernel-based behavior analysis for Android malware detection," in *Proc. 7th Int. Conf. Comput. Intell. Secur.*, Dec. 2011, pp. 1011–1015.

[9] F. Shahzad, M. Akbar, S. Khan, and M. Farooq, "Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on android," Nat. Univ. Comput. Emerg. Sci., Islamabad, Pakistan, Tech. Rep., 2013.

[10] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "'Andromaly': A behavioral malware detection framework for Android devices," *J. Intell. Inf. Syst.*, vol. 38, no. 1, pp. 161–190, 2012.

[11] M. A. Alsheikh, D. Niyato, S. Lin, H.-P. Tan, and Z. Han, "Mobile big data analytics using deep learning and apache spark," *IEEE Netw.*, vol. 30, no. 3, pp. 22–29, May 2016.

[12] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein, and T. Condie, "Adding data provenance support to Apache Spark," *VLDB J.-Int. J. Very Large Data Bases*, vol. 27, no. 5, pp. 595–615, Oct. 2018.

[13] Q. Zhang, L. T. Yang, Z. Chen, and P. Li, "A survey on deep learning for big data," *Inf. Fusion*, vol. 42, pp. 146–157, Jul. 2018.

[14] N. Smyth, *Android Studio 2 Development Essentials*. New York, NY, USA: eBookFrenzy, 2016.

[15] W. Qiang, J. Yang, H. Jin, and X. Shi, "PrivGuard: Protecting sensitive kernel data from privilege escalation attacks," *IEEE Access*, vol. 6, pp. 46584–46594, 2018.

[16] G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: A multi-level anomaly detector for Android malware," in *Proc. Int. Conf. Math. Methods, Models, Archit. Comput. Netw. Secur.* New York, NY, USA: Springer, 2012, pp. 240–253.

[17] H.-H. Kim and M.-J. Choi, "Linux kernel-based feature selection for Android malware detection," in *Proc. 16th Asia–Pacific Netw. Oper. Manage. Symp.*, Sep. 2014, pp. 1–4.

[18] L. Xue, C. Qian, H. Zhou, X. Luo, Y. Zhou, Y. Shao, and A. T. Chan, "NDroid: Toward tracking information flows across multiple Android contexts," *IEEE Trans. Inf. Forensic Security*, vol. 14, no. 3, pp. 814–828, Mar. 2019.

[19] S. Luo, Z. Liu, B. Ni, H. Wang, H. Sun, and Y. Yuan, "Android malware analysis and detection based on attention-CNN-LSTM," *J. Comput.*, vol. 14, no. 1, pp. 31–44, 2019.

[20] H. Alptekin, C. Yildizli, E. Savas, and A. Levi, "TRAPDROID: Bare-metal Android malware behavior analysis framework," in *Proc. 21st Int. Conf. Adv. Commun. Technol. (ICACT)*, Feb. 2019, pp. 664–671.

[21] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. D. Joseph, R. H. Katz, S. Shenker, and I. Stoica, "Mesos: A platform for fine-grained resource sharing in the data center," in *Proc. NSDI*, vol. 11, 2011, p. 22.

[22] X. Pan, X. Wang, Y. Duan, X. Wang, and H. Yin, "Dark hazard: Learning-based, large-scale discovery of hidden sensitive operations in Android apps," in *Proc. NDSS*, 2017.

[23] Z. Meng, Y. Xiong, W. Huang, F. Miao, T. Jung, and J. Huang, "Divide and conquer: Recovering contextual information of behaviors in Android apps around limited-quantity audit logs," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng., Companion (ICSE-Companion)*, May 2019, pp. 230–231.

[24] S. Roy, J. Deloach, Y. Li, N. Herndon, D. Caragea, X. Ou, V. P. Ranganath, H. Li, and N. Guevara, "Experimental study with real-world data for Android app security analysis using machine learning," in *Proc. 31st Annu. Comput. Secur. Appl. Conf. (ACSAC)*, 2015, pp. 81–90.

[25] M. Chattopadhyay, P. K. Dan, and S. Mazumdar, "Application of visual clustering properties of self organizing map in machine–part cell formation," *Appl. Soft Comput.*, vol. 12, no. 2, pp. 600–610, 2012.

