ANDROID MALWARE CLASSIFICATION USING PARALLELIZED MACHINE LEARNING METHODS

by

Lifan Xu

A dissertation submitted to the Faculty of the University of Delaware in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Computer Science

Spring 2016

ANDROID MALWARE CLASSIFICATION USING

PARALLELIZED MACHINE LEARNING METHODS

by

Lifan Xu

Approved: _____

Kathleen F. McCoy, Ph.D.
Chair of the Department of Computer and Information Sciences

Approved: _____

Babatunde A. Ogunnaike, Ph.D.
Dean of the College of Engineering

Approved: _____

Ann L. Ardis, Ph.D.
Senior Vice Provost for Graduate and Professional Education

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

John Cavazos, Ph.D.
Professor in charge of dissertation

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Chien-Chung Shen, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Haining Wang, Ph.D.
Member of dissertation committee

I certify that I have read this dissertation and that in my opinion it meets the academic and professional standard required by the University as a dissertation for the degree of Doctor of Philosophy.

Signed: _____

Dongping Zhang, Ph.D.
Member of dissertation committee

ACKNOWLEDGEMENTS

I would never have been able to finish this dissertation without the guidance of my advisor and my committee members, support from my family and wife, and help from my friends and lab mates.

I am immensely grateful to my advisor, Prof. John Cavazos, for his guidance and support throughput my PhD years. I would like to thank Dr. Marco A. Alvarez for guiding my research in the past several years and helping me to develop my background in graph representation and graph kernels. I also would like to thank Dr. Dong Ping Zhang for her caring, advice, and helping me to improve my knowledge in high performance computing, processing in memory, and deep learning.

I would like to thank many friends who have helped me and enriched my life, particularly, William Killian, Sameer Kulkarni, Eunjung Park, Robert Searles, Tristan Vanderbruggen, Wei Wang and many others.

I would also like to thank my parents, two elder sisters, and elder brother. They have brought me tremendous love since I was born.

Finally, I would like to thank my son Aiden Xu, my daughter Lora Xu, and my wife Ruoxin Peng. They were always there cheering me up and accompanied me through good and bad times. This dissertation is dedicated to them.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Android is the most popular mobile operating system with a market share of over 80% [59]. Due to its popularity and also its open source nature, Android is now the platform most targeted by malware, creating an urgent need for effective defense mechanisms to protect Android-enabled devices.

In this dissertation, we present a novel characterization and machine learning method for Android malware classification. We first present a method of dynamically analyzing and classifying Android applications as either malicious or benign based on their execution behaviors. We invent novel graph-based methods of characterizing an application's execution behavior that are inspired by traditional vector-based characterization methods. We show evidence that our graph-based techniques are superior to vector-based techniques for the problem of classifying malicious and benign applications.

We also augment our dynamic analysis characterization method with a static analysis method which we call HADM, **H**ybrid **A**nalysis for **D**etection of **M**alware. We first extract static and dynamic information, and convert this information into vector-based representations. It has been shown that combining advanced features derived by deep learning with the original features provides significant gains [73]. Therefore, we feed each of the original dynamic and static feature vector sets to a Deep Neural Network (DNN) which outputs a new set of features. These features are then concatenated with the original features to construct DNN vector sets. Different kernels are then applied onto the DNN vector sets. We also convert the dynamic information into graph-based representations and apply graph kernels onto the graph sets. Learning results from various vector and graph feature sets are combined using hierarchical Multiple Kernel Learning (MKL) [37] to build a final hybrid classifier.

Graph-based characterization methods and their associated machine learning algorithm tend to yield better accuracy for the problem of malware detection. However, the graph-based machine learning techniques we use, i.e., graph kernels, are computationally expensive. Therefore, we also study the parallelization of graph kernels in this dissertation. We first present a fast sequential implementation of the graph kernel. Then, we explore two different parallelization schemes on the CPU and four different implementations on the GPU. After analyzing the advantages of each, we present a hybrid parallel scheme, which dynamically chooses the best parallel implementation to use based on characteristics of the problem.

In the last chapter of this dissertation, we explore parallelizing deep learning on a novel architecture design, which may be prevalent in the future. Parallelization of deep learning methods has been studied on traditional CPU and GPU clusters. However, the emergence of Processing In Memory (PIM) with die-stacking technology presents an opportunity to speed up deep learning computation and reduce energy consumption by providing low-cost high-bandwidth memory accesses. PIM uses 3D die stacking to move computations closer to memory and therefore reduce data movement overheads. In this dissertation, we study the parallelization of deep learning methods on a system with multiple PIM devices. We select three representative deep learning neural network layers: the convolutional, pooling, and fully connected layers, and parallelize them using different schemes targeted to PIM devices.

Chapter 1

INTRODUCTION

With over 260 million shipments, Android has dominated the smart phone market with a 78.0% share in the first quarter of 2015 [46]. Unfortunately, the growing popularity of Android smart phones and tablets has made this popular OS a prime target for security attacks. In 2014, nearly one million unique malicious applications were produced, a 391% increase from 2013. Some estimates say that Android has been targeted by 97% of the developed mobile malware [64], creating an urgent need for effective defense mechanisms to protect Android-enabled devices.

Researchers have proposed various characterization methods to counter the increasing amount and sophistication of Android malware. These methods can be categorized into: static analysis, dynamic analysis, and hybrid techniques. Static analysis is based on extracting features by inspecting an application's manifest and disassembled code [91, 38, 10, 98, 94]. By contrast, dynamic analysis methods monitor the application's behavior during its execution [32, 17, 93, 80, 68, 84, 30]. Hybrid methods typically analyze an application before installation and also record its execution behavior [14, 100, 83, 56, 89, 55, 99]. These sets of static and dynamic information are then used together to detect malicious behavior. Static analysis is usually lightweight and can be performed on a user's device while dynamic analysis is usually performed in an offline emulator due to simulation overhead. Static and dynamic analysis both have their disadvantages. Static analysis techniques can be defeated by malware packing and other malware obfuscation techniques. On the other hand, dynamic analysis techniques can be defeated if the malware notices it is running in an emulator or sandboxed environment [66]. The hybrid analysis method is gaining more popularity for

its combined advantages from both static and dynamic analysis and its capability to yield better accuracy in detecting malware.

This dissertation first proposes a dynamic-analysis-based method for Android malware classification. We use the Linux tool strace [4] to dynamically collect system calls for each application during its execution. We then develop a set of scripts to incorporate the low level information collected by the strace utility into a data representation that can be used with machine learning. One elegant way to represent this low level information is by using graphs. Instead of employing a traditional flat feature vector representation, graphs allow us to represent the data in a more structured manner. In this dissertation, we evaluate three traditional feature vector representations including histograms, n-grams, and the Markov Chains. Histograms count the invocations of each system call while n-grams count the invocations of n contiguous system calls. The Markov Chains are directed graphs where the vertices are the system calls and the edges are the transition probabilities between system calls. We then propose novel graph representations based on each of these vector representations, namely the Histogram System Call Graph (HSCG), the N-gram System Call Graph (NSCG), and the Markov Chain System Call Graph (MCSCG). In these graphs, each vertex is a process belonging to the corresponding application. The root node of each graph is the main process which is the first process created for the application during its startup. The graph is then formed by collecting processes with direct ancestral lineage to the main process and then connecting parent/children processes. Different graphs have different labels for their vertices. For HSCG, the label is a histogram vector; for NSCG, the label is an n-gram vector; and for MCSCG, the label is a Markov Chain vector. We also explore two other graphs, the Ordered System Call Graph (OSCG) and the Unordered System Call Graph (USCG), in which each system call invocation is treated as a vertex and labeled with they system call name. In the OSCG, each vertex is connected to the previous vertex of the same process. The first vertex of one process is connected to the system call spawned this process. In the USCG, all vertices of one process are connected to the same system call that spawned the corresponding

process. To perform classification, we apply graph kernels on the graph representations and feed the graph similarities into a machine learning model, e.g., Support Vector Machine (SVM), for classification. Similarly, we also feed the feature vectors into an SVM to construct the classification model and compare it with the graph model. Figure 1.1 shows our two different dynamic analysis models.

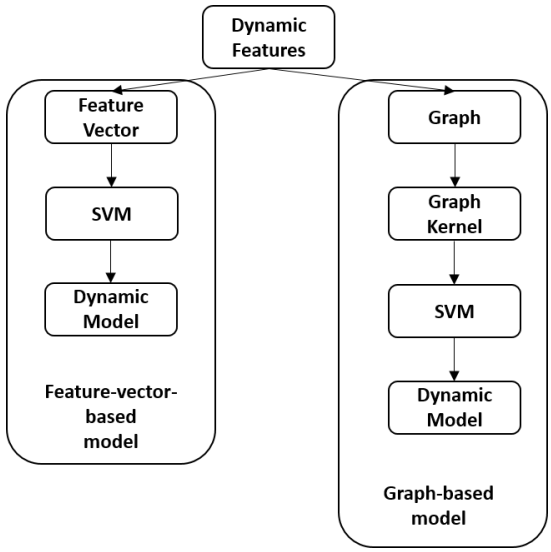


Figure 1.1: This figure shows feature-vector-based model and graph-based model for dynamic Android malware analysis.

Next, we augment our dynamic analysis method with a static analysis method and propose **HADM**, **H**ybrid **A**nalysis for **D**etection of **M**alware. In addition to dynamic execution behavior, we extract a set of static features from each Android application and convert this information into feature vector representations. For each feature vector representation, we train one Deep Neural Network (DNN) and then apply the SVM algorithm to construct a static classification model. Figure 1.2 shows how we build our feature-vector-based static analysis model. To construct our hybrid analysis model, we apply a hierarchical Multiple Kernel Learning (MKL) to combine different static and dynamic models as shown in Figure 1.3.

To evaluate the performance of HADM, we collected thousands of Android applications across all categories of Google Play. We also collected a large number of

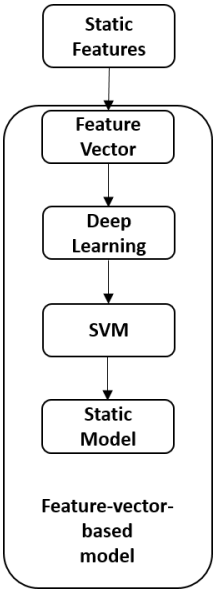


Figure 1.2: This figure shows feature-vector-based model for static Android malware analysis.

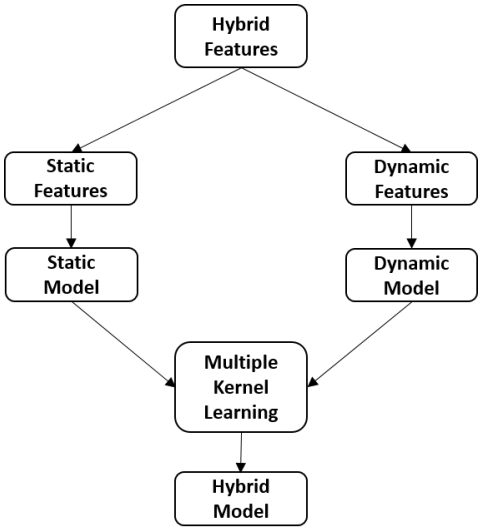


Figure 1.3: Combining static model and dynamic model using multiple kernel learning to construct hybrid Android malware classification model.

Android malware from VirusShare [1]. In our dataset, 4002 samples are categorized as benign applications and 1886 samples are categorized as malware. Experiments on this dataset show, for dynamic features, the best classification accuracy that can be achieved is 83.3% by feature-vector-based representations and 87.3% by graph-based representations. On average, graph-based representations are able to achieve 5.2% absolute classification accuracy improvement over the feature-vector-based representations. For original static feature vector sets, the best classification accuracy that can be achieved is 93.5%. Finally, by applying hierarchical MKL, classification accuracy of the final hybrid classifier is further improved to 94.7%.

For HSCG, NSCG, and MCSCG graph sets, we select the Shortest Path Graph Kernel (SPGK) to compute pairwise similarities. SPGK has been proven more accurate than other graph kernels in other domains [15]. However, given that the running time for SPGK is $O(n^4)$, it may not be appropriate when applied to a large graph dataset. We therefore propose the parallelization of the SPGK graph kernel on multicore CPUs and GPUs. We first split the original shortest path kernel into two parts and present a fast sequential implementation of SPGK which we refer to as Fast Computation of Shortest Path Kernel (FCSP). Then, we explore two different parallelization schemes on the CPU using OpenMP and four different parallel implementations on the GPU using OpenCL. After analyzing the advantages of each we propose a hybrid version, which dynamically predicts and chooses the best implementation between the multicore CPU and the GPU parallel implementation based on the characteristics of the graphs being processed. The results show that the sequential FCSP algorithm running on CPU is able to achieve a maximum speedup of 76x over a naive sequential implementation of the shortest path graph kernel algorithm running on the same CPU. The results also show that our GPU implementation of FCSP offers a maximum 18x speedup over the sequential FCSP. Our GPU implementation also achieves a maximum of 2x speedup over a parallel CPU implementation of FCSP. At the end, we show the hybrid algorithm

[1] http://virusshare.com

works best for all of our datasets.

For vector sets, we train them using deep learning methods. Prior work has successfully accelerated deep learning by mapping the application to existing architectures including CPUs and GPUs [52, 24, 21]. However, relatively little research has been done to evaluate the potential of emerging architecture designs, such as in-memory computing, to improve the performance and energy efficiency of deep learning. Moving data close to computation reduces data movement overhead and therefore speeds up the computation. In this dissertation, we also explore the potential of Processing In Memory (PIM)[2] implemented via 3D die stacking to improve the performance of deep learning. From popular deep learning models, we select three frequently used and representative neural network layers : the convolutional layer, the pooling layer, and the fully connected layer. we parallelize these layers individually across multiple PIM devices. We also evaluate two different parallelization schemes, data parallelism and model parallelism. Experiments show that by scaling deep learning models to multiple PIMs available in a system, we are able to achieve better or competitive performance compared with a high-performance host GPU in many cases across the different layers studied. We show that model parallelism consumes much less memory than data parallelism on fully connected layers, and it also reaches better performance when the number of input images per training batch is small. However, as batch size increases, data parallelism scales better due to the absence of synchronization and it outperforms model parallelism.

The rest of the dissertation is structured as follows. Chapter 2 provides an overview and related work on different Android malware classification methods and parallelizations of graph kernel and deep learning. Chapter 3 describes our dynamic Android malware classification method. Chapter 4 augments the dynamic method with a static analysis method and propose a hybrid method named HADM. Our parallelizations of SPGK is detailed in Chapter 5. Parallelization of deep learning using multiple

[2] This dissertation uses PIM as an abbreviation interchangeably for process*ing* in memory and process*or* in memory depending on the context.

PIM devices are presented in Chapter 6. Conclusions reached from the research in this dissertation are presented in Chapter 7.

Chapter 2

BACKGROUND

Malware analysis and detection is a continuously evolving field of research. Several concepts and techniques have been proposed in the last few years to defend against the increasing amount and sophistication of malware. In general, Android malware can be analyzed using three different methods: static and dynamic analysis and hybrid analysis, which is a combination of both. Static methods mainly focus on extracting features from the manifest and code of the application. Static analysis is usually lightweight but can be circumvented by malware with packing and other obfuscation techniques [66]. Dynamic methods concentrate on scrutinizing behavior of malware during its execution in an emulation environment. Therefore, dynamic analysis is usually done before it is installed on the Android device. However, dynamic analysis techniques can also be circumvented if the malware notices it is running in a simulated environment. The hybrid analysis method is increasing in popularity because it takes advantages of the strengths of both static and dynamic analysis and therefore is able to yield better performance. In this dissertation, we first propose a dynamic analysis method with novel graph-based representation. We then extend our method to hybrid analysis method by augmenting the dynamic analysis with a static analysis method.

2.1 Android Malware Detection using Static Analysis

Static analysis involves extracting information from the application's manifest the Android application's bytecode. The features often used in static analysis include API usage, requested permissions, used permissions, control flow, data flow, hardware components, application components, intents, sensitive API calls, network address, etc.

Schmidt et al. [76] performed static analysis on a common binary file format called Executable and Linking Format (ELF). They performed static analysis on the ELF files to extract function calls using the command readelf. Function call lists were extracted from the malware executables to classify them with Decision Trees (DT), Rule Induction (RI), and Nearest Neighbor (NN) algorithms. The authors collected 240 malware, which targeted Linux systems. All of their algorithms achieved an 80% or higher detection rate.

Wu et al., proposed DroidMat [91], which detects malware by characterizing applications using the manifest file and API call tracing. DroidMat extracts static information including permissions, deployment of components, intent message passing, and API calls to characterize the Android application in a feature vector. Then, it applies the K-means and K-Nearest Neighbor (KNN) algorithms to classify the application as benign or malicious. The authors collected 238 Android malware from Contagio [1], and they were able to reach 97.87% classification accuracy. However, the results were only reported on the training data, and these excellent results are likely due to overfitting.

Peng et al., explored an approach to rank the risk of Android applications using probabilistic generative models [63]. They extracted permissions, specifically the top 20 most frequently requested, in their dataset from manifest files as a key feature. Then, the risk scores were computed using different probabilistic models ranging in complexity from simple naive Bayes through hierarchical mixture of naive Bayes models. They tested their method on 378 malware mixed with different subsets of benign applications. The best configuration had a detection accuracy of 78%.

Sahs et al., trained a one class Support Vector Machine (SVM) on benign applications to detect malicious applications [70]. Permissions required by the application as well as Control Flow Graph (CFG) information are combined and fed into a custom kernel for classification. Experiments on 2081 benign and 91 malicious Android applications showed their method can only detect approximately 50% of the malware.

Grace et al., implemented RiskRanker for zero-day Android malware detection [38]. A two-order risk analysis is performed in RiskRanker. The first-order handles non-obfuscated applications by evaluating the risks in a straightforward manner. The second-order analysis collects and correlates various signs or patterns of behavior common among malware, yet not among benign applications. RiskRanker was applied to examine 118318 Android applications collected from various markets over September and October 2011. It took four days to process all the applications and reported the presence of 3281 risky applications. Among the reported applications, 718 malware samples were uncovered and 322 of them were zero-day malware.

Sanz et al., presented PUMA [71], a system that uses the permissions that the application requests upon installation to detect whether the application is malicious or not. Machine learning models including simple logistic regression, naive Bayes, Bayes net, SMO, IBK, J48, Random Tree (RT) and Random Forests (RF) were evaluated on a dataset consisting of 357 benign and 249 malicious applications. The best overall accuracy was reached by random forests at 86%.

Sanz et al., then improved PUMA and proposed MAMA [72] which employs not only permissions, but also the features under the user-feature tag in the manifest file. Machine learning algorithms including K nearest neighbors, decision trees, Bayes net, and SVMs were evaluated. The best configuration of MAMA was able to reach 94.83% classification accuracy on 333 malicious and 333 benign applications

Glodek et al., extended previous static analysis based work by using combinations of features extracted from the manifest file and bytecodes of the program as training features for a random forest algorithm [36]. In total, 147 features were used that included characteristics such as permission and receiver requests and whether the app included native code. Experiments on 500 malicious and 500 benign samples showed the proposed method can provide true positive rates in excess of 90%.

Gascon et al., proposed a method for malware detection based on efficient embedding of Function Call Graphs (FCG), which are high level characteristics of the applications [35]. The authors extracted function call graphs using the Androguard

framework [29]. The nodes in the graph were labeled according to the type of instructions contained in their respective functions. A neighborhood hash graph kernel was applied to evaluate the count of identical substructures in two graphs. Finally, an SVM algorithm was used for classification. In an evaluation of 12158 malware samples, the proposed method detected 89% of the malware.

Lee et al., presented a detection mechanism using runtime semantic signatures, which are supposed to have high family classification accuracy [54]. The authors used three elements to construct the signature. The first set was binary patterns of malicious API call instructions and their runtime semantics for control and data flow. The second set was the malware family characteristics including family common string, constants, methods, and classes for malware belonging the same family. The third set was weights of each behavior within family. Experiments on 1759 Android malware including 79 variants of 4 malware families showed the proposed method was able to accomplish 99.89% accuracy on detecting the family of a particular variant of malware.

Wolfe et al. [90], proposed a method of screening malicious Android applications that used two types of features: the requested permissions in the manifest and Percentage of Valid Call Sites (PVCS) calculated from a data dependency graph. They used a collection of 1433 malware and 2436 benign applications. For classification, K nearest neighbors, naive Bayes, SVM algorithms, Boosted Decision Trees (BDT), and random forests were evaluated. The best classifier detected 83.75% of the malware from unfamiliar families and 96% of those from familiar families.

Arp et al., proposed DREBIN [10], which is a similar approach to the method proposed by Peng et al. [63]. Eight different static feature sets were extracted by DREBIN including hardware components, requested permissions, application components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network address. However, unlike Peng et al., which used a naive Bayes approach [63], a linear SVM algorithm was used for classification. In an evaluation including 123453 benign applications and 5560 malware samples, DREBIN detected 94% of the malware.

Zhang et al., implemented DroidSIFT [98]. They extracted a weighted contextual API dependency graph as program semantics to construct feature sets. Graph similarity metrics were introduced to uncover homogeneous application behaviors. Experiments on 2200 malware samples and 13500 benign samples were performed using naive Bayes. The results show that DroidSIFT can detect 93% of malware instances.

Yang et al., developed DroidMiner [94] which uses static analysis to automatically mine malicious program logic from known Android malware. A two-tiered behavior graph is constructed in DroidMiner. The upper tier is a Component Dependency Graph (CDG) in which each node represents an activity, service or broadcast receiver. The lower tier uses Component Behavior Graphs (CBG) to represent each component's lifetime behavior functionalities. From the behavior graph, different malicious patterns, named modalities by the authors, can be mined. In particular, function modality, which represents an ordered sequence of API functions, and resource modality, which represents a set of sensitive resources, are extracted and converted to a modality vector by DroidMiner. The vectors are then fed into several machine learning classifiers including naive Bayes, SVM algorithms, decision trees, and random forests for malware detection. The best algorithm of DroidMiner can achieve a 95.3% detection rate on a dataset of 2466 malware. It can also reach 92% for classifying malware into its proper family.

2.2 Android Malware Detection using Dynamic Analysis

Dynamic analysis records the execution behavior of an application and tries to identify malicious behavior. It is well known for being resilient to obfuscation techniques. However, dynamic analysis introduces overhead because it requires running the application first and then deciding if it is malware based on run-time behavior. As a consequence, it is mostly applied to offline detection of malware on a server. One other deficiency of dynamic analysis is code coverage. Since some malicious behaviors are guarded by trigger conditions, dynamic analysis will not record an application's malicious behavior if the conditions are not triggered.

Enck et al., implemented TaintDroid [32], which was the first work to propose taint tracking for monitoring data flow dependencies and data leakage in Android applications. They used TaintDroid to study the behavior of 30 third-party applications. Their study revealed that two thirds of the applications exhibit suspicious handling of sensitive data.

Burguera et al., proposed CrowDroid [17] for identifying repackaged malware using dynamically collected behavior features. Repackaged malware is created by repackaging a benign application with additional malicious code. CrowDroid collects the frequencies of several system calls from several users running the application on different devices using Linux strace [4] tool. The system call histograms are then fed into K-means clustering with $K = 2$ to separate benign applications from repackaged malicious instances. The authors performed experiments on four artificial malware created by the authors and two real world examples. CrowDroid detected all four author-created malware but generated false positives in real world malware.

Shabtai et al., designed Andromaly [80] which uses 88 dynamic features including memory page activity, SMS message events, CPU usage, network usage, touch screen pressure, binder information, battery information, etc. The authors evaluated different combinations of features using information gain and Fisher scores. Features with the best scores were selected. Then applied several classifiers including decision trees, naive Bayes, Bayes nets, histograms, K-means, and logistic regression were applied. For experiments, four artificial malware were used and the best configuration was able to achieve approximately 88% accuracy.

Yan et al., proposed DroidScope [93] which is similar to TaintDroid [32]. While TaintDroid focuses on taint analysis, DroidScope enables introspection at different layers of the platform. By building two levels of semantic information: the operating system and Java, DroidScope enables dynamic instrumentation of both the Dalvik bytecode as well as native instructions. Therefore, the analyst is able to reveal the behavior of a malware sample's Java and native components as well as interactions between them and the rest of the system. Evaluation on two real world Android malware

samples, DroidKungFu and DroidDream from Genome [101], proved the capability of DroidScope. No malware classification experiments were reported in the paper describing DroidScope.

Ham et al. [39] proposed a method that is very similar to CrowDroid. They also aggregated real-time system calls to create a histogram using Linux strace [4] tool. They discovered that some system call patterns can only occur in malicious applications and some only in benign applications. Different from the K-means used in CrowDroid, Ham et al., applied a discrimination algorithm based on Euclidean distance on 1260 malware samples distributed by Genome [101]. No classification accuracy was reported by the authors.

Amos et al. [7] also used hand-selected dynamic features similar to Andromaly. They collected memory, CPU, and binder information and evaluated their method on a dataset consisting of 1330 malware and 408 benign applications. For classification, random forests, naive Bayes, multilayer perceptrons, Bayes nets, logistic regression, and decision tress were applied. The experiments showed they can achieve 95% accuracy on new traces from applications included in the training set and 82% on traces from applications that were not included.

Reina et al., presented CopperDroid [68], an approach built on top of QEMU [12] to automatically perform out-of-the-box dynamic behavioral analysis of Android malware. CopperDroid records system call invocations by instrumenting QEMU to record information when the swi instruction is executed. Binder analysis is also performed in CopperDroid for reconstructing high-level Android-specific behaviors. Experiments on malware distributed by Genome [101] and Contagio [1] were carried out by the authors to assess the effectiveness of CopperDroid. No malware classification experiments were reported in the paper describing CopperDroid.

Google is currently running a detection system called Bouncer. Little is known about it, except that it is a QEMU-based dynamic analysis framework that bounces applications off of the official Google Play market if they are deemed to be malicious.

Tchakounté et al. [85] scrutinized system call invocations initiated by the malicious code at the moment the user runs it using the Linux strace [4] tool. With their tool they discovered new scenarios of how the user can be lured to aid the malicious developer.

Demme et al., examined the feasibility of building a malware detector in hardware using existing performance counters [26] such as arithmetic operations executed and L1 exclusive hits. After embedding the information collected from performance counters into different feature vectors, several machine learning algorithms including K nearest neighbor, decision trees, and random forests were applied for classification. Experiments on 503 malware and 210 non-malware programs from both Android ARM and Intel X86 platforms were performed. Results showed that the robustness and security of hardware anti-virus techniques have the potential to advance state-of-the-art online malware detection.

Wei et al. [88] recorded system call invocations by manually installing and executing each application on a real Android phone. N-gram vectors were generated from the system call invocations and fed into an SVM and a naive Bayes for classification. Experiments on 96 benign applications and 92 digital book malware samples showed their methods can reach 94% accuracy.

Dimjasevic et al. [30] proposed MALINE, which also recorded system call invocations for Android malware and converted them into two representations. One was histogram and the other was a variant of the Markov Chain representation. Experiments on 4289 malware and 12789 benign applications showed they can achieve 93% detection accuracy.

2.3 Android Malware Detection using Hybrid Analysis

Hybrid methods combining static analysis and dynamic analysis have also been done in prior work. These methods typically consist of analyzing the application before installation and also recording the execution behavior. Static and dynamic analysis are then used together for malware detection.

Blasing et al., proposed a tool called the AASandbox [14] that performs both static and dynamic analysis. It was the first system applying hybrid analysis in a very basic way for the Android platform. AASandbox scans the software for malicious patterns without installation. It also intervenes and logs low-level interaction with the system for further analysis during the application execution. An example run was described by the authors to prove the correctness of AASandbox.

Zhou et al., proposed DroidRanger [102], which implements a combination of permission-based behavioral footprinting to detect new samples of already known malware families and a heuristic-based filtering scheme along with dynamic execution monitoring to detect unknown malicious families. Experiments with 204,040 applications collected from different Android markets in June 2011 revealed 32 malware from the official Android market and 179 from alternative marketplaces.

Zheng et al., proposed SmartDroid [100], which is a hybrid analysis method to reveal UI-based trigger conditions in Android applications. SmartDroid first uses static analysis to extract expected activity switch paths by analyzing Activity Call Graphs (ACG) and Function Call Graphs (FCG). ACG graphs contain nodes representing activities. An edge between two activities if there is an intent created to switch activities. Nodes in FCG graphs are sensitive APIs and a node is connected to another node if it is called by the "invoke" instruction. SmartDroid uses dynamic analysis to traverse each UI elements and explore the UI interaction paths towards the sensitive APIs. SmartDroid found 7 malware families among 19 applications in their experiments. No malware classification experiments on a large sample of applications were reported in the paper describing SmartDroid.

Canfora et al., proposed a method for detecting malware based on the occurrences of a specific subset of system calls, a weighted sum of a subset of permissions that the application required, and a set of combinations of permissions [19]. Various machine learning methods including J48, LadTree, NBTree, random forests, random tree, and RepTree were used for classification. Experiments on a dataset made up of 200 benign and 200 malicious Android applications were carried out and the best

algorithm achieved a detection accuracy of 80%.

Lindorfer et al., proposed Andrubis [56, 89, 55], which performs both static and dynamic analysis. During the static analysis stage, Andrubis extracts information from an application's manifest and bytecode such as requested permissions, services, broadcast receivers, activities, package name, SDK version, and a complete list of available Java objects and methods. During dynamic analysis, Andrubis introduces multiple stimulation approaches to increase the exploration of the application's functionality. Taint tracking, method tracing, system call invocation recording are also performed in the dynamic analysis stage. Other than this, auxiliary analysis on network traffic is also carried out. Andrubis provides a web interface for users to submit Android apps, and it has collected a dataset of over one million Android applications including 40% malware.

Spreitzenbarth et al., proposed Mobile-Sandbox [83], which is similar to Andrubis [56, 89, 55]. Mobile-Sandbox first matches the hash value of the testing application against the VirusTotal database. Then it analyzes the manifest to extract permissions, intents, services, and receivers. API calls that happen frequently in malware are also extracted from the Dalvik bytecode. During the dynamic analysis stage, ltrace [3], a common Linux debugging utility that intercepts library calls of a monitored application, is included to track native code. Network traffic is also logged as auxiliary analysis. Similar to Andrubis, Mobile-Sandbox provides a web interface for a user to submit Android apps and returns reports of static and dynamic features. According to the paper describing Mobile-Sandbox, the major difference between Andrubis and Mobile-Sandbox is that the latter can support applications beneath Android API level 11 while Andrubis is limited to applications beneath API level 8. Additionally, Andrubis is not able to track native code.

Yuan et al., proposed Droid-Sec [95]. In Droid-Sec, over 200 features from both static and dynamic analysis are extracted including required permissions, sensitive API,

and dynamic behaviors emulated using DroidBox[1]. The features are represented as a flat feature vector and fed into a DNN for classification. Experiments on 250 malicious and 250 benign applications show Droid-Sec is able to reach 96.5% accuracy.

Yuan et al., then proposed DroidDetector [96]. It uses the same method as proposed in Droid-Sec. In total, 192 features from both static and dynamic analyses of Android applications are extracted by DroidDetector. It then characterizes malware using a DNN-based deep learning model. DroidDetector was evaluated with 20,000 benign samples and 1760 malware samples. It achieved 96.76% detection accuracy.

2.4 Originality of Our Android Malware Analysis Method

There appears to be no existing work that adapts graph-based representation to a dynamic analysis based method. Our dynamic method is different from all the aforementioned methods because we developed novel graph-based representations for Android applications by converting system call invocations to graphs. System calls have been used for malware detection in other methods [85, 68, 17, 93, 39, 88, 30]. However, in all the previous work, one feature vector is constructed for the entire application. We used this as the baseline for our experiments. Our graph-based methods augment the feature-vector-based methods by constructing a graph to represent the caller-callee relationship of different processes.

Comparing with other hybrid analysis-based methods, our method applies deep learning to improve the performance of each characterization techniques we use and combine them using optimal weights computed by MKL.

Table 2.1 summarizes the previous Android malware analysis works and our proposed method.

2.5 Graph Computation Parallelization

Graph-based methods have become an increasingly popular approach for learning patterns from graphs. It has been successfully used on protein classification [15]

[1] https://code.google.com/p/droidbox/

Table 2.1: This table shows the summary of previous Android malware research works.

System	Static	Dynamic	Hybrid	Parallelization	Open Source	Web Interface
Schmidt et al. [76]	Yes			No	No	No
DroidMat [91]	Yes			No	No	No
Peng et al. [63]	Yes			No	No	No
Sahs et al. [70]	Yes			No	No	No
RiskRanker [38]	Yes			No	No	No
PUMA [71]	Yes			No	No	No
MAMA [72]	Yes			No	No	No
Glodek et al. [36]	Yes			No	No	No
Gascon et al. [35]	Yes			No	No	No
Lee et al. [54]	Yes			No	No	No
Wolfe et al. [90]	Yes			No	No	No
DREBIN [10]	Yes			No	Malware Samples	No
DroidSIFT [98]	Yes			No	No	No
DroidMiner [94]	Yes			No	No	No
TaintDroid [32]		Yes		No	Source Code	No
CrowDroid [17]		Yes		No	No	No
Andromaly [80]		Yes		No	Source Code	No
DroidScope [93]		Yes		No	Source Code	No
Ham et al. [39]		Yes		No	No	No
Amos et al. [7]		Yes		No	Source Code	No
CopperDroid [68]		Yes		No	No	Yes
Tchakount et al. [85]		Yes		No	No	No
Demme et al. [26]		Yes		No	No	No
Wei et al. [88]		Yes		No	No	No
MALINE. [30]		Yes		No	Source Code	No
AASandbox [14]			Yes	No	No	No
DroidRanger [102]			Yes	No	No	No
SmartDroid [100]			Yes	No	No	No
Canfora et al. [19]			Yes	No	No	No
Andrubis [56, 89, 55]			Yes	No	No	Yes
Mobile-Sandbox [83]			Yes	No	No	Yes
Droid-Sec [95]			Yes	No	No	No
DroidDetector [96]			Yes	No	No	No
Proposed HADM			**Yes**	**Yes**	**No**	**No**

and image classification [60] [5] [16] [6] [41] [48]. However, the core part of all graph-based classification methods, i.e., graph matching or graph similarity computation can be expensive.

To the best of our knowledge, no parallelization of graph kernels has been published. However, the transformation of a graph into a shortest path graph on the GPU, in particular the Floyd-Warshall algorithm, has been done in the past. Harish and Narayanan [42] presented a simple GPU implementation. They assigned each atomic task to a single GPU thread. Their approach is limited by the time spent on accessing global memory. Katz and Kider [49] improved Harish's work by using a blocked approach. In their implementation, shared memory is used, which resulted

in a 5x speedup over Harish's work. Lund and Smith [57] applied a multi-stage approach, which enables them to remove data dependencies and make a more efficient use of registers and shared memory. In the end, they achieved a 5x speedup over Katz's implementation.

2.6 Deep Learning Parallelization

Deep learning methods have become the most popular approach in many machine learning domains including speech recognition, image classification, and natural language processing [9]. However, the computation involved in deep learning often comes at a great cost. Therefore, parallelization of deep learning has been studied by many researchers using different architectures. For example, AlexNet [52] parallelized the deep learning training using two GPUs, DistBelief [24] used 16,000 CPU cores for parallelization, and a COTS HPC system [21] consisting of 16 GPU servers has also been studied for deep learning parallelization. Our method differs from the previous methods because, instead of using traditional architectures, we explore the parallelization of deep learning on PIM implemented via 3D die stacking.

Chapter 3

ANDROID MALWARE CLASSIFICATION USING DYNAMIC ANALYSIS

3.1 Introduction

One key behavioral feature used in dynamic analysis of malware is the system call invocations [85, 68, 17, 93, 39, 19, 40, 88, 30]. In previous work on Android malware analysis, the most common representation of the set of system call invocations is to convert them into histograms. However, other representations including signatures, n-grams, and the Markov Chains have been studied previously in Windows malware analysis research [31, 8, 69, 18, 67]. Despite the differences, all methods work reasonably well with high classification accuracies and low false positive rates. However, each of these approaches has its drawbacks. The histogram representation can capture the distribution of system calls, but ignores the structural information. The signature method inherently prevents the detection of unknown malware of which no signatures exist. N-gram analysis is not only unable to capture malware structural information, but also introduces pressure on computational resources due to its large feature space. The Markov Chain representation takes advantage of the transition probabilities between system calls, but it cannot record their order and structure. In this work, we show that using of structure is important. Using structure we are able to achieve up to 87.3% classification accuracy while without using structure we only achieve 83.3%.

The traditional histogram, signature, n-gram, and the Markov Chain representations can all be considered as feature-vector-based representations. In this chapter, we first reimplement the representations used in previous work, namely system call histograms, n-grams, and the Markov Chains for Android malware analysis. Then, we

develop novel graph-based representations, one for each of the three traditional feature-vector-based representation, namely the Histogram System Call Graph (HSCG), the N-gram System Call Graph (NSCG), and the Markov Chain System Call Graph (MCSCG). In the HSCG, processes with direct ancestral lineage to the main process are collected. The main process is the first process created for the application, and thus the first node in the graph. Each process is treated as a vertex and labeled with a histogram of its system call invocations. The graph is formed by connecting parent/children processes. The NSCG is similar to HSCG except the nodes in NSCG are labeled with n-gram vectors. Similarly, nodes in the MCSCG are labeled with the Markov Chain vectors. We also propose Ordered System Call Graph (OSCG) and Unordered System Call Graph (USCG). In OSCG and USCG, each system call invocation is treated as a vertex and labeled with the system call name. The difference between the OSCG and the USCG is how the vertices are connected. We define a process subtree as a tree containing all system call invocations of the corresponding process. Then, each vertex connects to the previous system call invocation in OSCG, whereas all vertices connect to the root of the process subtree in USCG. In both OSCG and USCG, an artificial node labeled *root* is added as the root for the main process subtree, and all the other process subtrees are rooted at the vertex that spawned the corresponding process.

In this research, we first use *strace*, the Linux system call utility to dynamically collect system call invocations from the execution of an Android application. We run each application in an Android emulator called Genymotion[1]. At the beginning of emulation, we run each application for a certain amount of time without interference. We then simulate a series of interactive events, and all the system call invocations that happened during the emulation are recorded. We present a set of methods to convert the system call trace from execution of a specific application into different feature-vector-based and graph-based representations. After conversion, we use graph kernel to compute the pairwise graph similarities of the Android applications for each graph

[1] http://www.genymotion.com

representation. The similarity measures are subsequently constructed and provided as a kernel matrix to a machine learning model, e.g., Support Vector Machine (SVM) for classification. Similarly, vector representations are also fed into an SVM to construct the classification model. Figure 3.1 shows how we construct the dynamic analysis model using graph representations and Figure 3.2 shows the model construction using vector representations.

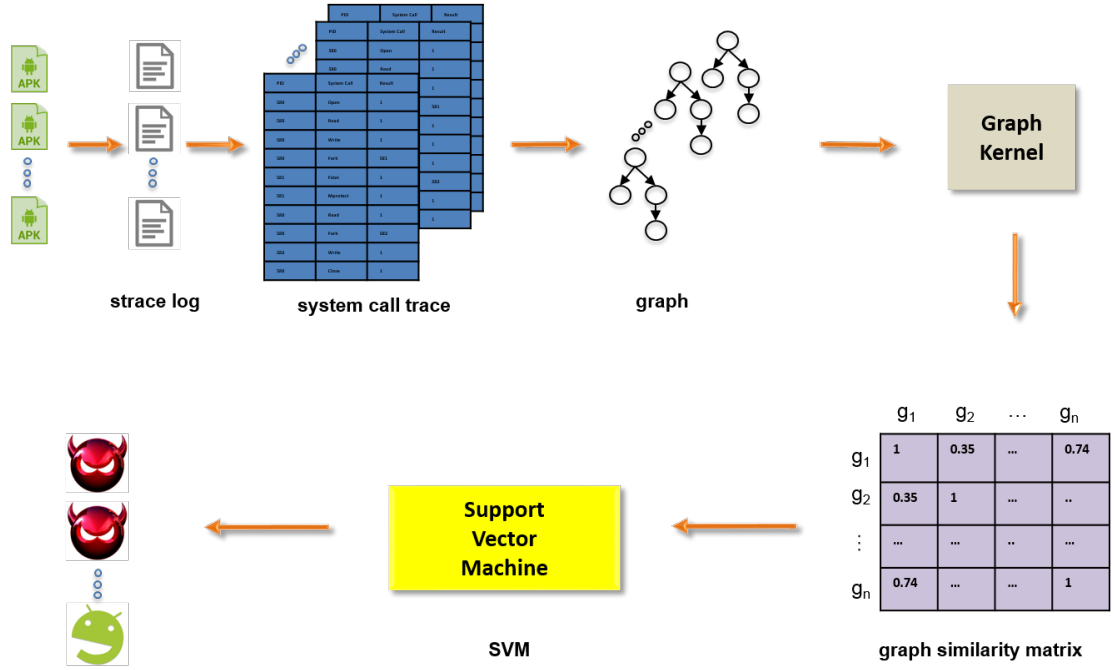


Figure 3.1: Dynamic analysis using graph-based representations. The system call traces are converted to graphs and graph kernels are applied to construct similarity kernel matrix. An SVM is used at the end for classification.

For feature-vector-based representations, Gaussian kernel, Linear kernel, and Intersect kernel algorithms are evaluated. For graph-based representations, we use the Shortest Path Graph Kernel (SPGK) algorithm with the HSCG, NSCG, and MCSCG graphs. SPGK is suitable for similarity computations of graphs with continuous labels (e.g., vectors). For the OSCG and USCG graphs, we use the Fast Subtree Kernel (FSK) method, which is designed for graphs with discrete labels (e.g., strings). Since

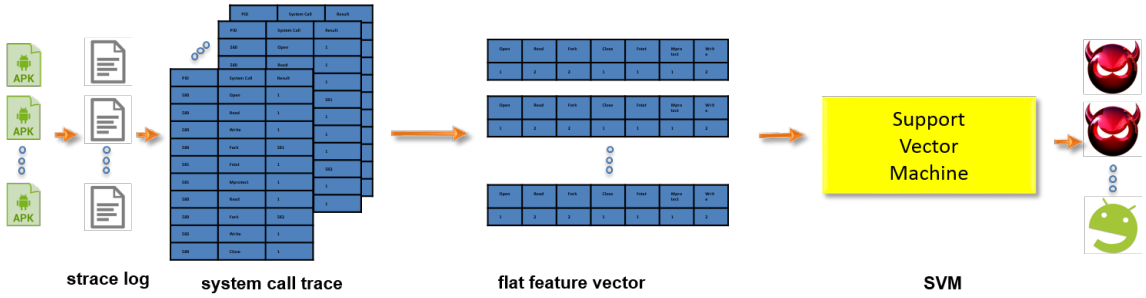


Figure 3.2: Dynamic analysis using vector-based representations. The system call traces are converted to vectors and fed into an SVM for classification.

SPGK is computationally expensive with a running time of $O(n^4)$, we parallelize this algorithm using a hybrid method described in Chapter 5.

3.2 Android Application Emulation

To capture runtime execution behavior of an application, we record the system call invocations during execution of the application using the Linux strace tool [4]. Our dynamic analysis of Android applications is performed in an emulated Android OS environment named Genymotion. It is well known that malware tends to carry out critical tasks upon initial execution. However, some malware may not execute malicious code until after user interaction or until it is triggered by particular events. Therefore, during our emulation, we first start the application and keep it running for some time without any interference. Then we stimulate a series of user interaction events using the Monkey toolkit provided by the Android SDK. We also stimulate other events including phone call, SMS message, and movements while the app is running.

3.2.1 Emulation Procedure

The process of dynamically analyzing Android APK files is performed in a fully automated manner. Before analysis begins, the system is initialized as follows: First, from a Ubuntu 14.04 desktop, the Genymotion emulator is launched. Second, a folder of APK files is created, which serves as the sample repository to analyze. Third, a folder is created as the output repository to store the analysis result files. Given these

initialization steps, a python script is executed to manage the analysis procedure and issues all necessary commands to the runtime environment. Communication between the python script and the runtime environment is performed using the Android Debug Bridge (ADB) tool. Given the initialization described above, the following steps are performed by the python script to load and launch an APK in the runtime environment and collect the execution data:

1. Install application from input repository
2. Retrieve *zygote* process id (PID)
3. Run strace on *zygote*
4. Start an application and run for 20 seconds
5. Retrieve the application PID
6. Simulate user interactions using Monkey and run for 10 seconds
7. Simulate phone call events and run for 10 seconds
8. Simulate SMS message events and run for 10 seconds
9. Simulate phone movement events and run for 10 seconds
10. Simulate a second run of user interactions using Monkey and run for 10 seconds
11. Stop strace, move log files to output repository

Steps 1-11 are iterated until all samples in the repository have been analyzed. Step 1 copies and installs the APK file into the Android emulator. Step 2 applies the *ps* command to retrieve the PID of the *zygote* process. Step 3 starts the *strace* command to record all system call invocations of *zygote* and its descendant processes. The *zygote* process is a standard process running in the Dalvik Virtual Machine that is a component of the Android runtime environment. Whenever an application is launched in Android, the associated process of the application is created and assigned a PID by *zygote*. By recording execution behavior of *zygote*, we are able to record the runtime execution behaviors of all the applications that are going to start later from the moment of their creation by *zygote*. Step 3 assures that we collect all relevant

data of the application from its launching time because we trace its parent process *zygote*. Step 4 launches the application under analysis. The application is executed for 20 seconds without any interference. Step 5 issues a *ps* command which provides a list of all currently running processes and their PID. This list is saved to a file used for identifying the PID of the application under analysis. Since the strace records all processes with direct ancestral lineage to *zygote*, there can be multiple processes that we can ignore during analysis. By recording the list of PIDs we are able to retrieve information only related to the application under analysis and its descendant processes and threads. The package name for the application under analysis is used to identify the running process in the *ps* output. We conduct our analysis by using the PID associated with the application's package name in the *ps* output. Steps 6-10 stimulate different events. In step 6 and 10, we use Monkey to generate 200 random events including touch events, motion events, trackball events, navigation events, system key events, and activity launching events. We also set the delay between events to 100 milliseconds. In step 7, we simulate four phone call events including incoming phone call, answering phone call, making phone call, and rejecting phone call. In step 8, we simulate receiving and sending SMS messages containing sensitive information like password and bank account. In step 9, we stimulate the movement of a phone from one location to another. After each stimulation step, we keep the application running for 10 seconds without any interference. Step 11 stops the strace and places the resulting files in the output repository. Two files are retrieved, one containing the strace data and the other with the *ps* output.

3.2.2 System Call Invocation Extraction

After we collect the strace data and the *ps* output for each application, the package name, retrieved from each application's manifest file, along with the strace and *ps* files, are used as input to a script that converts the strace files to multiple representations. This strace conversion method has two parts. The first extracts system call invocations only belonging to the testing application since the strace log

file contains system call invocations of *zygote* and all its child processes. A system call trace returned by the first part serves as input to the second part, which converts the trace to feature vector or graph representations.