[26] J. L. Reyes-Ortiz, L. Oneto, and D. Anguita, "Big data analytics in the cloud: Spark on Hadoop vs MPI/OpenMP on Beowulf," *Procedia Comput. Sci.*, vol. 53, pp. 121–130, Jan. 2015.

[27] A. P. Bradley, "The use of the area under the ROC curve in the evaluation of machine learning algorithms," *Pattern Recognit.*, vol. 30, no. 7, pp. 1145–1159, Jul. 1997.

[28] T. Fawcett, "An introduction to ROC analysis," *Pattern Recognit. Lett.*, vol. 27, no. 8, pp. 861–874, Jun. 2006.

[29] S. Sun, X. Fu, H. Ruan, X. Du, B. Luo, and M. Guizani, "Real-time behavior analysis and identification for Android application," *IEEE Access*, vol. 6, pp. 38041–38051, 2018.

[30] H.-J. Zhu, Z.-H. You, Z.-X. Zhu, W.-L. Shi, X. Chen, and L. Cheng, "DroidDet: Effective and robust detection of Android malware using static analysis along with rotation forest model," *Neurocomputing*, vol. 272, pp. 638–646, Jan. 2018.

[31] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," *HotCloud*, vol. 10, p. 10, Jun. 2010.

[32] *Apache Cassandra*. Accessed: 2010. [Online]. Available: http://cassandra.apache.org

[33] H.-S. Ham and M.-J. Choi, "Analysis of Android malware detection performance using machine learning classifiers," in *Proc. Int. Conf. ICT Converg. (ICTC)*, Oct. 2013, pp. 490–495.

[34] L.-P. Yuan, W. Hu, T. Yu, P. Liu, and S. Zhu, "Towards large-scale hunting for Android negative-day malware," in *Proc. 22nd Int. Symp. Res. Attacks, Intrusions Defenses*, 2019, pp. 533–545.

[35] Y. Zhang, Y. Sui, S. Pan, Z. Zheng, B. Ning, I. Tsang, and W. Zhou, "Familial clustering For weakly-labeled Android malware using hybrid representation learning," *IEEE Trans. Inf. Forensics Security*, to be published.

[36] Y. Zhang, W. Ren, T. Zhu, and Y. Ren, "SaaS: A situational awareness and analysis system for massive Android malware detection," *Future Gener. Comput. Syst.*, vol. 95, pp. 548–559, Jun. 2019.

[37] V. Afonso, A. Bianchi, Y. Fratantonio, A. Doupe, M. Polino, P. De Geus, C. Kruegel, and G. Vigna, "Going native: Using a large-scale analysis of Android apps to create a practical native-code sandboxing policy," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2016, pp. 1–15.

[38] I. J. Mojica, B. Adams, M. Nagappan, S. Dienst, T. Berger, and A. E. Hassan, "A large-scale empirical study on software reuse in mobile apps," *IEEE Softw.*, vol. 31, no. 2, pp. 78–86, Mar. 2014.

[39] D. Sounthiraraj, J. Sahs, G. Greenwood, Z. Lin, and L. Khan, "SMV-HUNTER: Large scale, automated detection of SSL/TLS man-in-the-middle vulnerabilities in Android apps," in *Proc. Netw. Distrib. Syst. Secur. Symp.*, 2014.

[40] Y. Li, J. Liu, Q. Li, and L. Xiao, "Mobile cloud offloading for malware detections with learning," in *Proc. IEEE Conf. Comput. Commun. Workshops (INFOCOM WKSHPS)*, Apr. 2015, pp. 197–201.

[41] H. Huang, C. Zheng, J. Zeng, W. Zhou, S. Zhu, P. Liu, S. Chari, and C. Zhang, "Android malware development on public malware scanning platforms: A large-scale data-driven study," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 1090–1099.

[42] R. Paranthaman and B. Thuraisingham, "Malware collection and analysis," in *Proc. IEEE Int. Conf. Inf. Reuse Integr. (IRI)*, Aug. 2017, pp. 26–31.