An important step in our strace conversion script is to look up the package name in the *ps* output to identify the PID of the application. With the strace data and PID of the application under analysis, our script can extract only the processes and system call invocations belonging to the testing application. In the strace log file, each line records one system call invoked by a particular PID. The lines can be parsed into columns that record PID, invocation time, system call name, parameters and return values respectively. Since the strace log file contains not only system call invocations of the application under analysis, but all processes with direct ancestral lineage to *zygote*, we need to extract information only related to the testing application. To achieve this, our strace conversion script first creates a process list containing only the PID of the application. Then, it traverses the strace log file. If an invocation is made by a process in the process list, then the PID, name, and return value of this invocation are added as an entry into the system call trace. If the name of this invocation is $fork$ or *clone*, it means a child process is spawned. On success, both system calls return the PID of the child process. On failure, -1 is returned. If the return value of $fork$ or *clone* is not -1, our conversion script adds the return value that is the PID of the spawned child process into the process list, then it continues the search. Finally, a system call trace serving as an input for the subsequent conversion part is returned. Algorithm 1 shows the detailed steps of extracting the related system call invocations. For the purpose of demonstration, we create one synthetic system call trace shown in Figure 3.3.

3.2.3 System Call List

From the full dataset of system call traces, we collect a system call list containing 213 unique system calls referred to as the full system call list. Since the length of the system call list plays a key role in most representations in terms of computation

Algorithm 1 Extract System Call Invocations

Input: strace, ps, package_name
Output: system_call_trace

```
1: root_pid = ps.search(package_name)
2: process_list.add(root_pid)
3: for line in strace.readlines() do
4:     (pid,time,name,parameter,ret_val)=parse(line)
5:     if pid in process_list then
6:         system_call_trace.add(pid,name,ret_val)
7:         if name == "fork" or name == "clone" then
8:             if ret_val != −1 then
9:                 process_list.add(ret_val)
10:            end if
11:        end if
12:    end if
13: end for
14: return system_call_trace
```

PID	Name	Ret_val
580	open	28
580	read	52
580	write	22
580	fork	581
581	fstat	0
581	mprotect	0
580	read	37
580	fork	582
582	write	27
580	close	0

Figure 3.3: This figure shows an example System Call Trace. First column is PID, second column is the instruction name, and the last column is the return value.

time, we want to keep the list as short as possible while maintaining the same classification accuracy. To find out the appropriate list, we compute the average number

of invocations per application for each system call and sort them. The sorting is done separately for benign traces and malicious traces. We extract the top K system calls from both traces and merge them to get the reduced system call list. We then perform experiments by setting K to be 5, 10, 20, and 213. Results show that using the top 20 system calls is not as accurate as using the full system call list but it is able to reach a very close classification accuracy with significantly reduced computation time. Detailed results are in Section 3.5 and the top 20 system calls for benign and malicious applications are shown in Figure 3.4.

3.3 Dynamic Characterization

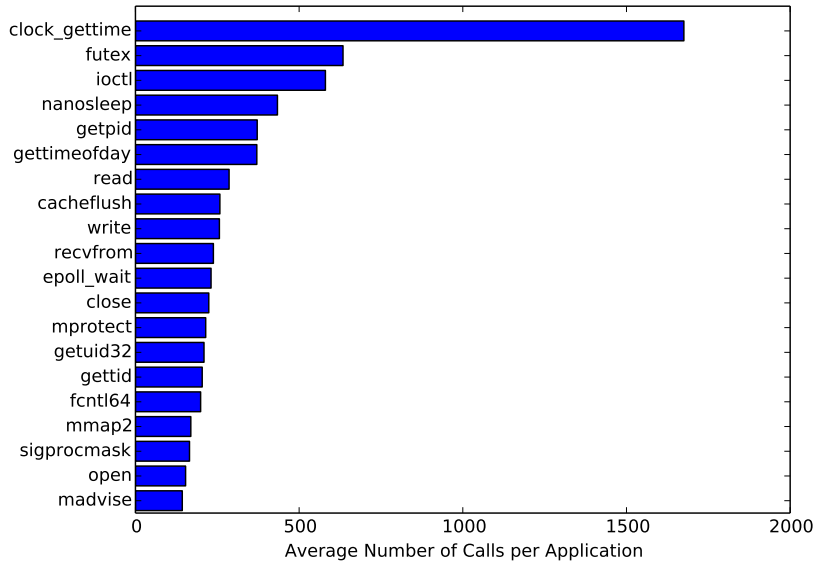
Section 3.2 describes how we collect the system call trace for each application. Along with the system call list, they serve as input to the second part of our script, which converts the trace to a feature vector or a graph representation.

3.3.1 Feature Vector Representations

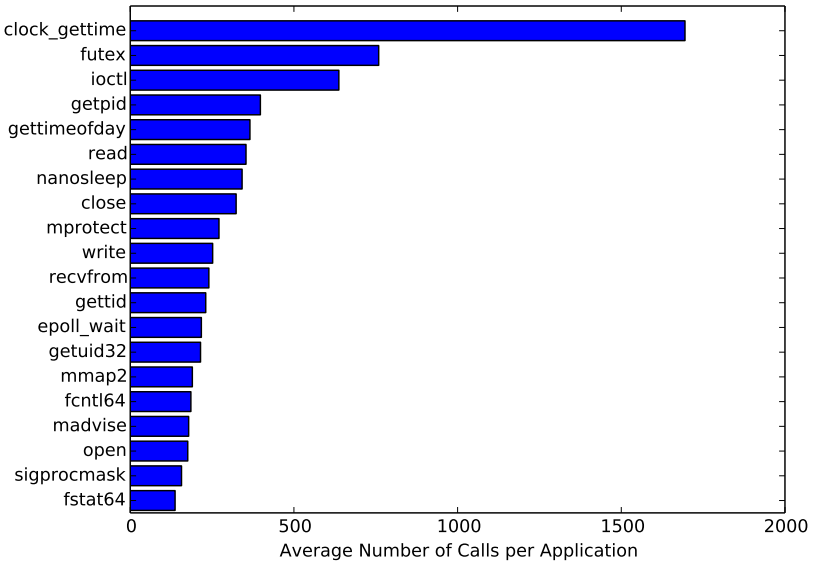
We first convert the system call traces to three previously studied feature-vector-based representations used in Android and Windows malware analysis.

3.3.1.1 System Call Histogram

The first previously studied representation of system call usage in Android malware analysis we look at is a histogram. To convert a system call trace into a histogram, our strace conversion script takes the system call trace and a system call list as input. Then it parses each line of the trace in order, finds the index of each system call name, and increments the corresponding element of the histogram by one. Algorithm 2 shows the algorithm to convert a system call trace to a histogram. For demonstration purposes, we show the resulting histogram in Figure 3.5 converted from the system call trace listed in Figure 3.3. In our experiments, we feed our script both the full system call list and the top 20 system call list, and the resulting histograms are named *histogram-full* and *histogram-top*20, respectively.



(a) Benign application



(b) Malicious application

Figure 3.4: These figures show top 20 system calls per application on average for benign and malicious applications.

Algorithm 2 System Call Histogram Conversion

Input: system_call_trace, system_call_list
Output: histogram

```
1: for line in system_call_trace do
2:     (pid,name,ret_val)=parse(line)
3:     index=system_call_list.index(name)
4:     histogram[index]+=1
5: end for
6: return histogram
```

open	read	fork	close	fstat	mprotect	write
1	2	2	1	1	1	2

Figure 3.5: This figure shows a System Call Histogram converted from the system call trace shown in Figure 3.3.

3.3.1.2 N-gram

Another previously studied representation of system call trace in malware analysis is an n-gram [88]. An n-gram is a contiguous sequence of n system calls from the system call trace. There are two parameters associated with n-gram: n as the number of system call invocations in the sequence, and L as the number of unique system calls which is also the size of the system call list. Given n and L, there can be L^n different n-grams. Therefore, the dimension of the resulting n-gram feature vector grows exponentially as we increase the value of n. In our experiments, we merge the top 20 system calls from benign and malicious applications, which sets the L to be 23 instead of 213 to reduce the n-gram feature vector dimensions. For n, we test three values numbered

2, 3, and 4. Algorithm 3 shows the algorithm of converting a system call trace into an n-gram histogram. Figure 3.6 shows the 2-gram histogram converted from the trace listed in Figure 3.3.

Algorithm 3 N-gram histogram Conversion

Input: system_call_trace, system_call_list, n
Output: N-gram-histogram

```
1: s=len(system_call_trace)
2: l=len(system_call_list)
3: for i = 0 → s − n do
4:     pos=0
5:     for j = 0 → n do
6:         line=system_call_trace[i+j]
7:         (pid,name,ret_val)=parse(line)
8:         index=system_call_list.index(name)
9:         pos+=index*pow(l,j)
10:    end for
11:    N-gram-histogram[pos]+=1
12: end for
13: return N-gram-histogram
```

open	read	1
read	write	1
write	fork	1
fork	fstat	1
fstat	mprotect	1
mprotect	read	1
read	fork	1
fork	write	1
write	close	1

Figure 3.6: This figure shows a 2-gram Histogram converted from the system call trace shown in Figure 3.3.

3.3.1.3 Markov Chain

Another representation of system calls that has been studied in Windows malware analysis research is the Markov Chain [8]. It can be viewed as a directed graph where the vertices are the system calls and the edges are the transition probabilities calculated by the data contained in the trace. For a Markov Chain graph, $G = \langle V, E \rangle$, it consists of two sets, the vertex set V and the edge set E. V corresponds to the system calls, while E, corresponds to the transition probability from one system call to another. Given n system calls in the system call list, an adjacency matrix $A_{n \times n}$ can be used to represent the Markov Chain graph. For each element $A_{i,j}$ in the matrix, it presents the transition probability from system call i to system call j. The adjacency matrix can be treated as a 1D feature vector and fed into a machine learning model for classification. Algorithm 4 shows pseudo code for converting a system call trace into a Markov Chain adjacency matrix. Figure 3.7 shows the resulting Markov Chain converted from Figure 3.3. In our experiments, we generate the Markov Chains using the top 20 system calls and the full system call list, which we have named $MarkovChain\text{-}top20$ and $MarkovChain\text{-}full$, respectively.

Algorithm 4 Markov Chain Conversion

Input: system_call_trace, system_call_list
Output: Adj_mat

```
1: s=len(system_call_trace)
2: for i = 0 → s − 1 do
3:     line1=system_call_trace[i]
4:     line2=system_call_trace[i+1]
5:     (pid1,name1,ret_val1)=parse(line1)
6:     (pid2,name2,ret_val2)=parse(line2)
7:     index1=system_call_list.index(name1)
8:     index2=system_call_list.index(name2)
9:     Adj_mat[index1][index2]+=1
10: end for
11: for i = 0 → len(system_call_list) do
12:     Adj_mat[i]=Adj_mat[i] / sum(Adj_mat[i])
13: end for
14: return Adj_mat
```

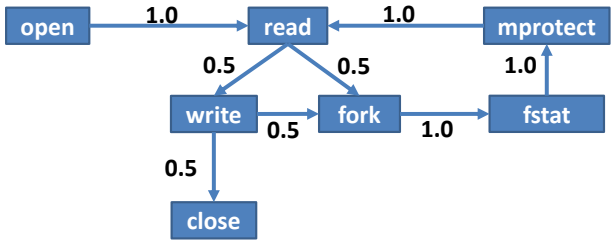


Figure 3.7: This figure shows a Markov Chain converted from the system call trace shown in Figure 3.3.

3.3.2 Graph Representations

To improve the classification accuracy of the feature-vector-based representations, we propose a graph-based representation that augments each traditional feature vector representation. In these graphs, each vertex represents a process of the Android application and each vertex is labeled with a feature vector. We also propose two other graphs, in which each vertex represents a system call invocation and each vertex is labeled with the system call name.

3.3.2.1 Histogram System Call Graph

First, we can construct the Histogram System Call Graph (HSCG) alternative to a simple system call histogram representation. To generate an HSCG, our conversion script reads the trace line by line and parses each one to the triple of (PID, system call name, and return value). If the process corresponding to the PID of this record is not yet a vertex of the graph, it is added as a vertex to the graph. If this is a $fork$ or *clone* system call, the return value, which is the PID of the spawned child process, is also added as a vertex into the graph, and an edge from parent to child is also added. In an

HSCG graph, each vertex is associated with a system call histogram generated from the system calls performed by the corresponding process. The algorithm of converting a system call trace to an HSCG is shown in Algorithm 5. Figure 3.8 shows the resulting Histogram System Call Graph converted from the system call trace in Figure 3.3.

Algorithm 5 Histogram System Call Graph Conversion

Input: system_call_trace, system_call_list
Output: graph

```
1: for line in system_call_trace do
2:     (pid,name,ret_val)=parse(line)
3:     if pid not in graph.vertices() then
4:         graph.add_vertex(pid)
5:     end if
6:     if name == "fork" or name == "clone" then
7:         graph.add_vertex(ret_val)
8:         graph.add_edge(pid,ret_val)
9:     end if
10:    index=system_call_list.index(name)
11:    graph.vertices(pid).feat_vect[index]+=1
12: end for
13: return graph
```

An example of an HSCG graph converted from the *DroidKungFu* malware is shown in Figure 3.9. Figure 3.9(a) shows the full HSCG and Figure 3.9(b) shows the details of the root node (highlighted) of Figure 3.9(a). The vertex contains information including PID, package name (*com.safetest.myapn*), and a list of system calls that have been invoked. All the other vertices contain similar information, but refer to different PIDs and will contain different system call vectors.

3.3.2.2 N-gram System Call Graph

We also create an N-gram System Call Graph (NSCG) as a graph-based alternative to the previously studied n-gram histogram. An NSCG shares the graph structure with an HSCG. The only difference is that the vertices in NSCG are labeled with an n-gram histogram for the corresponding process instead of the system call histogram.

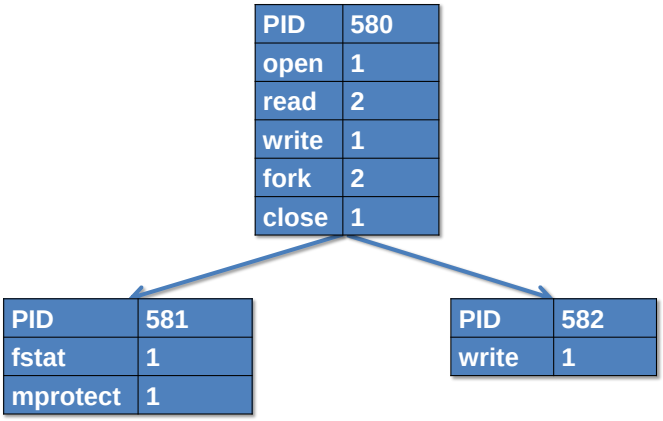
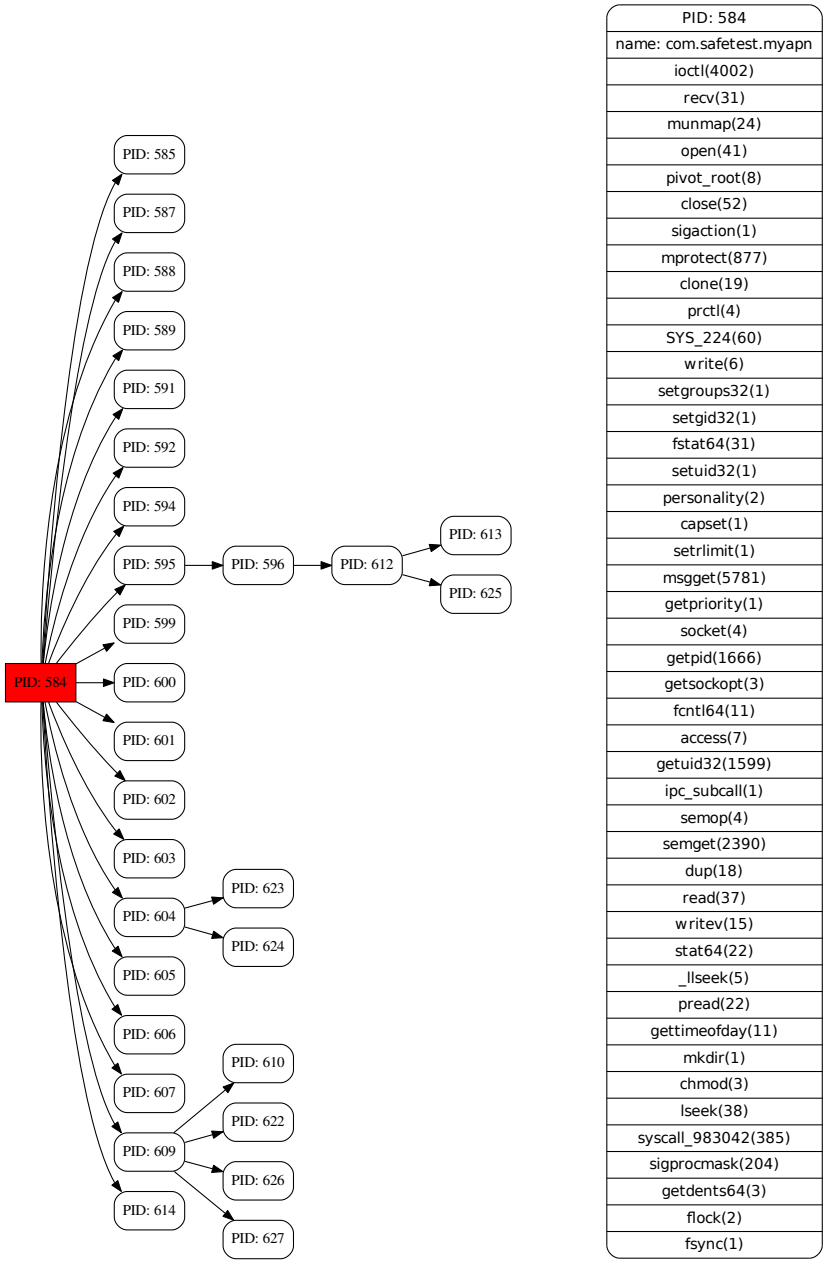


Figure 3.8: This figure shows a Histogram System Call Graph converted from the system call trace shown in Figure 3.3.

In our experiments, we created 2-gram, 3-gram, and 4-gram graphs for the corresponding n-gram histogram using the top 20 system call list. Dimensions of vertex labels in these graphs are therefore 23^2, 23^3, and 23^4 respectively. Figure 3.10 shows the resulting 2-gram System Call Graph converted from the system call trace shown in Figure 3.3.

3.3.2.3 Markov Chain System Call Graph

Similarly, we create the Markov Chain System Call Graph (MCSCG) as an alternative to the traditional Markov Chain representation. MCSCG shares the graph structure with HSCG and NSCG. However, the vertices in MCSCG are labeled with a Markov Chain, instead of a histogram or an n-gram. In our experiments, we create MCSCG for both $MarkovChain\text{-}top20$ and $MarkovChain\text{-}full$. Dimensions of vertex labels in the resulting graphs are 23^2 and 213^2 respectively. Figure 3.11 shows the resulting MCSCG converted the system cal trace shown in Figure 3.3.



(a) HSCG (b) Root Node

Figure 3.9: Visualization of an HSCG graph generated from the *DroidKungFu* malware. (a): the complete HSCG; (b): details of the root node (marked in red) of (a).

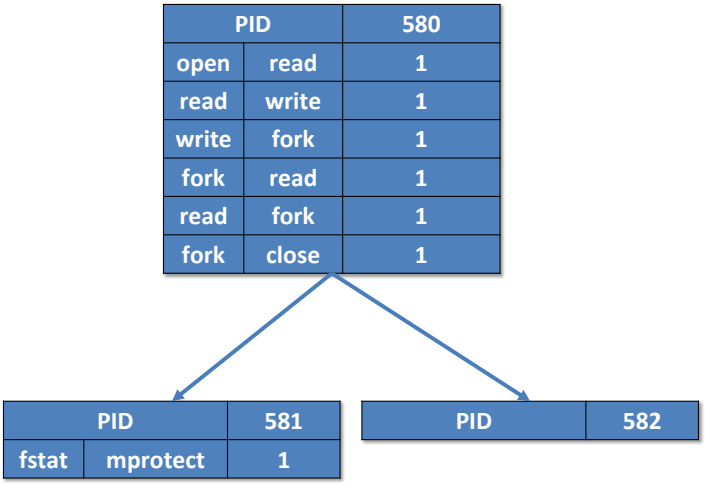


Figure 3.10: This figure shows an N-gram (2-gram) System Call Graph converted from the system call trace shown in Figure 3.3.

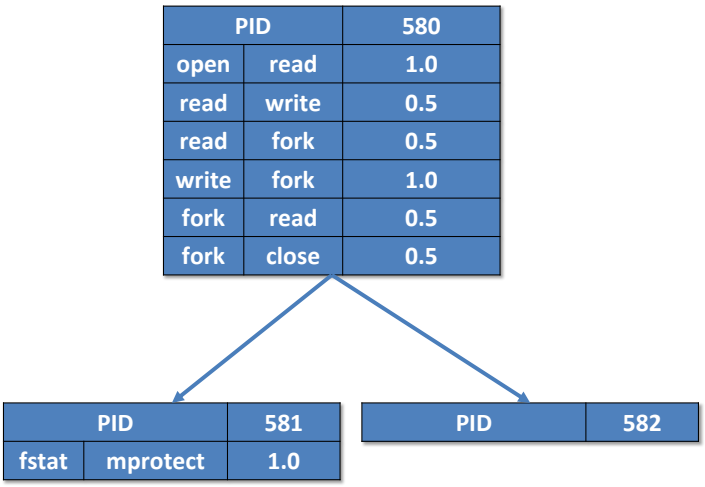


Figure 3.11: This figure shows a Markov Chain System Call Graph converted from the system call trace shown in Figure 3.3.

3.3.2.4 Ordered System Call Graph

Different from the HSCG, the NSCG, and the MCSCG graphs, an Ordered System Call Graph (OSCG) graph treats each system call invocation as a vertex and connects a vertex to the next system call invocation of the same process. To construct an OSCG graph, the strace conversion script first creates a root node for the graph, then reads the system call trace line by line and treats each line as a vertex to add to the graph. Each line is parsed into a PID of the process, the system call name, and return value. The name becomes the discrete label of the vertex. Each vertex is connected to the previous system call invocation of the same process. If the system call is a *fork* or *clone* call, this invocation becomes the root of the subtree which contains all system call invocations of the spawned child process. Algorithm 6 shows the algorithm for converting a system call trace into a OSCG. A dictionary (also known as associative array) named *previous_vertex* is used to track indices of the previous system call invocations for different processes. Figure 3.12 shows the resulting OSCG converted from the system call trace listed in Figure 3.3.

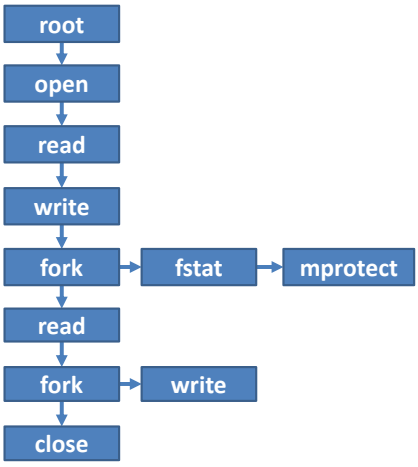


Figure 3.12: This figure shows an Ordered System Call Graph Converted from the system call trace shown in Figure 3.3.

Algorithm 6 Ordered System Call Graph Conversion

Input: system_call_trace
Output: graph

```
1: index = 0
2: graph.add_vertex(0)
3: graph.label(0) = "root"
4: for line in system_call_trace do
5:     index += 1
6:     (pid,name,ret_val)=parse(line)
7:     if index == 1 then
8:         parent = 0
9:     else
10:        parent = previous_vertex(pid)
11:    end if
12:    previous_vertex(pid) = index
13:    graph.add_vertex(index)
14:    graph.label(index) = name
15:    graph.add_edge(parent,index)
16:    if name == "fork" or name == "clone" then
17:        previous_vertex(ret_val) = index
18:    end if
19: end for
20: return graph
```

3.3.2.5 Unordered System Call Graph

The Unordered System Call Graph (USCG) is very similar to OSCG. It also treats each system call invocation as a vertex. In USCG, vertices of one process are connected to a *fork* or *clone* call that spawned the process instead of the previous vertex of the same process. To build a USCG, our strace conversion script first adds to the graph a root vertex representing the PID of the application. It then reads down the system call trace and again parses each line into a PID, name, and return value. Each invocation is added to USCG as a vertex labeled with the system call name and connected to the vertex representing the PID of this invocation. If it is a *fork* or *clone* call, this vertex represents the PID of the spawned child process and therefore becomes the root of the subtree of the child process's invocations. Algorithm 7 shows

the algorithm to convert a system call trace into a USCG. Similar to Algorithm 6, a dictionary named *parent_list* is also used to track the parent vertex for each system call invocation. Figure 3.13 shows the resulting USCG converted from the system call trace listed in Figure 3.3.

Algorithm 7 Unordered System Call Graph Conversion

Input: system_call_trace
Output: graph

```
1: index = 0
2: graph.add_vertex(0)
3: graph.label(0) = "root"
4: for line in system_call_trace do
5:     index += 1
6:     (pid,name,ret_val)=parse(line)
7:     if index == 1 then
8:         parent_list(pid)=0
9:     end if
10:    parent = parent_list(pid)
11:    graph.add_vertex(index)
12:    graph.label(index) = name
13:    graph.add_edge(parent,index)
14:    if name == "fork" or name == "clone" then
15:        parent_list(ret_val) = index
16:    end if
17: end for
18: return graph
```

3.4 Classification

To automatically classify the Android applications into benign or malicious applications, we calculate similarities between feature vectors and similarities between graphs depending on the representation we are using. The similarity measures are constructed as a kernel matrix and fed into the Support Vector Machine (SVM) for classification. We choose the SVM algorithm due to its accuracy as a supervised approach for binary classification. Additionally, SVMs can perform classification based on a precomputed kernel matrix constructed using graph kernels or Multiple Kernel

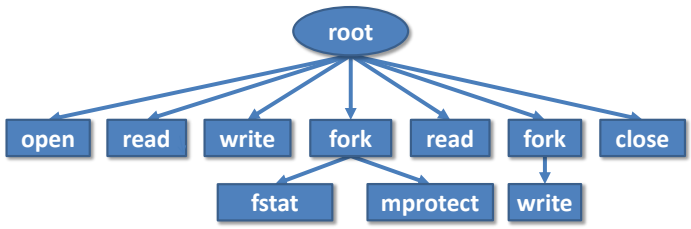


Figure 3.13: This figure shows an Unordered System Call Graph converted from the system call trace shown in Figure 3.3.

Learning (MKL) in our context while most of the other machine learning models cannot.

3.4.1 Kernel Matrix Construction for Vectors

For feature vector representations, we use different kernels to calculate similarities between each pair of vectors. The similarity measures are constructed as kernel matrices and fed into an SVM. For a given dataset $D = \{v_1, v_2, \ldots, v_n\}$ of vectors, a kernel matrix $M_{n \times n}$ is a symmetrical matrix where every element $M(i, j) = k(v_i, v_j)$ refers to the kernel function applied to a pair of vectors v_i and v_j. We evaluated three popular kernels including the Gaussian kernel (Eq. 3.1), the Intersect kernel (Eq. 3.2) and the Linear kernel (Eq. 3.3).

$$k_{gaussian}(x, y) = exp(-\sum_{i=1}^{n} \frac{(x_i - y_i)^2}{\sigma}) \tag{3.1}$$

$$k_{intersect}(x, y) = \sum_{i=1}^{n} min(x_i, y_i) \tag{3.2}$$

$$k_{linear}(x, y) = \sum_{i=1}^{n} x_i * y_i \tag{3.3}$$

3.4.2 Kernel Matrix Construction for the HSCG, the NSCG, and the MCSCG Graphs

For the HSCG, the NSCG, and the MCSCG graphs, we use the Shortest Path Graph Kernel (SPGK) algorithm to compute graph similarities and construct kernel matrices. Details of the SPGK algorithm are described in Chapter 5. Here, we just give a brief introduction to this algorithm. In the SPGK algorithm, first an input graph is converted into an all pair shortest path graph using a Floyd-Washall algorithm. Then, the SPGK algorithm for two shortest path graphs $S_1 = \langle V_1, E_1 \rangle$ and $S_2 = \langle V_2, E_2 \rangle$ is computed as:

$$K_{SPGK}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2) \tag{3.4}$$

where k_{walk} is a kernel for comparing two edge walks. The edge walk kernel k_{walk} is the product of kernels on the vertices and edges along the walk. It can be calculated based on the starting vertex, the ending vertex, and the edge connecting both. We assign e_1 as the edge connecting nodes u_1 and v_1 of graph S_1, and let e_2 be the edge connecting nodes u_2 and v_2 of graph S_2. The edge walk kernel is defined as follows:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \tag{3.5}$$

where k_{node} and k_{edge} are kernel functions for comparing vertices and edges, respectively. The same notation is also applied in the following sections.

In our experiments, we pick the Brownian Bridge kernel (Eq. 3.6) as used in Borgwardt et al. [15] with a c value of 2 for k_{edge}. For k_{node}, we evaluated the same kernels used on constructing kernel matrices for vector sets including the Gaussian kernel (Eq. 3.1), the Intersect kernel (Eq. 3.2) and the Linear kernel (Eq. 3.3).

$$k_{brownian}(e_1, e_2) = max(0, c - |e_1 - e_2|) \tag{3.6}$$

3.4.3 Kernel Matrix Construction for the OSCG and the USCG Graphs

For graphs with discrete labels, e.g. the OSCG and the USCG graphs, the Fast Subtree Kernel (FSK) algorithm [81] is applied to compute graph similarities. The FSK algorithm iteratively constructs the *fingerprints* of a graph based on Weisfeiler-Lehman's procedure for isomorphism testing [81]. The algorithm is shown in Algorithm 8. In each iteration, each vertex and its neighbors are represented by a string consisting of their labels from the previous iteration (line 3). The string is then mapped to a unique value also known as a *fingerprint* (shown in lines 4-7). After H iterations, the graph is associated with a vector of $H|V|$ fingerprints because each vertex is relabeled once in each iteration, and, eventually, every vertex will be associated with H fingerprints. $|V|$ is the total number of nodes in the graph. With the vectors of *fingerprint counters*, a kernel on two graphs can be obtained by calculating the inner product of the two vectors.

Algorithm 8 The Weisfeiler-Lehman Relabeling Process

```
1: for h = 1 → H do
2:     for each vertex v in the graph do
3:         cur_sub ← labelToString(v & its neighbors)
4:         if hashtable.find(cur_sub) then
5:             label(v) ← hashtable.get(cur_sub)
6:         else
7:             label(v) ← f(cur_sub)
8:             hashtable.insert(cur_sub, label(v))
9:         end if
10:    end for
11: end for
```

3.4.4 Support Vector Machine

SVMs consist of two phases: training and testing. Given positive and negative samples in the training phase, an SVM finds a hyperplane which is specified by the normal vector w and perpendicular distance b to the origin that separates the two classes with the largest margin γ [22]. Figure 3.14 shows a schematic depiction of an SVM. During the testing phase, the samples are classified by the SVM prediction

model and assigned either a positive or negative label. The decision function f of the linear SVM is given by

$$f(x) = \langle w, x \rangle + b \tag{3.7}$$

where x is a feature vector representing the sample. It is classified as positive if $f(x) > 0$ and negative otherwise. In the training phase, $\langle w, b \rangle$ are computed as the SVM prediction model from the training data. In the testing phase, the samples are classified using Eq. 3.7 with w and b from the prediction model. To use a kernel matrix as input, the decision function can be transformed to Eq. 3.8. In this equation, y_i is the class label of training data, w^* and α_i are parameters of the prediction model computed from the training data. $K(R_i, R)$ is the kernel value between a testing representation R and a training representation R_i [77]. Once we fill the kernel values with the kernel matrix, we can classify the testing applications.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i y_i K(R_i, R)) \tag{3.8}$$

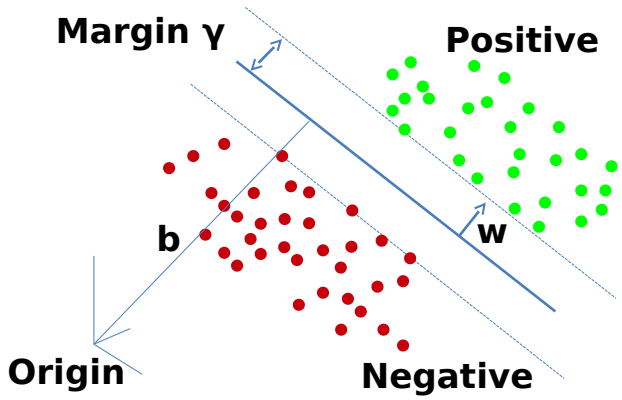


Figure 3.14: This figure shows an illustration of the SVM method. w is the normal vector and b is the perpendicular distance to the origin.

3.5 Experimental Results

In our experiments, we train our SVM on a classification problem with two classes, malicious or benign. For each representation, we construct a kernel matrix. These kernel matrices are then fed into an SVM algorithm using ten-fold cross validation. We also evaluate 15 different values for the regularization parameter C in the SVM, varying from 2^{-2} to 2^{12} with a step value of 2. The experiments are repeated five times with different cross validation partitions and the average classification accuracy rates are reported.

The experiments are performed on a workstation with 128 GB memory, an AMD Opteron 6386 CPU with 32 Piledriver cores clocked at 3.2 GHz, an AMD Radeon HD 7970 GPU with 32 compute units and 3 GB global memory, and a 2 TB hard drive.

3.5.1 Dataset

We collected 5888 applications from Google Play and VirusShare [2]. To reveal malicious and benign applications, we submit our samples to the VirusTotal [3] web service and inspect the output of 51 commercial Anti-Virus (AV) scanners. We label all applications as malicious that are detected by at least two of the scanners. The other applications are labeled as benign. We end up with 1886 malicious applications and 4002 benign applications. The malicious samples were mostly discovered in 2014, and they are categorized into 39 families by a commercial AV scanner named AVG [4].

We recorded the runtime execution behaviors of the Android applications and then converted them to different representations using our strace conversion scripts. Table. 3.1 records the statistics including number of vertices, edges, and shortest paths for the graphs generated from our malicious and benign samples. Since HSCG, NSCG, and MCSCG graphs have exactly the same graph structures, we only show numbers

[2] http://virusshare.com

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

[4] http://free.avg.com/us-en/homepage

for HSCG in the table. Similarly, the OSCG and the USCG graphs have the same statistics. Therefore, we only show numbers for OSCG in the table. Please note that shortest paths are not used in FSK for the OSCG graphs, and thus we do not count them for the OSCG graphs. On the statistics table, the graphs generated from malware are slightly larger than the graphs that came from benign samples on average. We hypothesize that malware tends to spawn additional processes to perform malicious behaviors.

Table 3.1: Detailed statistics of vertices, edges, and shortest paths for graph representations of Malicious (M) and Benign (B) applications. HSCG, NSCG, and MCSCG graphs have the same statistics. OSCG and USCG graphs have the same statistics.

	Vertices			Edges			Shortest Paths		
	Min.	Max.	Avg.	Min.	Max.	Avg.	Min.	Max.	Avg.
HSCG (M)	7	114	29	6	113	28	6	229	42
HSCG (B)	7	109	24	6	108	23	6	411	33
OSCG (M)	150	29401	9005	149	29400	9004			
OSCG (B)	140	32549	8528	139	32548	8527			

3.5.2 Evaluation Metrics

In our experiments, we train our SVM on a classification problem with two classes, malicious or benign. A confusion matrix is used in our method to evaluate the effectiveness of different kernels. From the confusion matrix, we can calculate True Positive Rate (TPR), False Positive Rate (FPR), False Negative Rate (FNR), Accuracy, and Precision.

We let $True\ Positive\ (TP)$ be the number of Android malware that are correctly detected, $True\ Negative\ (TN)$ be the number of benign applications that are correctly classified, $False\ Negative\ (FN)$ be the number of malware that are predicted as benign application, and $False\ Positive\ (FP)$ be the number of benign applications that are classified as malware. Then our evaluation metrics are defined as follows:

$$TPR = \frac{TP}{TP + FN} \tag{3.9}$$

$$FPR = \frac{FP}{FP + TN} \tag{3.10}$$

$$FNR = \frac{FN}{TP + FN} \tag{3.11}$$

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN} \tag{3.12}$$

$$Precision = \frac{TP}{TP + FP} \tag{3.13}$$

3.5.3 Different Kernels

For the SPGK algorithm, we need to pick a valid kernel function k_{node} for comparing vertices and another valid kernel function k_{edge} for comparing edges. In our experiments, we pick a Brownian Bridge kernel for k_{edge} as used in Borgwardt et al. [15] with a c value of 2 for k_{edge}. For k_{node}, we evaluate Gaussian kernels with different σ values from 2^{-6} to 2^{9} with a step value of 2^{3}, an Intersect kernel, and a Linear Kernel. However, we report only the best classification results although they may be achieved by different kernels in different graph sets.

Similarly, we apply different Gaussian kernels, an Intersect kernel, and a Linear Kernel on feature-vector-based representations. Only the best classification accuracies are reported.

For the FSK algorithm, there is no need for picking kernel functions or evaluating parameters, so we only need to run FSK once for the OSCG graphs and once for the USCG graphs. Note that, due to the large depth of the OSCG graphs, we cannot exhaustively compare all the subtrees using FSK. We simply did not have enough memory for that much computation on our machine. Hence, we limit the height of the subtrees in OSCG graphs that we evaluate to be up to 18 in our experiments.

3.5.4 Result from Interaction Stimulation

To understand the importance of interaction stimulation, we first emulate each application for 20 seconds without any interference and then apply various stimulation.

All system call invocations are recorded in one strace log file. We generate HSCG graphs using only the first 20 seconds of the strace log files and name them *HSCG-nointeraction*. We also generate HSCG graphs using the whole strace log files and name them *HSCG-interaction*. By applying graph kernels on these two graph sets, the SVM results show *HSCG-nointeraction* can reach 80.2% classification accuracy while *HSCG-interaction* reaches 85.3%. The 5.1% improvement reveals that by applying different stimulation, we are able to expose more malicious behaviors.

3.5.5 Result from Incomplete Strace

In our experiments, we run strace on *zygote* so we can record all system call invocations of the testing application from the moment it is launched. This is an important step because malware tends to carry out malicious tasks upon initial execution. If we only record the execution behavior after the application has been launched, for example, like the method proposed by Wei et al. [88], important malicious behavior may not be recorded. To understand the importance of complete strace log, we ignore all system call invocations that happen during the first second of the strace log files and generate HSCG graphs named *HSCG-incomplete*. We then compare its performance with *HSCG-interaction*. Experiments show *HSCG-incomplete* reaches 84.5% classification accuracy which is 0.8% less than *HSCG-interaction*. Therefore, recording system call invocations during initial execution can help reveal more malicious behaviors. The experiments in the rest of the dissertation are all based on full strace log files.

3.5.6 Result from Top K System Call List

As mentioned in Section 3.2.3, we extract the top K system calls to reduce the computation time. We generate HSCG graphs from strace log files using the top 5, top 10, and top 20 most frequent system calls. Then, we compare the classification results using these graphs to using the HSCG graphs generated using the full system call list.

Table 3.2 shows the accuracy achieved by using different system call lists and the corresponding graph kernel computation time on our workstation. It shows that using the top 20 system calls cannot reach the same accuracy as using the full system call list. Nevertheless, using the top 20 system calls is able to reach an accuracy level of about 1% better than using top 15 system calls, 2% better than using top 10 system calls, and 3.3% better than using top 5 system calls. In terms of computational cost, using only the most frequent 20 system calls is 1.8x faster than using the full list. Therefore, we use only the top 20 system call list for most of our experiments, unless otherwise noted. For some representations that are not computationally expensive, we experiment with both the full system call list and the top 20 system call list.

Table 3.2: This table shows the best classification accuracy and graph kernel computation time for the HSCG graphs generated using different system call lists.

graph	Accuracy	Time (sec)
HSCG-full	85.3%	68
HSCG-top20	83.3%	38
HSCG-top15	82.3%	30
HSCG-top10	81.3%	23
HSCG-top5	80.0%	17

3.5.7 Results from Feature Vector Representations

Here, we evaluate previously studied feature-vector-based representations including histogram, n-gram, and the Markov Chain. We build system call histograms with top 20 system calls and the full system call list. We name them *histogram-top*20 and *histogram-full*, respectively. For n-grams, we use the top 20 system call list and evaluate different N values, where N is 2, 3, and 4. The resulting vector sets are named 2-*gram-histogram*, 3-*gram-histogram*, and 4-*gram-histogram*. To build our Markov Chains, we use the top 20 system calls and the full list. The resulting feature vectors are named *MarkovChain-top*20 and *MarkovChain-full*, respectively. These feature vectors are fed into different kernels for constructing the kernel matrices. Then, the

matrices are fed into an SVM for five runs of ten-fold cross validation. We report the best classification accuracy that each representation can achieve and the corresponding False Positive Rate (FPR) in Table 3.3. The results show that an n-gram representation performs better when N value is larger. This is reasonable because larger N values means more system call combinations are taken into consideration. Please note that *histogram-top*20 is essentially 1-*gram-histogram* in our experiment. We also observe that using a full system call list can achieve better classification accuracy for histogram and Markov Chain compared to using top 20 system calls for these representations. However, the FPR rates are also increased using the full system call list.

Table 3.3: This table shows the best classification accuracy and False Positive Rate achieved by different feature-vector-based representations.

Vector set	Accuracy	FPR
histogram-top20	74.5%	8.2%
histogram-full	80.5%	9.1%
MarkovChain-top20	81.3%	8.4%
MarkovChain-full	82.6%	9.2%
2-gram-histogram	80.9%	8.1%
3-gram-histogram	82.8%	8.0%
4-gram-histogram	83.3%	8.0%

3.5.8 Result from HSCG, NSCG, and MCSCG Graphs

For each feature vector representation, we generate its corresponding graph representation. HSCG is the graph representation for the system call histogram. In particular, we generate HSCG using the top 20 system call list and the full list. We name them $HSCG$-*top*20 and $HSCG$-$full$, respectively. The N-gram System Call Graph (NSCG) is the graph representation for n-grams. We generate NSCG graphs for 2-*gram-histograms*, 3-*gram-histograms*, and 4-*gram-histograms* using the top 20 system call list. We name these $NSCG$-2, $NSCG$-3, and $NSCG$-4, respectively. For Markov Chain, we build a Markov Chain System Call Graph (MCSCG) and experiment with a $MCSCG$-*top*20 and $MCSCG$-$full$ using the top 20 and the full system call

list. The kernel matrices for these graph representations are fed to an SVM algorithm for five runs of ten-fold cross validation. We also evaluated 15 different values for the regularization parameter C in our SVM algorithm. Figure 3.15 shows the classification accuracy for these different graph representations for different values of C. From the figure, we observe that $HSCG\text{-}top20$ performs the worst and $HSCG\text{-}full$ is slightly better. $NSCG\text{-}2$ is not as effective as $MCSCG\text{-}top20$ because MCSCG has encoded transitional probability information. $MCSCG\text{-}full$ outperforms $MCSCG\text{-}top20$ due to the utilization of a full system call list. Although, $MCSCG\text{-}full$ is inferior to $NSCG\text{-}3$. Overall, $NSCG\text{-}4$ reaches the best accuracy at 87.3%.

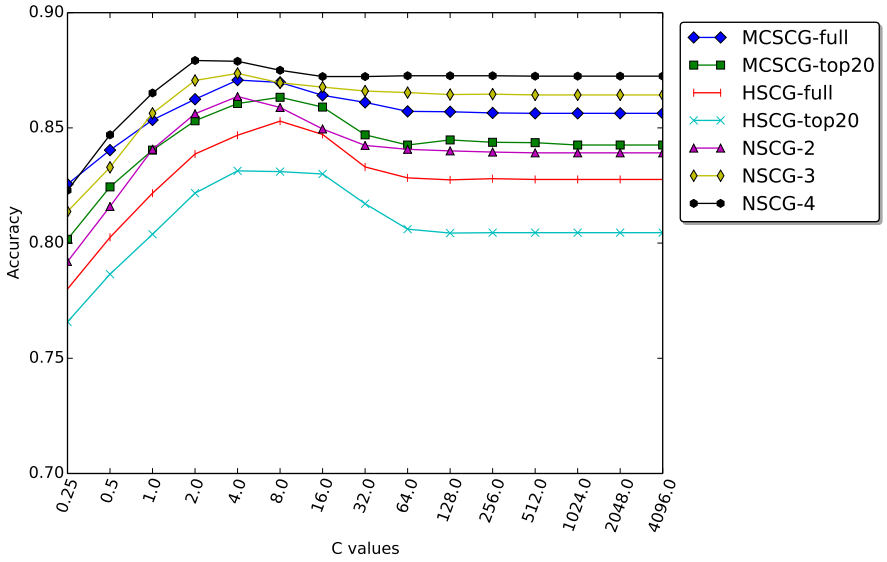


Figure 3.15: This figure shows classification accuracy that achieved using different C values for HSCG, NSCG, and MCSCG graphs

We directly compare the classification accuracy between feature-vector-based representations and their corresponding graph-based representations in Table 3.4. On average, a graph-based representation is able to reach **5.2%** classification accuracy improvement over the corresponding feature-vector-based representation. Thus, we can conclude graph-based representation performs better than flat feature vectors using the same strace information. This shows that the topology of the graph-based techniques

adds predictive power to the model.

Table 3.4: This table shows classification accuracy comparison between feature-vector-based representation and its corresponding graph-based representation.

	Vector	Graph	Improvement
histogram-top20	74.5%	83.3%	8.8%
histogram-full	80.5%	85.3%	4.8%
MarkovChain-top20	81.3%	85.9%	4.6%
MarkovChain-full	82.6%	87.2%	4.6%
2-gram-histogram	80.9%	85.9%	5.0%
3-gram-histogram	82.8%	87.1%	4.3%
4-gram-histogram	83.3%	87.3%	4.0%
Average			5.2%

3.5.9 Result from OSCG and USCG graphs

For the Ordered System Call Graph (OSCG) and the Unordered System Call Graph (USCG) representations, we computed the graph similarities using the Fast Subtree Kernel (FSK) algorithm. Similar to the other representations, the resulting kernel matrices were fed into our SVM algorithm for five runs of ten-fold cross validation with 15 different values for the regularization parameter C. The classification results are shown in Figure 3.16. Experiments show that using USCG graphs we are able to achieve a classification accuracy of **79.1%** while using OSCG graphs we can only reach **53.5%**. This was expected because we cannot compare all subtrees in our OSCG representation due to the subtree height limitation for applying FSK on this representation. We observe that USCG graphs are not as accurate as most of the feature-vector-based representations and other three graph-based representations due to the loss of ordering information.