[43] J. Deloach, D. Caragea, and X. Ou, "Android malware detection with weak ground truth data," in *Proc. IEEE Int. Conf. Big Data (Big Data)*, Dec. 2016, pp. 3457–3464.

[44] L. Li, T. F. Bissyandé, D. Octeau, and J. Klein, "Reflection-aware static analysis of Android apps," in *Proc. 31st IEEE/ACM Int. Conf. Automated Softw. Eng. (ASE)*, 2016, pp. 756–761.

[45] R. Goyal, A. Spognardi, N. Dragoni, and M. Argyriou, "SafeDroid: A Distributed Malware Detection Service for Android," in *Proc. IEEE 9th Int. Conf. Service-Oriented Comput. Appl. (SOCA)*, Nov. 2016, pp. 59–66.

[46] J. C. B. Quintero. *Virus Total*. Accessed: 2004. [Online]. Available: https://www.virustotal.com

[47] *Google Play*. Accessed: 2012. [Online]. Available: https://play.google.com/store?hl=en

[48] *Google Play Downloader*. Accessed: 2015. [Online]. Available: https://github.com/matlink/gplaycli

[49] P.-N. Tan, M. Steinbach, and V. Kumar, *Introduction to Data Mining*, vol. 1. Reading, MA, USA: Addison-Wesley, 2006.

[50] A. Kusiak, "Feature transformation methods in data mining," *IEEE Trans. Electron. Packag. Manufact.*, vol. 24, no. 3, pp. 214–221, Jul. 2001.

[51] S. Chen, C. Cowan, and P. Grant, "Orthogonal least squares learning algorithm for radial basis function networks," *IEEE Trans. Neural Netw.*, vol. 2, no. 2, pp. 302–309, Mar. 1991.

[52] C. M. Bishop, *Neural Networks for Pattern Recognition*. New York, NY, USA: Oxford Univ. Press, 1995.

[53] F. D. A. De Carvalho, E. C. Simões, L. V. Santana, and M. R. Ferreira, "Gaussian kernel c-means hard clustering algorithms with automated computation of the width hyper-parameters," *Pattern Recognit.*, vol. 79, pp. 370–386, Jul. 2018.

[54] S. S. Du, X. Zhai, B. Poczos, and A. Singh, "Gradient descent provably optimizes over-parameterized neural networks," 2018, *arXiv:1810.02054*. [Online]. Available: https://arxiv.org/abs/1810.02054

[55] L. Armijo, "Minimization of functions having Lipschitz continuous first partial derivatives," *Pacific J. Math.*, vol. 16, no. 1, pp. 1–3, Jan. 1966.

[56] *Apache Mesos*. Accessed: 2019. [Online]. Available: http://archive.apache.org/dist/mesos/0.27.1

XINNING WANG received the B.S. and M.E. degrees from the Ocean University of China, Qingdao, China, in 2009 and 2012, respectively, and the Ph.D. degree from the Department of Computer Science and Software Engineering, Auburn University, in 2017. She is currently a Postdoctoral Research Fellow with the Ocean University of China. Her research interests include spanning data mining and analytics, computer architecture and systems, cloud computing, machine learning, and cybersecurity.

CHONG LI received the B.E. and M.E. degrees from the Ocean University of China, Qingdao, China, in 2009 and 2012, respectively, and the Ph.D. degree from Auburn University, Auburn, AL, USA, in 2016. He was a Postdoctoral Research Fellow with the Integrated MEMS Laboratory (IMEMS), Center for MEMS and Microsystems Technologies (CMMT), Georgia Tech, from 2016 to 2018. He is currently an Associate Professor with the Department of Automation & Measurement, Ocean University of China. His research interests include MEMS, control systems, high-performance computing, machine learning, and artificial intelligence.

DALEI SONG received the Ph.D. degree from the Harbin Institute of Technology, Harbin, Heilongjiang, China, in 1999.

He is currently a Professor with the Department of Automation & Measurement, Ocean University of China, Qingdao, China. His research interests include control systems, robotics technology, high-performance computing, machine learning, and artificial intelligence.