3.5.10 Graph Kernel Running Time

For n input graphs, each graph kernel returns a kernel matrix of size $n \times n$. The kernel matrix is symmetric, therefore we only compute its diagonal and the top half

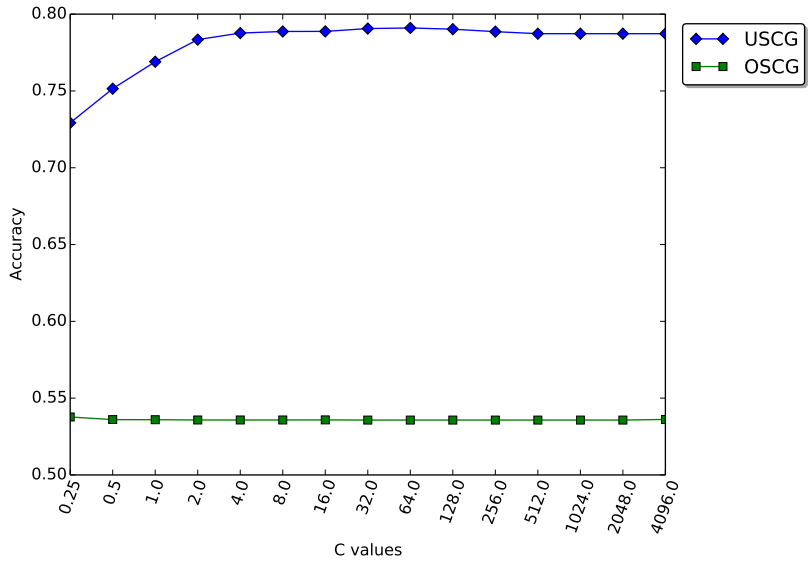


Figure 3.16: This figure shows FSK classification accuracy with different C values for OSCG and USCG graphs.

corresponding to $(n^2 + n)/2$ entries. In our dataset, we have 5888 samples in total. Therefore, we generate 5888 system call traces, one trace for each application execution and then convert these into 5888 graphs for each graph representation and feed the graphs into the corresponding graph kernels. Consequently, each graph kernel needs to compute the similarity between $(5888^2 + 5888)/2$ pairs of graphs given n is 5888.

To speedup computation, we parallelize the Shortest Path Graph Kernel using the hybrid method described in Chapter 5. We also parallelize the kernel methods applied on feature vector sets using OpenMP. Table 3.5 shows the kernel matrix computation time for HSCG, NSCG, MCSCG, and 4-gram-histogram. $SPGK\text{-}Intersect$ is applied on HSCG, NSCG, and MCSCG while $Intersect$ is applied on 4-gram-histogram. It shows that the computation time of $SPGK\text{-}Intersect$ algorithm increases as the vertex label dimension increases. Overall, the time to construct a kernel matrix for the whole training set is reasonable. Moreover, once the training is complete, computing graph similarities against the training graphs for one testing graph is much faster.

Table 3.5: This table shows kernel matrix computation time (seconds) for HSCG, NSCG, MCSCG, and 4-gram-histograms.

HSCG -top20	HSCG -full	MCSCG -top20	MCSCG -full	NSCG -2	NSCG -3	NSCG -4	4-gram -histograms
38	68	177	225	172	276	369	12

3.6 Related Work

To the best of our knowledge, there is no existing work that systematically analyzes different representations of system call invocations for Android malware. There is also no prior work comparing the feature-vector-based representation with the graph-based representation for system calls.

Wei et al., recorded system call invocations for 96 benign applications and 92 malware samples by manually installing and executing each application on an Android phone [88]. They converted the system call invocations into 1-gram, 2-gram, 3-gram, and 4-gram feature vectors. However, their method was not automated and thus cannot be applied to a larger number of Android applications. They only recorded the system call invocations after the application had been started. Information about how the application was launched and executed was ignored by the authors, and as a result their strace log files are incomplete. In our method, we automatically analyze each application. Since we strace the *zygote* process, instead of the testing application, we are able to record the complete set of system call invocations. Dimjasevic et al., also recorded system call invocations for Android malware and converted them into two representations [30]. One was histograms and the other one was a variant of our Markov Chain representation. Canzanese et al. [20], recorded system call traces for Windows binaries and convert them into n-gram vectors. Classification algorithms including logistic regression, naive Bayes, random forests, nearest neighbors, and nearest centroid are tested. The methods proposed in [88], [30], and [20] are reimplemented as the baseline in this chapter.

Canali et al., performed a thorough evaluation of accuracy of system-call-based Windows malware detection [18]. They built different signatures of system calls based on n-grams, n-bags, and n-tuples. Then, a signature matching is performed to detect malware. Our method is different because our method is not signature-based. Anderson et al., compared Markov Chains with n-gram representations based on instruction traces collected from Windows executables [8]. Wagner et al., proposed a graph model based on Linux system call traces which is very similar to our HSCG [86]. However, their random walk graph kernel is expensive and does not scale. In our work, we adapt previous representations and propose novel representations for comparison. We also parallelize the graph kernel to achieve reasonable computation time.

3.7 Conclusion

In this chapter, we evaluate the classification performance of traditional feature-vector-based representations and novel graph-based representations for system call invocations. We first implement previously studied histogram, n-gram, and the Markov Chain representations for system call usage in Android malware analysis. To improve the classification accuracy of the traditional feature-vector-based representations, we propose three graph-based representations where each process is treated as a vertex and labeled with a feature vector. We also explore two other graph representations where each system call invocation is treated as a vertex and labeled with the system call name. Graph kernels are then applied to the graph-based representations to compute graph similarities that are subsequently classified with an SVM algorithm.

To evaluate these representations, we collected a dataset of 4002 benign and 1886 malicious Android applications. We show by feeding system interactions while dynamically analyzing our applications, the classification accuracy can be improved. Subsequent experiments on this dataset showed using graph-based representations are capable of improving the classification accuracies of the corresponding feature-vector-based representations by 5.2% on average.

Chapter 4

ANDROID MALWARE CLASSIFICATION USING HYBRID ANALYSIS

4.1 Introduction

In Chapter 3, we present our dynamic Android malware classification method using graph-based representations. In this chapter, we augment the dynamic analysis with a static analysis method and present **HADM**, **H**ybrid **A**nalysis for **D**etection of **M**alware

For dynamic analysis based on system call invocations, n-gram vectors and the NSCG graphs with different N values including 1, 2, 3, and 4 are selected to construct HADM. We select them because they each achieved good classification accuracy. For static analysis, we extract nine different feature sets including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, and low level instruction sequences. Among these features, the instruction sequences are represented using n-gram feature vectors. Similar to the system call n-gram feature vectors, we evaluate four different N values including 1, 2, 3, and 4 for instruction sequences. All the other static features are converted into histograms. As a result, we generate four feature vector sets for instruction sequences, four feature vector and four graph sets for system call sequences, and one feature vector set for each of the other static features. The feature vector sets are subsequently fed into deep learning methods and combined with the four graph sets using hierarchical Multiple Kernel Learning (MKL) to construct the hybrid classifier.

Deep learning has been widely studied and shown to perform well on machine learning domains including speech recognition, natural language processing, and image classification in the past two decades. In our method, we train one Deep Neural

Network (DNN) constructed by stacking Restricted Boltzmann Machines (RBM) for each of our feature vector sets including system call feature vectors and static feature vectors. The DNN learned features are concatenated to the original features to form the new DNN feature vector sets. Experiments show that higher level features learned from DNN in conjunction with the original features can improve the classification accuracy of each individual feature vector set. Different kernels are then applied on the new DNN feature vector sets to compute similarities of the Android applications. Similarly, different graph kernels are applied on the graph feature sets. The similarity output from each vector kernel or graph kernel can be subsequently constructed as a kernel matrix and fed into an SVM for classification. In HADM, a two-level MKL is applied to combine the discriminative power of different kernel matrices. In the first level, kernel matrices from different kernels are combined as the learning result of the corresponding feature vector set. Similarly, kernel matrices from different graph kernels are combined. In the second level, MKL is applied again to combine all learning results from the first level. The final kernel matrix is then fed into an SVM to construct our hybrid classification model. Figure 4.1 shows the framework of our HADM method.

4.2 Hybrid Characterization

In total, 10 static and dynamic feature sets are extracted from our malicious and benign Android applications including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, instruction sequences, and system call sequences. In total, we generate 16 feature vector sets and 4 graph sets.

4.2.1 Hybrid Analysis Features

Requested permissions: Permission system is the first barrier and one of the most important security mechanisms introduced by Android. Therefore, the requested permission is one of the most used static features in Android application analysis [33]. Prior to installation of an application, it provides users with a list of requested

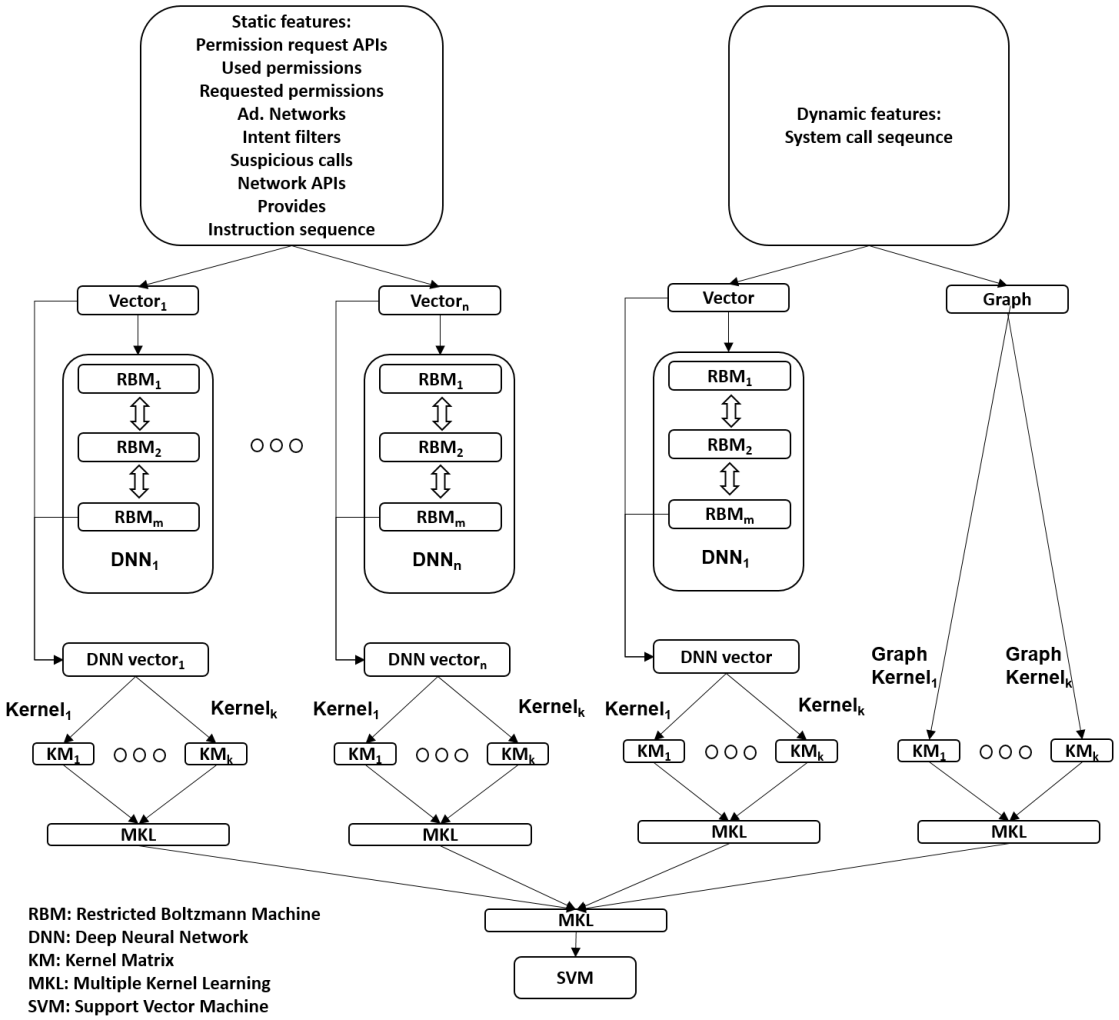


Figure 4.1: This figure shows framework of HADM. Static features are converted to feature vector representations and dynamic features are converted to feature vector and graph representations. Each feature vector set is fed into a DNN for learning. The DNN features are concatenated with the original feature vectors to construct DNN feature vector sets. Multiple kernels and graph kernels are applied to each DNN or graph feature set. The learning results are then combined using a two-level MKL.

permissions (e.g., $SEND\_SMS$, $RECEIVE\_SMS$, $INSTALL\_PACKAGE$). Users normally grant the permissions without knowledge of the capabilities of these permissions, therefore an application can install itself and perform malicious behaviors such as sending premium SMS messages. In our experiments, we collect 1304 requested permissions listed in manifest files of our Android samples.

Permission request APIs: The Android permission can be requested by a series of critical API calls. For example, a *installPackage* API call can request permission $INSTALL\_PACKAGE$ and a $sendDataMessage$ call requests permission $SEND\_SMS$. In total, 246 such API calls are collected from our benign and malicious samples.

Used permissions: Some Android applications request multiple permissions, but only use a subset of the requested permissions. By extracting the used permissions, we can obtain a more precise observation of an application's intention. In total 66 used permissions are collected from our dataset.

Advertising networks: Advertising networks are increasing in numbers in the Android platform to offer developers a variety of monetization models and to help them maximize their revenues. This feature may not be necessarily related to malicious behaviors, but we collect 76 different advertising networks from our samples. The most popular networks are Google Ads, AdMob, and MobClik.

Intent filters: Intent is information about inter-process and intra-process communication. It is a passive data structure holding an abstract description of an action to be performed. Therefore, we can infer it as the intentions of the application. For example, an application can take a picture or can dial a phone number. In total, we collect 1016 different intent filters from our dataset.

Suspicious calls: A subset of API calls is capable of accessing sensitive data, communicating over the network, sending and receiving messages, and executing external commands. These suspicious API calls are frequently used by malware developers. For example, $readSMS$ can read SMS messages, $sendSMS$ can send SMS messages,

getCellLocation is able to get your location, *Runtime→exec* is able to execute external commands, and *System→load* is able to load external libraries. In total, we collect 394 such calls.

Network APIs: We extract the used network APIs because malware tends to access the network and send out sensitive data. For instance, $android.net.wifi.STATE\_CHANGE$ broadcasts an intent action indicating that the state of Wi-Fi connectivity has changed; $android.net.wifi.supplicant.CONNECTION\_CHANGE$ notices both connections to and disconnections from a wifi network. In total, we collect 29 such APIs from our samples.

Providers: The provider declares a component which supplies structured access to data managed by the application. For example, $android.provider.Telephony.SMS\_RECEIVED$ broadcasts that a new text-based SMS message has been received by the device and this intent will be delivered to all registered receivers as a notification. In total, we are able to collect 966 providers from our samples.

Instruction sequences: We utilize an Android tool called Androguard[1] to extract low level instructions (also known as Dalvik bytecode) from an application. For each instruction, we keep only its name, while parameters and output are abandoned. We are able to collect 159 unique instructions from our samples.

System call sequences: For dynamic analysis, system call is the most used feature [33]. We described the details about recording system call invocations in Section 3.2.

4.2.2 Feature Vector Representations

After extracting the features, we embed them into vector space using n-gram representation described in Section 3.3.1.2. In HADM, we first build 1-gram vectors for all features. Then for instruction and system call sequences, we extract the top 20 instructions and system calls, and build 2-gram, 3-gram, and 4-gram vectors for both. In total, we generate four feature vector sets for instruction sequences, four feature

[1] https://code.google.com/p/androguard/

vector sets for system call sequences, and one feature vector set for each of the other features. The 12 static and 4 dynamic feature vector sets are inputs for subsequent deep learning.

4.2.3 Graph Representations

For dynamic system call invocations, we can convert them into a graph-based representation described in Section 3.3.2.2. For simplicity, we refer to the graph representation as an n-gram graph for the rest of this dissertation. To construct the graph, a process tree of the Android application is first extracted from the strace log. Each process is represented as a vertex and connected with its child processes. Then, for each vertex, we collect system call invocations belonging to the corresponding process, and convert them to an n-gram vector. The resulting n-gram vector is attached to the vertex as its label. In total, we generate 1-gram, 2-gram, 3-gram, and 4-gram graphs for the system call sequences.

4.3 Deep Learning Model

Deep learning has shown promise in speech recognition, image classification, and other machine learning domains. It has also been shown that combining advanced features derived by deep learning with the original features provides significant gains. For example, Sarikaya et al. obtained 0.1% to 1.9% absolute classification accuracy improvements on a problem of natural language understanding using combined features [73]. After generating 16 feature vector sets, we train one Deep Neural Network (DNN) for each of the vector sets. Then the DNN learned features are concatenated with the original feature vectors and used for classification.

In our experiment, we select Deep Auto-encoder as our deep learning model. It is a DNN whose output target is the input data itself which serves our purpose of learning new features and combining new features with the original features for classification. The building block of a deep auto-encoder is a probabilistic model called Restricted Boltzmann Machine (RBM). DNN is often initialized or pre-trained using

stacked RBMs. In some literature, DNN is also referred to as Deep Belief Network (DBN) [28].

4.3.1 Restricted Boltzmann Machine

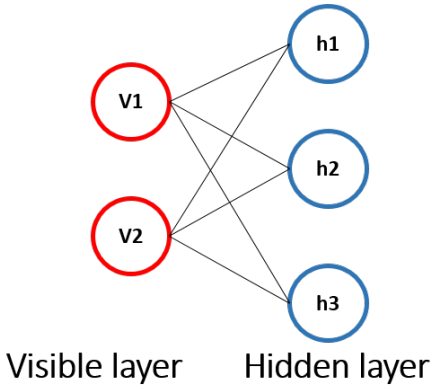
An RBM is an energy-based generative model that consists of two layers: a layer of binary visible units v and a layer of binary hidden units h. The units in different layers are fully connected with no connection between units in the same layer. Figure 4.2(a) shows an RBM with 2 units in the visible layer and 3 units in the hidden layer.

Details of how to train an RBM can be found in [43]. In our experiments, the standard Contrastive Divergence (CD) learning procedure is applied. In the training process, input vectors are first divided into batches and then fed into a training process for a number of iterations until convergence. The training process consists of three steps. The first step is called positive or forward propagation. In this step, probabilities of hidden units are sampled from the input and the positive gradient is computed. The second step is negative or backward propagation where the visible units are reconstructed from the hidden units and then the hidden activities are re-sampled from the reconstructed visible units. The negative gradient is also computed in this step. In the third step, the weight matrix is updated based on the difference of the positive gradient and the negative gradient. Algorithm 9 shows the training process of RBM.

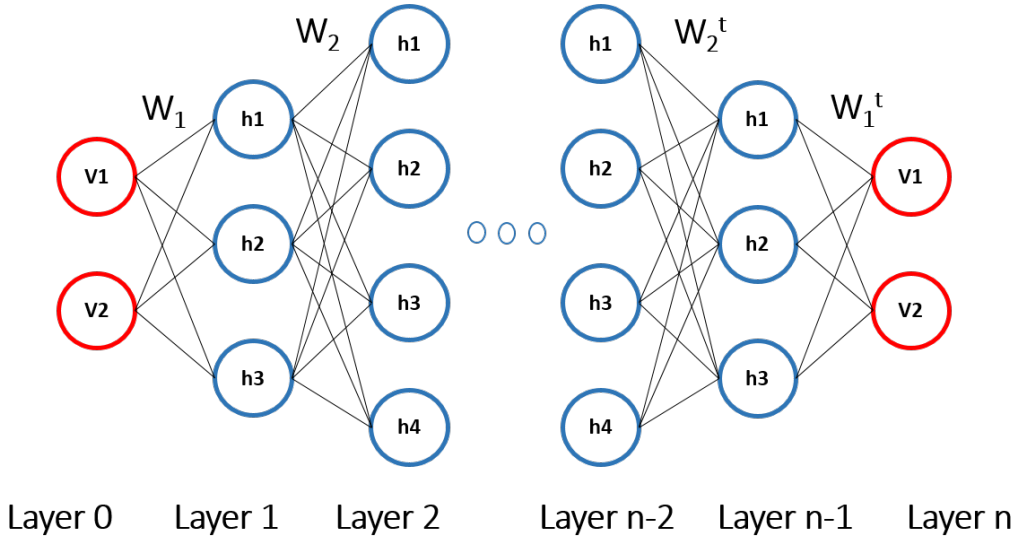
In our experiments, we use the same parameters for training RBMs as used in [51] . The RBMs are initialized with very small random weights and trained for 80 iterations using mini-batches of size 128. The learning rate is set to be 0.001. We also use a momentum of 0.9 to speedup the learning process. All the training in our experiments are performed on an AMD Radeon^{TM} HD 7970 GPU.

4.3.2 Deep Auto-encoder

Deep auto-encoder can be constructed by stacking independently trained RBMs. Each RBM is stacked on top of previous RBM such that the hidden layer of previous



(a) RBM



(b) Deep Auto-encoder

Figure 4.2: An example of RBM and Deep Auto-encoder. (a): a RBM with 2 units in the visible layer and 3 units in the hidden layer. (b): Deep auto-encoder constructed by flipping the stacked RBMs.

Algorithm 9 RBM Training Process

Input: batch set, Weight matrix W, learning rate l
Output: Weight matrix W

for number of iterations **do**
  for number of batches **do**
    From batch V, compute hidden activation $H = V \times W$
    Compute positive gradient $G_p = V \times H$
    From H, sample a reconstruction $V^{'} = H \times W$
    From $V^{'}$, re-sample hidden activation $H^{'} = V^{'} \times W$
    Compute negative gradient $G_n = V^{'} \times H^{'}$
    Update weight matrix W, $W+ = l \times (G_p - G_n)$
  end for
end for

RBM become the visible layer of the current RBM. Each new layer of deep auto-encoder aims to extract higher-level dependencies between the original input vectors, thereby improving the ability of the network to capture the underlying regularities in the data [65]. The first layer of the network is expected to extract low-level features from the input vectors while each new layer is expected to gradually refine previously learned concepts, and therefore produce more abstract concepts [53].

After layer-by-layer pre-training the RBMs, we stack them and then "unroll" the generative model to form a deep auto-encoder. In our experiments, we first train four RBMs and stack them to form a five-layer network. The weight matrices of RBMs are used as the initial weight matrices for the five-layer network. By "unrolling" the stacked RBMs, we flip the five-layer network and create a deep nine-layer network whose lower layers use the matrices to encode the input and whose upper layers use the matrices in reverse order to decode the input. Figure 4.2(b) shows a deep auto-encoder constructed by stacking multiple RBMs and then "unrolling" the stacked RBMs. The auto-encoder can be fine-tuned using back-propagation of error derivatives [27]. Algorithm 10 shows the training process of the deep auto-encoder. To fine-tune the auto-encoder, we use a learning rate of 10^{-6} for all layers and train for 5 iterations. The fine-tune processes in our experiments are also performed on an AMD Radeon$^{\text{TM}}$ HD 7970 GPU. Output from the central layer of the deep auto-encoder is concatenated to the original input to

improve the classification accuracy. We refer to the resulting feature vectors as DNN vectors in the remaining sections of this dissertation.

Algorithm 10 Deep Auto-encoder Training Process

Input: batch set, Weight matrices, learning rate l
Output: Weight matrices

for number of iterations **do**
  for number of batches **do**
    Assign batch V to be A_0, the activity of layer 0
    for layer i from 1 to n **do**
      Compute the activity of layer i,$A_i = A_{i-1} \times W_{i-1}$
    end for
    Compute error for last layer, $E_n = A_n - A_0$
    for layer i from n-1 to 0 **do**
      Back propagate error, $E_i = E_{i+1} \times W_i$
    end for
    for layer i from 0 to n-1 **do**
      Update weight matrix, $W_i += l \times A_i \times E_{i+1}$
    end for
  end for
end for

4.4 Classification

To automatically classify the Android applications into benign or malicious applications, we calculate similarities between feature vectors and similarities between graphs. The similarity measures are constructed as a kernel matrix and fed into a Support Vector Machine (SVM) described in Section 3.4 for classification. To further improve the accuracy, we also apply a hierarchical Multiple Kernel Learning (MKL) method to combine different kernel matrices and build the final hybrid classifier.

4.4.1 Multiple Kernel Learning

One simple way to combine learning results from different features is to concatenate different feature vectors to create a large vector and use it for classification. However, this simple method assigns the same weight to different features which may

lead to suboptimal learning results compared to training on individual features because some features may play more important roles in the learning than other features. Therefore, we need to assign different weights to different features based on their significance during learning. Such optimal weights can be calculated by the MKL algorithm.

MKL is an SVM based method for use with multiple kernels. An SVM takes one kernel matrix as input to build a classifier. However, when it comes to learning, it makes more sense to extract different features from all available sources, learn these features separately and then combine the learning results. MKL does this by taking kernel matrices constructed from different features and different kernels, and is able to find an optimal kernel combination to build the classifier. In addition to the SVM α_i and bias term w^*, MKL learns one more parameter which is the kernel weights β_j in training. Eq. 4.1 shows the resulting kernel method from MKL.

$$f(R) = (w^* + \sum_{i=0}^{N} \alpha_i \sum_{j=0}^{M} \beta_j y_i K_j(R_i, R)) \tag{4.1}$$

4.4.2 Hierarchical MKL

In our experiments, we choose to use Generalized MKL with the Spectral Projected Gradient decent optimization algorithm (SPG-GMKL) [47] to perform MKL. Since we construct multiple kernel matrices for each vector set and each graph set, we first use SPG-GMKL to combine different kernel matrices of the same vector or graph feature set. Experiments show we gain limited classification accuracy improvement in the first level of MKL. Because we explore multiple kernels with different parameters and reported only the best result. Combining different kernels learned from the same features may not necessary improve the performance. However, in the second level, SPG-GMKL is applied again to combine all MKL kernel matrices generated in the first level to learn the final hybrid classifier and much better improvement is achieved.

4.5 Experimental Results

In our method, we extract 10 static and dynamic features and convert them to 16 feature vector sets and 4 graph feature sets. We first evaluate the performance of

each individual feature vector set. Then we train one DNN for each vector set and build the DNN vector set by concatenating DNN learned features with the original features. We show that using the DNN features are able to help improving the classification accuracy. Furthermore, for each of the DNN vector sets, learning occurs by multiple kernels and the learning results are combined using MKL. Similarly, we apply SPGK on the graph sets and construct multiple kernel matrices for each graph set. The kernel matrices of the graph sets are also combined using MKL. Finally, all resulting MKL kernel matrices are combined by applying MKL again to build the final hybrid classifier.

4.5.1 Experimental Setup

The same experimental setup described in Section 3.5 is applied to the feature sets described in this section. The same dataset consisting of 1886 malicious applications and 4002 benign applications are used. All the experiments are run on the same workstation with an AMD Opteron 6386 CPU and an AMD Radeon HD 7970 GPU. To measure the classification results, the same evaluation metrics are used. We also repeat the ten-fold cross validation five times and report the average classification accuracies.

4.5.2 Results from Original Vector and Graph set

First, we evaluate the performance of each original feature vector and graph set. As described in Section 3.5.3, three kernels consisting of Gaussian kernels, Intersect kernels, and Linear kernels are applied to our feature vector sets. Similarly, SPGK-Gaussian, SPGK-Intersect, and SPGK-Linear kernels are applied on the graph feature sets. For Gaussian kernel, we also evaluate different σ values from 2^{-6} to 2^{9} with a step value of 2^{3}. These kernel matrices are fed into an SVM for five runs of ten-fold cross validation.

In Table 4.1, the first column lists the different feature sets. The second column shows the best accuracy achieved by each of the original feature vector sets or graph sets. We observe that the overall best accuracy is achieved by 4-gram vector set

converted from instruction sequences. It reaches 93.5% accuracy. Among the static features other than instruction sequences, requested permissions, the most used static feature in previous static analysis methods [33] performs best and reaches 86.6% accuracy. For the vector and graph sets converted from system call sequences, 4-gram graph performs best with an 87.3% accuracy. As concluded in Chapter 3, the graph set performs better than the corresponding vector set by about 5% on average. This shows that the topology of the graph-based techniques adds predictive power to the model.

4.5.3 Results from DNN

Second, we evaluate the performance of each DNN vector set. In our deep auto-encoder, number of units in the first layer equals to the dimension of input feature vector. We only need to select the layer sizes for the hidden layers of four stacked RBMs. In our experiment, we evaluate all combinations of 4 layer sizes selected from 128, 256, 512, 1024, 2048, and 4096. The DNN learned features in conjunction with original features are then evaluated using the same method as described in Section 4.5.2. In Table 4.1, the third column shows the best accuracy achieved by combining original and DNN vectors. The fourth column lists the corresponding network sizes. In the third column, the data marked in bold shows DNN improves the performance of the original feature vector. We observe that 15 out of 16 feature vector sets can be improved by appending DNN learned features. By using DNN features the maximum accuracy improvement can be achieved is 0.5% by intent filters and 3-gram system call vectors. The best performance is also achieved by 4-gram instruction feature vectors at 93.8%.

4.5.4 Results from first level MKL

In the third experiment, we apply MKL on each DNN vector set and each graph set. Since we evaluate multiple Gaussian kernels with different σ values, we first select a Gaussian kernel matrix with the best performance, then combine it with Intersect

and Linear kernel matrices using SPG-GMKL. The same process is applied to SPGK-Gaussian, SPGK-Intersect, and SPGK-Linear for graph sets.

The fifth column of Table 4.1 shows the results of applying MKL on each vector or graph feature set. Similarly, the data marked in bold means a performance improvement was achieved. In total, 9 out of 20 DNN vector sets and graph sets can be improved by MKL, and the maximum absolute improvement is 0.3%. The sixth column of Table 4.1 shows the weights of Gaussian, Intersect, and Linear kernel matrices learned by MKL. It should be noted that in vector and graph sets converted from system call sequences, the weight of Gaussian kernel is 0.00 because it can actually degrade the accuracy if we include Gaussian kernel in MKL for these sets. We observe that improvement from the first level of MKL is limited because we evaluated multiple kernels with different parameters and reported the best results. After such a large kernel search, we may not be able to improve the performance further even with MKL.

4.5.5 Result from second level MKL

After applying the first level MKL to combine kernel matrices for each vector or graph set, we apply SPG-GMKL again to combine the 20 kernel matrices generated by the first level MKL. The second level MKL weights learned from SPG-GMKL for static and dynamic features are listed in Table 4.2 and Table 4.3, receptively. And, classification results of the final hybrid classifier are shown in Table 4.4. The best classification accuracy we are able to achieve using HADM on our dataset is 94.7% with a FPR of 1.8%. Compared to the best accuracy that can be achieved by the original features, which is 93.5%, we obtain a 1.2% absolute improvement.

4.5.6 Results from concatenating Original Feature Vectors

To compare our HADM method with traditional feature-vector-based methods, we perform experiments by concatenating original feature vectors and feeding them to an SVM for classification. A total of five experiments were performed. In the first experiment, we concatenated all original vector sets. In the second experiment, we

Table 4.1: This table shows classification results of different representations. Acc. means accuracy, Perm. means permissions, Req. means requested, Inst. means instructions, g. means gram, Sys. means system call sequence, and vect. means vector.

	Original Acc.	DNN Acc.	DNN Network Sizes	MKL Acc.	MKL Weights
Static Features					
Perm. APIs	83.9%	**84.3%**	4096-512-4096-512	**84.4%**	[0.32, 5.63, 5.63]
Used Perm.	80.8%	**80.9%**	4096-512-4096-512	**81.0%**	[0.33, 5.62, 5.62]
Req. Perm.	86.6%	**87.1%**	1024-512-256-128	**87.3%**	[0.66, 5.82, 5.59]
Ad. networks	72.8%	**72.9%**	128-128-128-128	72.9%	[0.07, 4.54, 4.54]
Intent filters	84.0%	**84.5%**	1024-1024-1024-1024	84.5%	[3.12, 4.51, 7.68]
Suspicious calls	81.3%	**81.5%**	4096-4096-4096-4096	81.5%	[0.53, 5.88, 5.88]
Network APIs	75.6%	**75.7%**	512-256-512-256	75.7%	[0.01, 8.39, 11.55]
Providers	69.1%	69.1%	4096-512-4096-512	69.1%	[0.00, 7.82, 9.74]
Inst. 1-g.	87.0%	**87.4%**	4096-4096-4096-4096	87.4%	[9.97, 4.66, 1.75]
Inst. 2-g.	90.2%	**90.3%**	512-1024-512-1024	**90.6%**	[9.67, 8.12, 3.79]
Inst. 3-g.	92.3%	**92.5%**	256-512-256-512	92.5%	[8.07, 11.37, 4.86]
Inst. 4-g.	93.5%	**93.8%**	2048-2048-2048-2048	93.8%	[6.90, 11.95, 4.44]
Dynamic System Calls					
Sys. 1-g. vect.	80.5%	**80.8%**	2048-2048-2048-2048	80.8%	[0.00, 5.35, 2.56]
Sys. 2-g. vect.	80.9%	**81.0%**	4096-4096-4096-4096	**81.3%**	[0.00, 6.11, 2.76]
Sys. 3-g. vect.	82.8%	**83.3%**	512-256-512-256	**83.6%**	[0.00, 7.51, 3.20]
Sys. 4-g. vect.	83.3%	**83.7%**	256-512-256-512	**83.9%**	[0.00, 8.20, 3.39]
Sys. 1-g. graph	85.3%			**85.5%**	[0.00, 7.84, 3.95]
Sys. 2-g. graph	85.9%			**86.2%**	[0.00, 8.99, 4.22]
Sys. 3-g. graph	87.1%			87.1%	[0.00, 9.83, 4.41]
Sys. 4-g. graph	87.3%			87.3%	[0.00, 10.43, 4.59]

Table 4.2: This table shows MKL weights of the static features for the final classifier.

Permission APIs	Used Permissions	Requested Permissions	Ad. networks
3.104	2.152	4.888	1.905
Intent filters	**Suspicious calls**	**Network APIs**	**Providers**
1.465	2.678	1.266	1.481
Instruction 1-gram vector	**Instruction 2-gram vector**	**Instruction 3-gram vector**	**Instruction 4-gram vector**
1.137	1.410	2.699	4.392

Table 4.3: This table shows MKL weights of the dynamic features for the final classifier.

System call 1-gram vector	System call 2-gram vector	System call 3-gram vector	System call 4-gram vector
0.578	0.673	1.041	1.357
System call 1-gram graph	**System call 2-gram graph**	**System call 3-gram graph**	**System call 4-gram graph**
1.221	1.498	2.054	2.434

Table 4.4: This table shows classification results from the final classifier.

TP	FN	FP	TN	TPR	FPR	FNR	Accuracy	Precision
1647	239	71	3931	87.3%	1.8%	12.7%	94.7%	95.9%

concatenated all static feature vector sets. In the third experiment, we concatenated all static feature vector sets other than the instruction vector sets and in the fourth experiment we concatenated all instruction vectors. Finally, in the last experiment, we concatenated all system call vector sets.

Results of these experiments are shown in Table 4.5. A check mark means the corresponding vector set is included in concatenation. The table shows by simply concatenating all vector sets, the best accuracy that can be reached is 93.4%, while concatenating only static features achieves 93.6% accuracy. These results are close to using just 4-gram instruction vector set which reaches 93.5% accuracy. These experiments show that adding more features may not necessarily increase the classification accuracy. However, in our HADM method, we refine each feature vector set with DNN and combine different features using weights learned by MKL. Therefore, we are able to improve classification accuracy over individual feature vector sets or graph sets or simply combining them.

Table 4.5: This table shows classification results from simply concatenating the original feature vector sets.

Permission APIs	✓	✓	✓		
Used permissions	✓	✓	✓		
Req. permissions	✓	✓	✓		
Ad. networks	✓	✓	✓		
Intent filters	✓	✓	✓		
Suspicious calls	✓	✓	✓		
Network APIs	✓	✓	✓		
Providers	✓	✓	✓		
Inst. 1-gram	✓	✓		✓	
Inst. 2-gram	✓	✓		✓	
Inst. 3-gram	✓	✓		✓	
Inst. 4-gram	✓	✓		✓	
syscall 1-gram	✓				✓
syscall 2-gram	✓				✓
syscall 3-gram	✓				✓
syscall 4-gram	✓				✓
Accuracy	93.4%	93.6%	92.5%	93.4%	83.8%

4.5.7 Comparison with State-of-the-art

There is one well-known Android malware classification method using hybrid analysis that provides public access: Andrubis[2]. We submitted all of our 5888 samples to Andrubis for analysis. For each application, Andrubis is able to return a maliciousness rating between 0 and 10. In their rating system, 0 means likely benign and 10 means likely malicious. After we get the maliciousness ratings for our samples, we set a threshold t. We evaluate $t = \{1 - 9\}$ where if $t = 1$ means that any application returning 1 or higher is classified as malicious. Table 4.6 shows the classification results obtained using different thresholds. First, we notice that 572 samples failed to be executed by Andrubis. However, these samples are emulated fine in our method. We believe this is because Andrubis can only analyze applications using API level 8 (Android 2.3) or lower [83]. In our method, API level 17 (Android 4.2) is used. Hence, we are able to emulate more recent applications. The third row of Table 4.6 shows the

[2] https://anubis.iseclab.org/

True Positive. The fourth row shows the True Negative. The fifth row shows classification accuracy without failure which is calculated as (TP+TN)/(5888-572). The last row shows overall accuracy which is calculated as (TP+TN)/5888. The best accuracy Andrubis achieved is 85.2% when we ignore failed samples and 76.9% when we count the failures as wrong detections. Consequently, the HADM proposed in this chapter is able to reach a significantly better accuracy on our dataset than the Andrubis method.

Table 4.6: This table shows classification results from Andrubis. Acc. means accuracy and F. means failure.

Threshold	1	2	3	4	5	6	7	8	9
Failure	572	572	572	572	572	572	572	572	572
TP	**1253**	1212	1176	1146	1130	1114	1083	1040	979
TN	2993	3166	3261	3327	3367	3406	3447	3483	**3518**
Acc. w/o F.	80.0%	82.4%	83.5%	84.1%	84.6%	85.0%	**85.2%**	85.1%	84.6%
Overall Acc.	72.1%	74.4%	75.4%	76.0%	76.4%	76.8%	**76.9%**	76.8%	76.4%

4.6 Related Work

A few hybrid methods have been proposed before for Android malware classification including AASandbox [14], DroidRanger [102], SmartDroid [100], Andrubis [56, 89, 55], and Mobile-Sandbox [83]. In contrast to these methods, HADM applies deep learning techniques to improve the performance of each feature vector set, and it combines the results from feature vector sets and graph sets using hierarchical MKL.

Droid-Sec [95] is the first work to apply deep learning to Android malware classification. It extracts over 200 features from both static and dynamic analyses and then feeds these into a DNN for classification. Experiments on 250 malicious and 250 benign applications show Droid-Sec is able to reach 96.5% accuracy. DroidDetector [96] uses the same method as proposed in Droid-Sec. It extracts 192 features from both static and dynamic analyses and characterizes malware using a DNN-based model. DeepSign is another work that applies deep learning [23] on Windows malware signature generation and classification. It uses the Cuckoo sandbox [3] to record the execution

[3] https://cuckoosandbox.org

behavior of each malware. Then, it treats the behavior report as a raw text file and uses uni-grams to convert each report into a 20,000 bit vector. The bit vectors are then fed into DNN to generate signatures. At the end, the signatures are fed into an SVM for classification. Experiments on 1800 malware samples without benign applications show that DeepSign is able to reach 96.4% accuracy. Saxe. et al. [74] also applied deep learning to Windows binary analysis. It extracts four different sets of static features and converts them into 1024-length vectors. The vectors are then fed into a DNN with two hidden layers for classification. In our method, we extract significantly more features from many more samples. More importantly, we train a DNN for each individual feature vector set and combine the DNN learned features with the original features and then perform classification using hierarchical MKL. This method has been shown to improve deep learning results.

MKL has been studied previously in Windows malware analysis by Anderson et. al. [8]. They apply Gaussian kernel and Spectral kernel on the same features and combine them using MKL. We improve upon their method by using hierarchical MKL and multiple features.

4.7 Conclusion

In this Chapter, we propose a hybrid Android malware classification method named HADM. We first evaluate the performance of 16 feature vector sets and 4 graph sets generated from 10 static and dynamic features collected from Android applications. To improve the classification accuracy, we train one DNN for each feature vector set and concatenate the DNN learned features with the original features. Multiple kernels are then applied on the DNN vector sets and multiple graph kernels are applied on the graph sets. The kernel learning results are combined using MKL to further improve accuracy. At the end, MKL is applied again to the combined resulting MKL kernel matrices to build the final hybrid classifier.

Evaluation of different features on our dataset show that the best classification accuracy that can be achieved using static analysis is 93.5% by 4-gram instruction

feature vectors, and the best accuracy that can be achieved using dynamic analysis is 87.3% by 4-gram system call graphs. Furthermore, the application of hierarchical MKL is able to yield a best classification accuracy among all our models by achieving 94.7%.

Chapter 5

PARALLELIZATION OF SHORTEST PATH GRAPH KERNEL

5.1 Introduction

In Chapters 3 and 4, we applied Shortest Path Graph Kernel (SPGK) on graphs to compute pairwise similarities. In this chapter, we focus on accelerating SPGK, as originally proposed by Borgwardt et al. [15]. Research has shown that it is highly competitive in terms of accuracy and running time, when compared with other kernel algorithms [81]. To the best of our knowledge, no other work has addressed the parallelization of shortest path graph kernels. Note that the sequential version of this algorithm runs in $O(n^4)$, which makes it only appropriate for instances from small graphs.

For a given dataset $D = \{g_1, g_2, \ldots, g_n\}$ of graphs, our experiments focus on the calculation of the corresponding kernel matrix $M_{n \times n}$, a symmetric matrix where every element $M(i, j) = SPGK(g_i, g_j)$ refers to the shortest path graph kernel function applied to a pair of input graphs g_i and g_j.

Our proposed method splits the original shortest path kernel into two parts and makes the calculation much faster. We call it the **F**ast **C**omputation of **S**hortest **P**ath Kernel, referred to as **FCSP**. We explored two different parallelization schemes of FCSP on the CPU using OpenMP. One focuses on the parallelization of FCSP on a single pair of graphs while the other focuses on the parallelization of calculating the entire kernel matrix. Next, we split the FCSP into three different GPU kernels using OpenCL, which calculates, the similarities between the vertices of the input graphs, the edges from the two input graphs, and the aggregation of similarities into the final value for $FCSP$ on the GPU. We implement four different GPU parallelization schemes. The

first uses a 1D scheme for domain decomposition; the second uses a 2D scheme for domain decomposition. The third and fourth overlap communication and computation of the first and second methods. We observed that the OpenMP implementations work better when the input graphs are small, while the GPU implementations perform better for larger graphs. This information suggests a hybrid implementation combining CPU parallelization with the best GPU parallelization. The hybrid scheme is based on a graph size threshold. Graphs smaller than the threshold size are assigned the CPU parallelization, and larger graphs are performed using GPU parallelization.

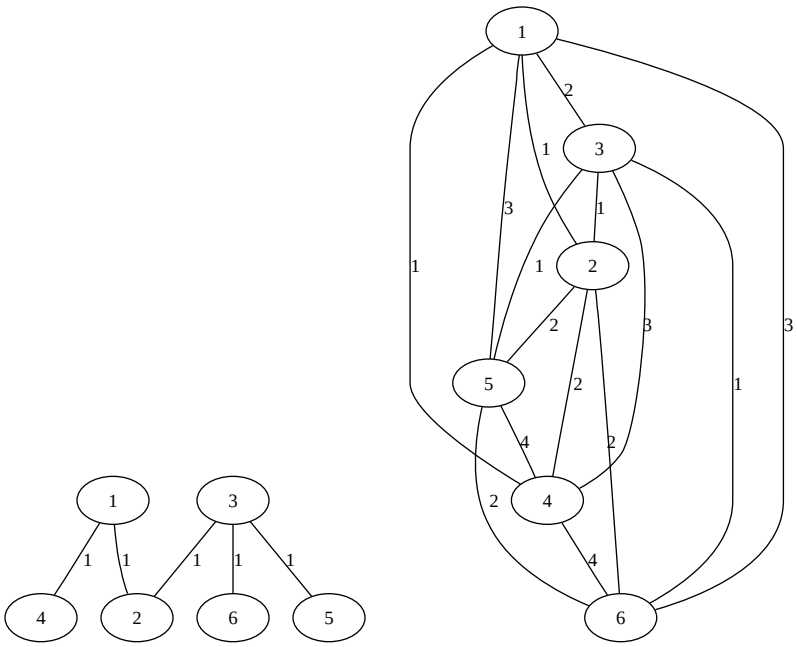
To measure the performance of different implementations, we conduct three separate experiments, in which we apply our different accelerated codes to several datasets. In the first experiment, we create nine synthetic datasets. Graphs in the same dataset have exactly the same number of nodes, and they are all fully connected. We also create one additional dataset containing graphs with various sizes to test the performance of the hybrid implementation. In the second experiment, we measure the speedups in the kernel matrix calculation for different samples of graphs from real-world scientific datasets; specifically four datasets from the bioinformatics domain are used. In the third experiment, we feed the $HSCG\text{-}full$ graphs created in Chapter 3 into different implementations and measure the speedups. As expected, a hybrid implementation achieved the best performance because the graph sizes can vary from less than ten nodes to over one hundred nodes in these datasets, and different graphs prefer either CPU or GPU parallelization implementations depending on their sizes.

5.2 Shortest Path Graph Kernel

The Shortest Path Graph Kernel (SPGK) was proposed by Borgwardt and Kriegel [15]. Basically, this kernel counts the number of shortest paths of the same length having similar start and end vertex labels in two input graphs. One of the motivations for using this kernel is that it avoids the problem of "tottering" found in graph kernels that use random walks [58]. Tottering is the act of visiting the same nodes

multiple times thereby artificially creating high similarities between the input graphs. In shortest path kernels, vertices are not repeated in paths, so tottering is avoided.

In practice, a graph kernel based on shortest paths will require determining all shortest distances in a graph, a problem that is solvable in polynomial time. For example, the Floyd-Warshall algorithm [34] calculates the shortest distances for all pairs of nodes in $O(n^3)$ time, where n denotes the number of vertices. In order to define a kernel that counts shortest paths of similar distances, the original graphs must be transformed into shortest path graphs. This step is a preprocessing requirement before calculating the shortest path graph kernel. Figure 5.1 illustrates the transformation of a labeled graph into a shortest path graph.



(a) Original Graph (b) Shortest Path Graph

Figure 5.1: Illustration of the transformation of a labeled graph into a shortest path graph. Note that the set of vertices is the same in both graphs. Every edge connecting a pair of vertices in the shortest path graph (5.1(b)) is labeled with the length of the shortest path between these pair of vertices in the original graph (5.1(a)).

Given a graph $G = \langle V, E \rangle$, a shortest path graph is a graph $S = \langle V', E' \rangle$, where $V' = V$ and $E' = \{e'_1, \ldots, e'_m\}$ such that $e'_i = (u_i, v_i)$ if the corresponding vertices u_i and v_i are connected by a path in G. Each edge in the shortest path graph is labeled with the shortest distance between the two nodes in the original graph.

SPGK for two shortest path graphs $S_1 = \langle V_1, E_1 \rangle$ and $S_2 = \langle V_2, E_2 \rangle$ is computed as follows:

$$K_{SPGK}(S_1, S_2) = \sum_{e_1 \in E_1} \sum_{e_2 \in E_2} k_{walk}(e_1, e_2) \tag{5.1}$$

where k_{walk} is a kernel for comparing two edge walks. The edge walk kernel k_{walk} is the product of kernels on the vertices and edges along the walk. It can be calculated based on the start vertex, the end vertex, and the edge connecting both. Let e_1 be the edge connecting nodes u_1 and v_1 of graph S_1, and e_2 be the edge connecting nodes u_2 and v_2 of graph S_2. The edge walk kernel is defined as follows:

$$k_{walk}(e_1, e_2) = k_{node}(u_1, u_2) \cdot k_{edge}(e_1, e_2) \cdot k_{node}(v_1, v_2) \tag{5.2}$$

where k_{node} and k_{edge} are kernel functions for comparing vertices and edges, respectively.

Pseudo-code for a naive implementation of the Shortest Path Graph Kernel is presented in Algorithm 11. Given two input graphs $g1$ and $g2$, line 2-7 loops over the shortest path matrices to find all pairs of paths. Line 8 calculates the k_{edge} and line 10-11 calculates k_{node}. Line 12 calculates k_{walk} and computes the summation.

This kernel is attractive because it retains expressivity while avoiding tottering. Moreover, it can be applied to all graphs in which Floyd-Warshall can be performed, and it allows for continuous labels in vertices and edges. The runtime complexity of this kernel is $O(n^4)$, because the Floyd-Warshall transformation can be done in $O(n^3)$ and the kernel calculation requires a pairwise comparison on the number of edges of the shortest path graphs. The latter takes $O(n^2 * n^2)$, because in the worst case scenario the shortest path graph is a complete graph, having n vertices and $\frac{n(n-1)}{2}$ edges.

Algorithm 11 Shortest Path Graph Kernel Algorithm

```
1: K ← 0
2: for i, j = 0 → n_node[g1] do
3:     w1 ← sp_mat[g1][i][j]
4:     if i ≠ j AND w1 ≠ INF then
5:         for m, n = 0 → n_node[g2] do
6:             w2 ← sp_mat[g2][m][n]
7:             if m ≠ n AND w2 ≠ INF then
8:                 k_edge ← EdgeKernel(w1, w2)
9:                 if k_edge > 0 then
10:                    k_node1 ← NodeKernel(g1, g2, i, m)
11:                    k_node2 ← NodeKernel(g1, g2, j, n)
12:                    K+ = k_node1 ∗ k_edge ∗ k_node2
13:                end if
14:            end if
15:        end for
16:    end if
17: end for
18: return K
```

5.3 Fast Computation of the Shortest Path Graph Kernel

An implementation of the Shortest Path Graph Kernel (Algorithm 11) has three issues that slow down its performance. The first is the number of control flow operations. Four for loops and two if statements slow down the program performance whether sequential or parallelized. The second issue is potential redundant computation of k_{node}. Let us consider two graphs as shown in Figure 5.2(a). When we compare walk $D \rightarrow E$ with $A \rightarrow B$, we need to compute k_{node} on (D, A) and (E, B). When we compare walk $D \rightarrow F$ with $A \rightarrow B$, k_{node} on (D, A) and (F, B) have to be calculated. So there is a redundant calculation of k_{node} on (D, A) which wastes computation resources and time. This can be solved by memorizing the k_{node} computation as discussed below. The third drawback is the random memory access pattern in Algorithm 11. Sequential memory access is preferred on the CPU and especially for SIMD architectures like the GPU. The sequential and random read bandwidths on a Nehalem CPU and Fermi GPU have been measured before [45] and the results are shown in Table 5.1. One can clearly observe a 9x difference in bandwidth on the CPU, and 28x difference on the

GPU.

Table 5.1: This table shows sequential and random memory read bandwidth on CPU and GPU.

Platform	Sequential Read	Random Read
Nehalem CPU	8.6 GB/s	0.9 GB/s
Fermi GPU	76.8 GB/s	2.7 GB/s

To address the issues of Algorithm 11, we propose a different way to calculate the shortest path graph kernel. We refer to this as the Fast Computation of Shortest Path Graph Kernel ($FCSP$). In this method, the calculation of the shortest path graph kernel is divided into two main components. First, we calculate all possible instances of k_{node} into a vertex kernel matrix. Second, we calculate all required values for k_{walk}. Note that the kernel functions k_{node} and k_{edge} used to calculate the similarity between a pair of nodes and a pair of edges can be different from application to application. In our experiments, we use the Gaussian kernel 3.1 and the Brownian Bridge kernel 3.6 which are positive semidefinite [78].

For the first component, we call $VertexKernel$, we proceed as follows. Assuming that the order of g_1 is m and the order of g_2 is n, we create a matrix $V_{m \times n}$ for storing the k_{node} values, where every entry is the value of $k_{node}(u, u')$ for u being a node of g_1 and u' being a node of g_2. By using this scheme, the redundant computation of k_{node} is eliminated.

The second component, we call $WalkKernel$, is responsible for calculating k_{walk}, and takes advantage of a new representation of the shortest path adjacency matrix. The new representation is composed of three equally-sized arrays. The length of these arrays is the number of edges in the corresponding matrix. The three arrays store the weight of the edge, the index of the starting vertex, and the index of the ending vertex. This representation is inspired by the formats of storing a sparse matrix on GPUs [11] which can solve the low memory utilization problem for sparse matrices access. By applying this transformation, the two if statements in Algorithm 11 can be removed and four for loops are reduced to two.

The pseudo-code of our new method is presented in Algorithm 12. Given input graphs $g1$ and $g2$, function $Vertex\_Kernel$ calculates all possible instances of k_{node} sequentially and stores them in a matrix V for later access. Function $Walk\_Kernel$ takes advantage of the three 1D arrays converted from shortest path matrix, which creates more sequential memory access and less branch divergence. It calculates all k_{walk} computation and sums them up as the final similarity between two input graphs.

Algorithm 12 Fast Computation of Shortest Path graph kernel

```
1: function VERTEX_KERNEL
2:     for i = 0 → n_node[g1] do
3:         for j = 0 → n_node[g2] do
4:             V[i][j] ← NodeKernel(g1, g2, i, j)
5:         end for
6:     end for
7: end function
8:
9: function WALK_KERNEL
10:    K ← 0
11:    for i = 0 → n_node[g1] do
12:        x1 ← edge_x1_g1[i]
13:        y1 ← edge_y1_g1[i]
14:        w1 ← edge_w1_g1[i]
15:        for j = 0 → n_node[g2] do
16:            x2 ← edge_x2_g2[j]
17:            y2 ← edge_y2_g2[j]
18:            w2 ← edge_w2_g2[j]
19:            k_edge ← EdgeKernel(w1, w2)
20:            if k_edge > 0 then
21:                k_node1 ← V[x1][x2]
22:                k_node2 ← V[y1][y2]
23:                K+ = k_node1 * k_edge * k_node2
24:            end if
25:        end for
26:    end for
27:    return K
28: end function
```

5.4 FCSP running on the Multi-Core CPU

In our experiments we evaluate the calculation of a kernel matrix from a given input dataset of n graphs. A kernel matrix is a symmetric matrix where every entry $M[i, j]$ for $i, j \leq n$ is the corresponding shortest path graph kernel between graphs g_i and g_j. Here, we present two different schemes of $FCSP$ parallelization on multicore CPUs. Both schemes are implemented using OpenMP.

In the first scheme called $OpenMP\_Graph$, we perform the $FCSP$ computation on a single pair of graphs running in parallel. For the first part of $FCSP$, which is $Vertex\_Kernel$, we create a shared 2D matrix for all the OpenMP threads. Then we parallelize the outside loop, which is line 2 in $VertexKernel$ in Algorithm 12, using the dynamic *parallel for* pragma. In $Walk\_Kernel$, we create an array with the size equal to the number of OpenMP threads. This array stores the summed k_{walk} from all OpenMP threads. We use the same dynamic *parallel for* pragma to parallelize line 11 of Algorithm 12 because the dynamic pragma is observed to be faster than the static pragma for this code.

The second scheme is parallelization of the kernel matrix calculation, and is referred to as $OpenMP\_Matrix$. In this scheme, we create the same number of threads as the number of CPU cores available. Each thread resides on one core. To calculate the symmetric kernel matrix for a set of input graphs, the top half triangle of the matrix is transformed to a 1D array. Each OpenMP thread takes one element in the 1D array in order, applies the $FCSP$, fills in the result, and then goes to the next iteration until all elements are computed.

5.5 FCSP running on the GPU

The GPU is a massively parallel co-processor present in many desktops and laptops. The most powerful GPUs can perform more FLOPS and have more memory bandwidth than the most powerful CPUs [61]. Moreover, the development of programming environments, like CUDA and OpenCL, allows programmers to run multiple threads in parallel using the power of the GPU for general-purpose applications.

The CUDA and OpenCL models use an SPMD model where a program known as a kernel represents a single scalar execution entity. In CUDA terminology, which we use throughout for simplicity, this is known as a thread. These CUDA threads are organized for execution into a 1D, 2D or 3D grid of structures known as thread blocks, which perform local computation and are co-located on the same execution unit of the GPU. Thread blocks have access to an on-chip shared memory and can synchronize their constituent threads with each other. Threads are further executed in 16-, 32- or 64- element batches called warps where each thread is a single SIMD lane and a warp is analogous to an x86 thread executing an SSE or AVX instruction. This mapping of neighboring threads to a single SIMD vector can lead to efficiency losses known as thread divergences when threads are mapped to the same warp, but follow different control flow paths.

We present our different GPU parallelizations of the $FCSP$. We first introduce two different domain decomposition techniques for $FCSP$ parallelization. Then we reduce the total running time of these two implementations by overlapping communication and computation. We then propose a hybrid method that combines multicore CPU and GPU parallelization.

5.5.1 Two Domain Decompositions in GPU Parallelization

$FCSP$ is a suitable application for parallelization. In FCSP, branches are removed, no load balancing issue exists between GPU threads, and the coalesced memory access is satisfied. We are therefore able to achieve significant speedups with this approach.

In our GPU implementation, the $FCSP$ is divided into three GPU kernels. The first one is $Vertex\_Kernel$. It calculates all possible instances of k_{node} and stores them in a matrix for later access. The second kernel is $Walk\_Kernel$ which calculates all the required values for k_{walk} and stores them in a matrix or array. The last component is $Reduction\_Kernel$ which sums up all k_{walk} values into a small array which is copied to CPU memory and summed up as the final similarity.

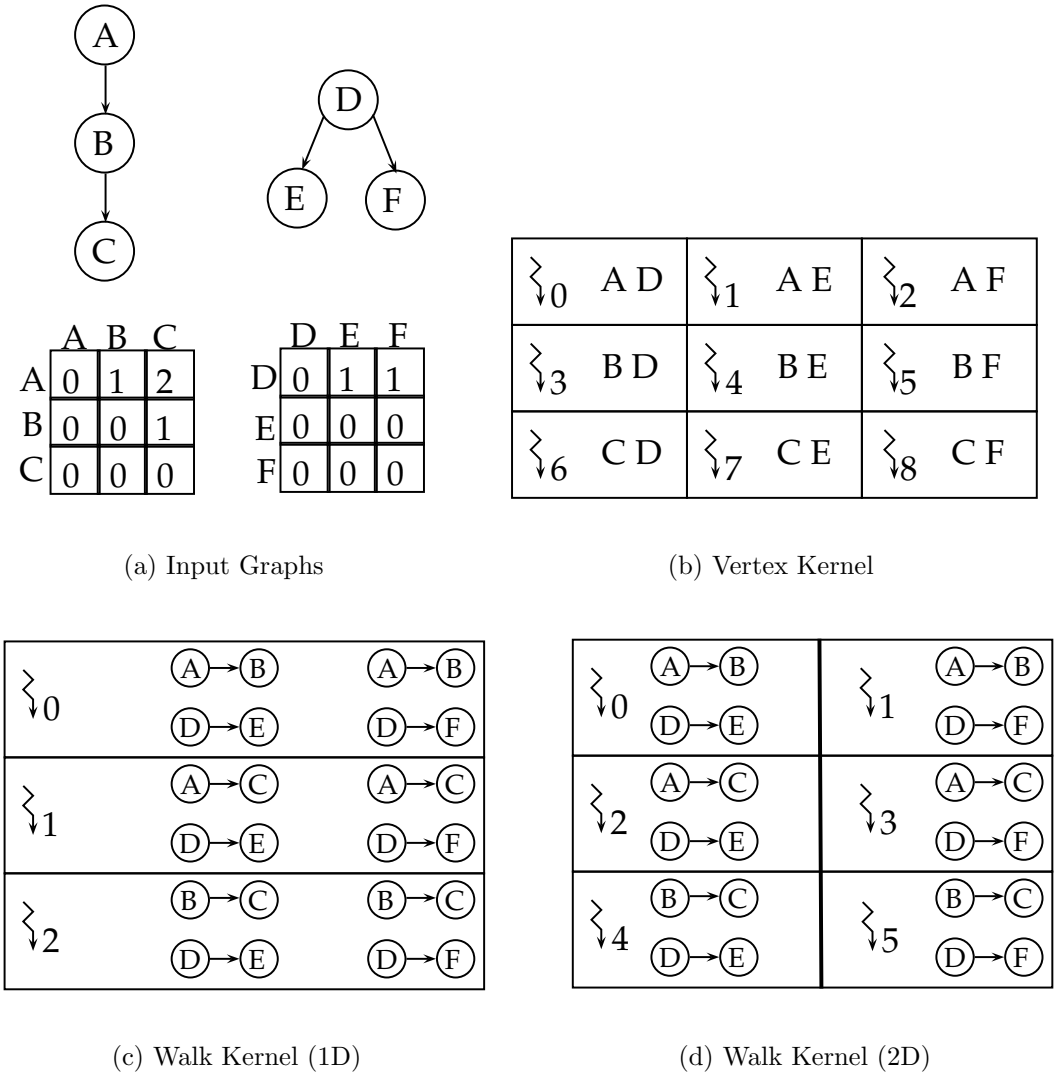
For the first component, named $Vertex\_Kernel$, we proceed as follows. Assuming that g_1 has m vertices and g_2 contains n, we allocate a buffer $V_{m \times n}$ on the GPU memory for storing the k_{node} value. A GPU thread grid is created, where each thread calculates an entry of V. Since we remove the divergence, all threads in this component are running in parallel.

The second component, named $Walk\_Kernel$, is responsible for calculating k_{walk}. Given two input graphs, suppose the number of paths in g_1 is a and g_2 has b paths. We assume g_1 has more paths than g_2 without a loss of generality. For the graph with n nodes, the paths can vary from 0 to n^2. Therefore, the domain decomposition for GPU threads can be challenging. In our implementation, we applied two different methods. The first method is 1D decomposition in which we assign a GPU thread to one path in g_1. This thread will loop through all the b paths in g_2, calculate the corresponding k_{walk} values, and sum them up. An array of a elements will be returned at the end. The second scheme is 2D decomposition. In this method we assign one GPU thread to one pair of paths. So each thread will calculate k_{walk} between two paths. A matrix of $a \times b$ will be returned. The calculation of k_{walk} requires k_{node}, that has already been calculated and cached.

The third GPU kernel is $Reduction\_Kernel$. If we use the 1D domain decomposition scheme in $WalkKernel$, a reduction is performed on a elements. Otherwise, the reduction is performed on $a \times b$ elements. After reduction, a small resulting array is copied back to the CPU. Finally, the similarity between the graphs is calculated by adding up all the values in the array.

The biggest advantage of parallelizing $FCSP$ on the GPU is efficiency. There is no execution divergence between threads because of the shortest path matrix conversion in $Vertex\_Kernel$ and $Walk\_Kernel$. The sequential coalesced memory access is satisfied in all three kernels.

To make the above discussion easy to understand, we show a simple example in Figure 5.2. Figure 5.2(a) shows the two input graphs and the corresponding shortest path adjacency matrices. Figure 5.2(b) demonstrates how $Vertex\_Kernel$ calculates



(a) Input Graphs

(b) Vertex Kernel

(c) Walk Kernel (1D)

(d) Walk Kernel (2D)

Figure 5.2: Example for applying Shortest Path Graph Kernel using *FCSP*. Figure 5.2(a) shows the input graphs and the corresponding shortest path adjacent matrices. Figure 5.2(b) depicts the *Vertex_Kernel* and each GPU thread's assignment. Figure 5.2(c) shows the *Walk_Kernel* with 1D domain decomposition and each GPU thread's calculations. Figure 5.2(d) shows the *Walk_Kernel* with 2D domain decomposition and each GPU thread's calculations.

$k_{node}(u, u')$ for the inputs. Since there are three vertices in each graph, we create a thread grid of size 3×3, as shown in the figure. Each thread in the grid is responsible for calculating one $k_{node}(u, u')$. Results are stored in a matrix and can be cached for later access. Figure 5.2(c) shows the $Walk\_Kernel$ with 1D decomposition. Since there are three edges in the first input graph, we create three GPU threads. Each thread loops through the two edges in the other input graph. The k_{node} values for its vertices are pre-calculated and cached. This allows the threads to finish the calculation of k_{walk} extremely fast. Figure 5.2(d) shows the $Walk\_Kernel$ with 2D decomposition. Since there are three paths in one graph and two paths in the other, six GPU threads are created. Each thread calculates the k_{walk} between one pair of paths.

Before the GPU kernel execution, the two input graphs have to be copied to the GPU memory. So if there are n comparisons, n memory transfers between the CPU and the GPU are needed. This results in a huge overhead. To avoid the unnecessary and duplicated memory transfers, we can simply copy all the graph data into GPU memory. Then at each kernel execution, the GPU thread can fetch needed data according to its targeting graph offset.

5.5.2 Overlapping Communication with Computation

In our GPU implementation, the last kernel is the $Reduction\_Kernel$. A small array is copied to the CPU and summed up for calculating the final similarity. This memory copy from GPU to CPU and computation on CPU may not require much time. However, given n input graphs, the GPU method needs to be called n^2 times. As a result, there may be considerable time spent on memory transfers and CPU computation. Our experiments show that the portion of time spent on the reduction memory transfer can vary from 6% to 50% of the total computation. Fortunately, this part can be hidden by overlapping it with GPU computation. When the reduction kernel completes, we initiate a non-blocking memory transfer. We then assign another pair of graphs to the $Vertex\_Kernel$. As the memory transfer is asynchronous it can be overlapped with the next $Vertex\_Kernel$ execution. When $Vertex\_Kernel$ completes

we initiate a non-blocking execution of $Walk\_Kernel$ and the CPU accumulates the reduction result array to obtain the similarity result while the GPU is executing. Our experiments show this scheme can hide most of the time spent due to memory transfers.

5.5.3 Hybrid Implementation – Combining CPU and GPU

From our experiments, we observed that the implementation with the best performance varied depending on different datasets. When the graphs were small, the CPU implementations beat all the GPU implementations. The GPU with 2D decomposition beat the GPU with 1D decomposition when graphs were small, but it did not improve over the OpenMP implementations. However, when the graphs are large, the GPU with 1D decomposition performs the best. The experiments show that the computation/communication overlapped implementations always performed better than the ones without overlapping. Combining a CPU/GPU implementation seems to be a good idea. We hypothesize that many real world datasets have graphs of a variety of different sizes. So in our $Hyrbrid$ implementation, we first set a threshold $T1$ for average graph size in the input dataset. If the average graph size is smaller than $T1$, then we use $OpenMP\_Matrix$ to calculate the full kernel matrix. Otherwise, we set another threshold $T2$ for graph size to decide graphs that should be run on the CPU vs. the GPU. When the number of shortest paths in both input graphs are smaller than $T2$, we use $OpenMP\_Graph$ to calculate the similarity. Otherwise, the $GPU\_1D\_overlap$ is used.

5.6 Experimental Results

All the experiments were conducted on a GPU cluster where each node has a NVIDIA C2050 GPU. This GPU is based on the GF100 (Fermi) architecture. It contains 14 multiprocessors with 32 processors each, totaling 448 parallel processors. Each multiprocessor contains 32K registers and 64KB, which are split between shared memory and L1 cache. Programmers can allocate 16KB for shared memory and 48KB for L1 cache or 48KB for shared memory and 16KB for L1 cache. In addition to the

GPUs, each node contains two Intel 5530 Quad core Nehalem CPUs clocked at 2.4 GHz with 8MB cache. For our OpenMP implementation, we used 16 CPU threads.

We tested our GPU accelerated versions of the shortest path graph kernel using three datasets. The first dataset is synthetic. The second dataset is a scientific dataset containing labeled graphs using discrete values. The third dataset is the $HSCG\text{-}full$ graphs created in Chapter 3. For performance comparisons, we take memory copies and all the other overhead into consideration, and the total running time is used.

5.6.1 Synthetic Datasets

To test the performance of all our implementations, we created several synthetic datasets. Our OpenMP implementations perform well on datasets with small graphs, while the GPU implementations perform better on datasets with larger graphs.

First, we created nine different datasets. We call these *homogeneous* datasets because each dataset contains graphs of the same sizes, i.e., graphs in the same set have the same number of nodes. All the graphs are fully connected, which means that for each pair of vertices there is an edge connecting them. Since the graphs are fully connected, the number of Shortest Paths (SP) equals to the number of edges. Each dataset has 256 graphs. The 9 datasets contain graphs with 10, 15, 20, 25, 30, 35, 40, 45, and 50 nodes. The largest number of nodes we use is 50 because the average number of nodes in the real scientific datasets we tested had less than 50 nodes. However, we are able to process large graphs with thousands of nodes, as long as they fit into GPU memory. If the graph size goes beyond the GPU's memory capability, we can still cut a graph into multiple chunks and process them chunk by chunk. However, in this dissertation we do not present results on such large graphs.

We first evaluated the naive sequential implementation of the shortest path graph kernel and the $FCSP$ on the CPU. Then, we evaluated our two different OpenMP implementations and four different GPU implementations on the synthetic datasets. Table 5.2 shows statistics for all nine datasets. In order to test the performance of our $Hybrid$ implementation, we created another dataset with mixed sized

graphs. In this mixed dataset, we create 180 graphs with 10 nodes and 76 graphs with 50 nodes. The comparison between graphs of 10 nodes only can be handled by *OpenMP_Graph*, and the comparison between two graphs where at least one of them has 50 nodes can be handled by *GPU_1D_overlap*. We pick the 180:76 ratio because the number of graphs assigned to the CPU would roughly equal the number of graphs assigned to the GPU in this case.

Table 5.2: This table shows statistics about the number of nodes and Shortest Paths (SP) for our synthetic datasets. Because the graphs are fully connected, the number of edges equals to the number of SP.

Dataset	Avg. Nodes	Avg. SP
10-nodes	10	90
15-nodes	15	210
20-nodes	20	380
25-nodes	25	600
30-nodes	30	870
35-nodes	35	1190
40-nodes	40	1560
45-nodes	45	1980
50-nodes	50	2450

Table 5.3 shows the total running time in seconds of the naive SPGK implementation and the FCSP on nine synthetic datasets. Thanks to the branch divergence removal, redundant computation elimination, and sequential memory access, our sequential FCSP algorithm running on a CPU is able to achieve a 76x speedup over the naive sequential SPGK algorithm running on the same CPU.

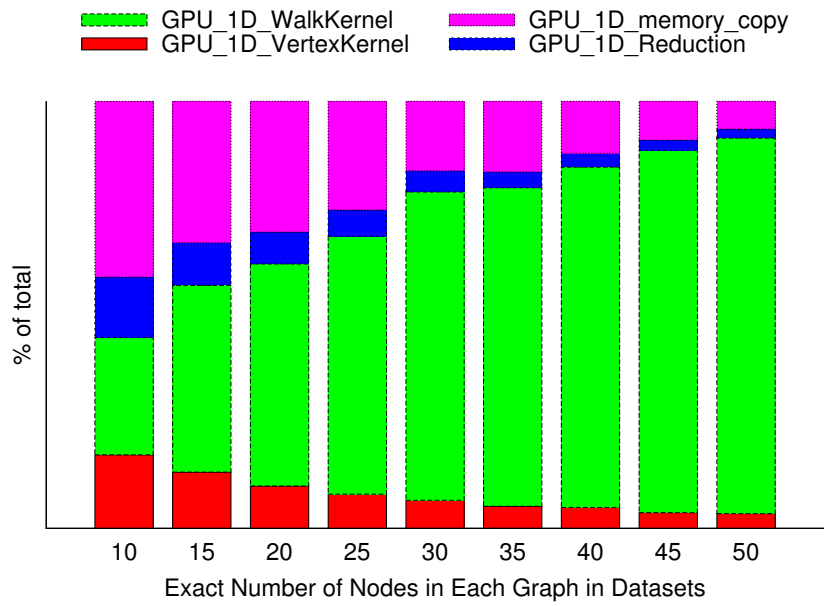
After comparing our FCSP with the naive implementation, we developed and evaluated six different FCSP parallelizations on the CPU and GPU. For *GPU_1D* and *GPU_2D*, we measured the running times spent on the three GPU kernels. The time breakdown is shown in Figure 5.3. As the graph size increases the ratio of $WalkKernel$ gets larger, and the percentages for $VertexKernel$ and memory copy decreases in both *GPU_1D* and *GPU_2D*. The percentage for reduction goes up in *GPU_2D* because the total number of k_{walk} values to be summed increases exponentially (n^2) as the graph size increases. However, in *GPU_1D*, the increase is linear. The percentage of memory

Table 5.3: This table shows speedups of FCSP over a naive SPGK implementation on CPU.

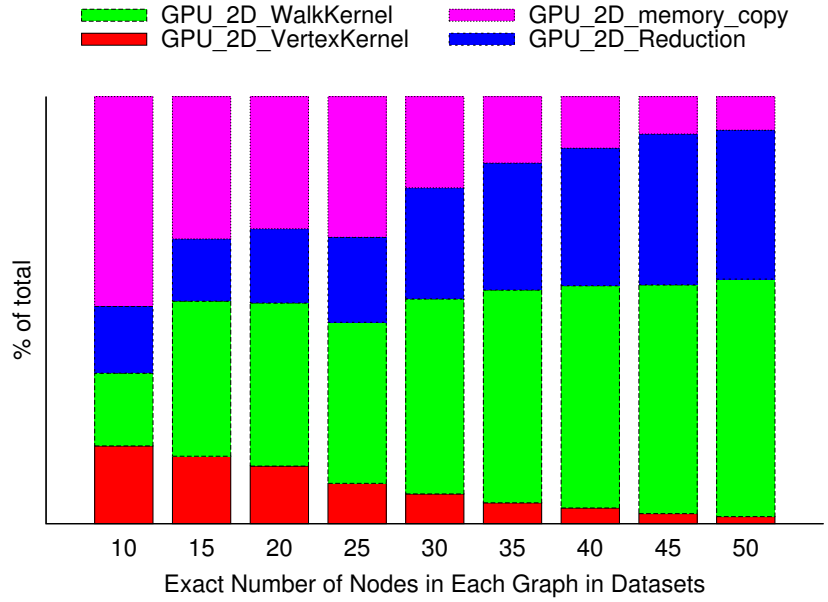
Dataset	SPGK time(sec)	FCSP time(sec)	Speedup
10-nodes	127.99	2.35	54.56
15-nodes	695.24	11.69	59.46
20-nodes	2275.60	37.16	61.25
25-nodes	5668.74	91.42	62.00
30-nodes	11990.24	190.82	62.83
35-nodes	26220.74	355.50	73.76
40-nodes	45850.24	609.67	75.21
45-nodes	74817.26	983.35	76.08
50-nodes	115728.37	1513.69	76.45

copy costs varies from 6% to 50% of the total running time in different datasets. So it is necessary to hide this cost.

The speedup of each parallelization version over sequential FCSP is shown in Figure 5.4. The x-axis shows the exact number of nodes of each graph in the corresponding dataset while the y-axis shows the speedup over sequential FCSP. It is apparent that $OpenMP\_Matrix$'s performance is stable obtaining 8x speedup on average. This is reasonable because there are sixteen OpenMP threads running in parallel in a shared memory system. Even FCSP is optimized, meaning it is still memory bandwidth bound, which prevents it from getting 16x speedup. In our $OpenMP\_Graph$ method, the overhead for OpenMP initialization occurs once for each pair of graphs. So the initialization happens $\frac{n(n-1)}{2}$ times given n input graphs. This is the main reason why $OpenMP\_Graph$ performs worse than $OpenMP\_Matrix$ especially when the graph size is small. For the GPU implementations, the overlapped implementations are faster than the non-overlapped implementations, which proves that hiding memory copy cost by overlapping communication with computation can help reduce total running time. The $GPU\_1D\_overlap$ performs best in four GPU parallelization methods on almost all datasets except the first one. Also, $GPU\_1D\_overlap$ starts to outperform OpenMP implementations when the graph size increases to 35. It reaches a speedup of 18x on the largest dataset.



(a) Time breakdown for GPU_1D



(b) Time breakdown for GPU_2D

Figure 5.3: Time breakdown for the GPU_1D and GPU_2D implementation on the nine datasets. (a) shows the running times in percentages for the $VertexKernel$, $WalkKernel$, $Reduction$, and memory copy for GPU_1D on nine synthetic datasets, and (b) shows the running times in percentages for the GPU_2D.

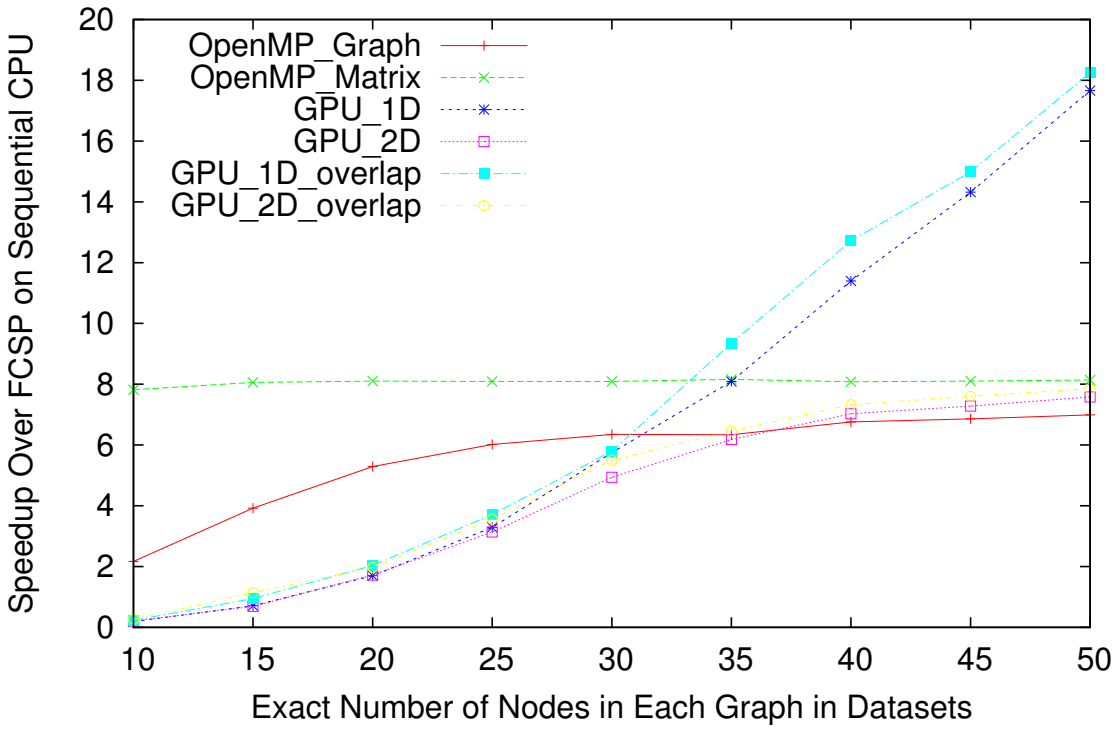


Figure 5.4: This figure shows speedups of CPU and GPU parallelization schemes over sequential FCSP on 9 synthetic homogeneous datasets. For graphs with small number of nodes, *OpenMP_Matrix* performs best. For graphs with large number of nodes, *GPU_1D_overlap* performs best.

To test the performance of our *Hybrid* implementation, one additional dataset was created. In this dataset, only two sizes of graphs were included: five node graphs and fifty node graphs. The average graph size for dataset is larger then the $T1$ threshold in our *Hybrid* implementation, so *OpenMP_Matrix* is not selected. However, there is a clear boundary $T2$ we can set in our *Hybrid* implementation for choosing which algorithm to use. Our *Hybrid* algorithm uses *OpenMP_Graph* when graph similarities of five nodes are calculated. It then switches to the *GPU_1D_overlap* implementation when similarities of large graphs are calculated. Table 5.4 shows the running time in seconds of our different implementations on the mixed dataset. As can be seen, *Hybrid* performs the best which is consistent with our expectation.

Table 5.4: This table shows running time (seconds) on the mixed dataset for different implementation.

OpenMP _Graph	OpenMP _Matrix	GPU _1D	GPU _2D	GPU _1D_overlap	GPU _2D_overlap	Hybrid
24.446	20.349	22.137	32.252	19.751	29.336	**19.042**

5.6.2 Scientific Datasets

We also carried out experiments with real-world scientific datasets from the bioinformatics domain, with graphs that contain discrete labels at the nodes. These datasets were used in prior work to highlight the effectiveness and efficiency of shortest path graph kernels [81]. The datasets are described as follows: (a) MUTAG contains mutagenic aromatic and heteroaromatic nitro compounds [25]; (b) ENZYMES is a dataset of protein tertiary structures of enzymes from the Brenda database [79]; (c) NCI1 and NCI109 are two datasets of chemical compounds screened for activity against non-small cell lung cancer and ovarian cancer cell lines, respectively [87]. Detailed statistics about these datasets are shown in Table 5.5.

In this experiment, we used each of the four scientific datasets as input to our two OpenMP implementations, four GPU implementations, and one hybrid implementation. We show the time breakdown for *GPU_1D* on four scientific datasets in

Table 5.5: This table shows detailed statistics about the number of nodes, edges, and Shortest Paths (SP) for the four scientific datasets.

Dataset	Total Graphs	Avg. Nodes	Avg. Edges	Min. SP	Max. SP	Avg. SP
MUTAG	188	17	39	90	756	324
ENZYMES	600	32	124	2	15500	1215
NCI1	4110	29	64	6	11130	1005
NCI109	4127	29	64	12	11130	995

Figure 5.5. It shows that the time spent on memory copy takes up to 31% in MUTAG, which has the smallest average number of SP, and 17% in ENZYMES, which has the largest average number of SP. We use the performance of *OpenMP_Graph* as a baseline for comparison. The results from all the other implementations are shown in Table 5.6. Clearly, the overlapped GPU implementations outperform the non-overlapped methods in all four datasets. In the first dataset MUTAG, the *OpenMP_Matrix* and *Hybrid* perform the same and beat all the other implementations because the dataset is so small that *Hybrid* selects *OpenMP_Matrix* for kernel matrix computation. MUTAG only has 324 shortest paths on average. The computation power of the GPU cannot be fully utilized on this dataset. As a result, all GPU implementations do not perform well. In the other three datasets, *Hybrid* always performs the best. We also notice that the *Hybrid* implementation achieves its maximum speedup for ENZYMES, which is the dataset with the largest average number of nodes and edges.

Table 5.6: This table shows speedups over *OpenMP_Graph* on four scientific datasets (M. stands for Matrix and o. stands for overlap)

Dataset	*OpenMP_M.*	*GPU_1D*	*GPU_2D*	*GPU_1D_o.*	*GPU_2D_o.*	*Hybrid*
MUTAG	**1.33**	0.28	0.21	0.33	0.38	**1.33**
ENZYMES	1.05	1.73	0.78	1.89	0.94	**1.92**
NCI1	1.13	1.45	0.73	1.66	0.90	**1.78**
NCI109	1.11	1.38	0.69	1.58	0.87	**1.70**

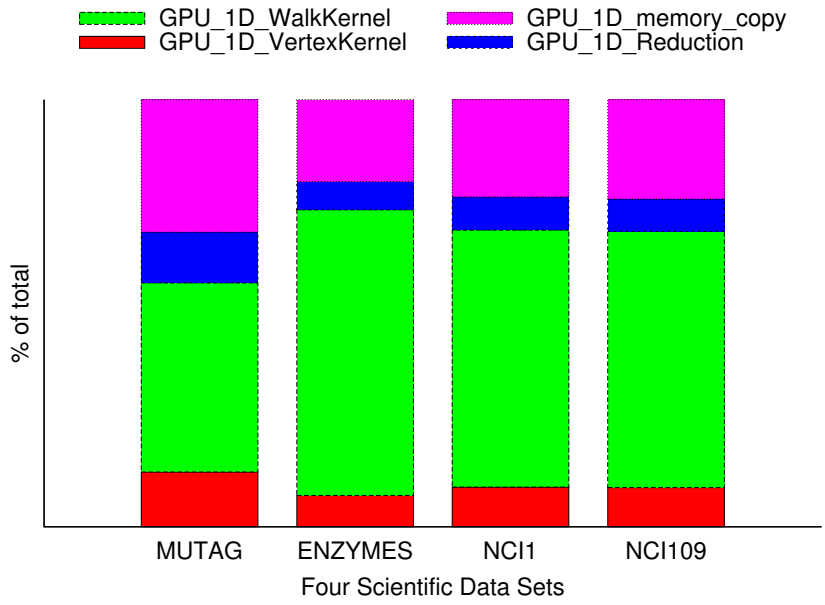


Figure 5.5: This figure shows time breakdown for GPU_1D on four scientific datasets.

5.6.3 Malware Dataset

In the last experiment, we evaluated different parallelization schemes with the malware graphs, e.g., the $HSCG\text{-}full$ graphs, created in Chapter 3. Statistics including number of vertices, edges, and shortest paths for the $HSCG\text{-}full$ graphs generated from our malicious and benign samples are recorded in Table. 3.1. Speedups for different implementations over the $OpenMP\_Graph$ are shown in Table 5.7. Because the $HSCG\text{-}full$ graphs are small, their average number of shortest paths is 42 for malware graphs and 33 for benign graphs, utilization of GPU computation power is low and result in bad performance. However, the $Hybrid$ implementation automatically selected $OpenMP\_Matrix$ for computation and therefore reaches the same best performance.

Table 5.7: This table shows speedups over $OpenMP\_Graph$ on the $HSCG\text{-}full$ graphs created in Chapter 3. (M. stands for Matrix and o. stands for overlap)

Dataset	$OpenMP\_M.$	GPU_1D	GPU_2D	$GPU\_1D\_o.$	$GPU\_2D\_o.$	$Hybrid$
HSCG	**1.36**	0.10	0.09	0.11	0.10	**1.36**

5.7 Conclusion

In this chapter, we targeted fast and efficient parallelization of the shortest path graph kernel on the CPU and GPU. We proposed the Fast Computation of Shortest Path graph kernel. We observed up to 76x speedup using FCSP over a naive implementation of SPGK when we run both sequentially. We parallelized FCSP on the CPU using two OpenMP methods and on the GPU using four OpenCL implementations. We also proposed a hybrid scheme to combine the benefit of CPU and GPU parallelization. Our experiments showed the best implementation of FCSP depends on the size of the graphs being processed. For small graphs, the OpenMP implementations on the CPU performs best while for large graphs the GPU implementation performs best. Therefore, a hybrid algorithm that selects the optimal algorithm per graph size works best in all datasets.

Chapter 6

PARALLELIZATION OF DEEP LEARNING

6.1 Introduction

Deep learning has shown promise in domains such as speech recognition, image classification, and natural language processing [9]. State-of-the-art deep learning models often contain a large number of neurons, resulting in millions or even billions of free parameters [21]. To train such complex models, tremendous computational resources, energy, and time are required. Recently, a significant amount of effort has been put into speeding up deep learning by taking advantage of high-performance systems. Despite that, AlexNet [52] takes more than five days to train on two GPUs; DistBelief [24] uses 16,000 cores to train a neural network in a few days; COTS HPC system [21] scales to neural networks with over 11 billion parameters using a cluster of 16 GPU servers. While these prior works have successfully accelerated computation by mapping deep learning to existing architectures, relatively little research has been done to evaluate the potential of emerging architecture designs, such as in-memory computing, to improve the performance of deep learning.

In Chapter 4, we built DNN vector sets by performing training on GPU. In this chapter, we explore the potential of Processing In Memory (PIM) implemented via 3D die stacking to improve the performance of deep learning. While PIM research has been active for a few decades, it has not been commercially viable due to manufacturing and economic challenges. However, recent advances in 3D die stacking technology make it possible to stack a logic die with one or more memory dies which enables a new class of PIM solutions. These solutions build on the same underlying 3D stacking technology used by recent memory technologies such as Hybrid Memory Cube (HMC) [62] and

High Bandwidth Memory (HBM) [2]. It has been demonstrated that a broad range of applications can achieve competitive performance on viable 3D-stacked PIM configurations compared with a representative mainstream GPU [97]. We evaluate the performance of scaling deep learning models on a system with multiple PIM devices. In this system, the host is a high-performance mainstream APU. This host is attached to several memory modules, each with PIM capabilities consisting of a small APU.

From the two most popular deep learning models, Convolutional Neural Network (CNN) and Deep Belief Network (DBN), we select three frequently used and representative layers: the convolutional layer, the pooling layer, and the fully connected layer. Across the multiple PIM devices, we parallelize these layers individually. Two parallelization schemes are evaluated, which are data parallelism and model parallelism. The data parallelism approach keeps a copy of the full neural network model on each device but partitions the input data into mini batches across them. We evaluate data parallelism on all three layers. The model parallelism approach partitions the neural network model and distributes one model partition to each device. We apply model parallelism to the fully connected layer, as the number of parameters in this layer increases drastically as the network grows. Memory capacity can often be a limiting factor for fully connected layers. When the model is too large to fit into a PIM's memory, it is essential to partition the model across multiple PIMs using model parallelism.

Experiments show that by scaling deep learning models to multiple PIMs available in a system, we are able to achieve better or competitive performance compared with a high-performance host GPU in many cases across the different layers studied. We show that model parallelism consumes much less memory than data parallelism on fully connected layers, and it also reaches better performance when the number of input images per batch is small. However, as the batch size increases, data parallelism scales better due to the absence of synchronization and it outperforms model parallelism.

6.2 Deep Learning Models

Deep Belief Networks (DBN) are constructed by chaining a set of Restricted Boltzmann Machines (RBM) [44]. We explain the details of RBM in Section 6.2.3. Here we show an example of a DBN trained for speech recognition in Figure 6.1. The model takes as an input a spectral representation of a sound wave. The input is then processed by several RBMs where each RBM may contain a different number of hidden units. Finally, the DBN translates the input sound wave to text output.

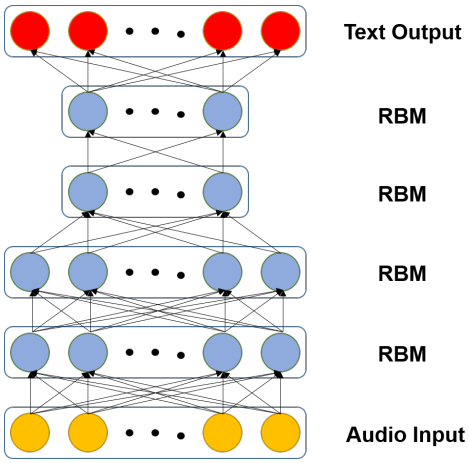


Figure 6.1: A simple DBN used for speech recognition. The input audio is processed by several RBMs and then translated to text.

Unlike DBN, CNN may consist of multiple different layers. The most basic ones are convolutional, pooling, and fully connected layers. The fully connected layer has effectively the same characteristics as the RBM. The details of each layer are discussed in the following subsections. Here we show an example of a CNN model trained for digit recognition in Figure 6.2. The input to this model is an image containing one hand-written digit. The input is first processed by the convolutional layer where each filter outputs one feature map. The feature maps are downsampled by the max pooling layer. Outputs from the pooling layer are then processed by the fully connected layer. The final output layer contains 10 neurons where each neuron represents one digit. The neuron with the highest probability is the prediction result of the input.

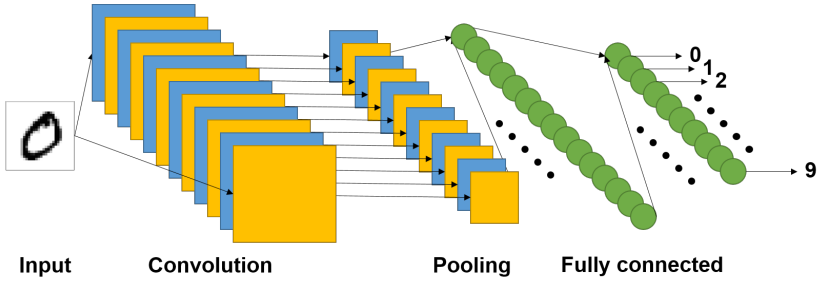


Figure 6.2: A simple CNN used for digit recognition. Input is an image of a handwritten digit. After processing by the convolutional layer, the pooling layer, and the fully connected layer, the CNN outputs a neuron with the highest probability as the prediction result.

Traditionally, deep learning applications consist of two phases: training and prediction. The training phase contains forward propagation and backward propagation for weight updates [13, 75]. In forward propagation, input images are processed through all layers in the deep learning model with initial weights. In backward propagation, error is computed based on the model output. The error is then propagated back through all layers and is then used to update the weights for each layer. The prediction phase contains only the forward propagation using the weights learned in the training phase. This study focuses on the three common layers in the forward propagation: convolutional, pooling, and fully connected, as the forward propagation is key to both the training and prediction phases.

6.2.1 Convolutional Layer

The Convolutional layer is the core building block of CNN. The input of this layer is a batch of images, and each image has 3 dimensions including width, height, and depth (or channels). The convolutional layer applies one or several convolutional filters (or kernels) to each 3D input volume. The filters are spatially small along the width and height dimensions, but they have the same depth as the input volume. Although not required, practitioners usually set the filter to have the same size along width and height dimensions in practice and they call this hyperparameter **filter size**.

During the forward propagation, each 3D filter is applied by sliding it across the width and height dimensions of each input volume, producing a 2D feature map of that filter. Each time we slide the filter across the input, we compute the dot product between the entries of the filter and the 3D sliding window. The hyperparameter **stride** defines how far we slide the filter. Assuming stride as 1, we slide the filter by only 1 spatial unit for the next convolution. Also, a technique called zero-padding can be applied to add zeros surrounding the input volume, so the filter can be applied to the border elements of the input.

Figure 6.3 shows an example of 2D convolution. The input is 4×4 with zero padding. The filter size is 3, and the output is also 4×4 because we set stride size to be 1. The red window slides along the width and height of the input. Dot products between entries in the input red window and filter are performed and output to the resulting feature map.

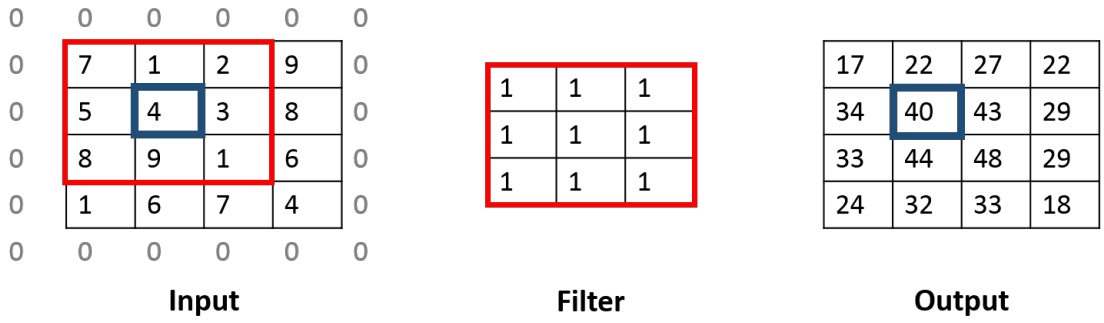


Figure 6.3: An example of 2D Convolution. Dot product for elements in the two red windows is performed.

6.2.2 Pooling Layer

Input to pooling layer is usually the output from convolutional layer after an element-wise non-linear transformation. The pooling layer is used to reduce the spatial size of the input through downsampling. By doing so, the amount of parameters and computation can be greatly reduced which can also help alleviate overfitting. The most common pooling operation in CNN is max pooling. It involves sliding a 2D window

along the width and height of the input on every channel. Each window outputs a max value of the elements within the window. Therefore, the output of the pooling layer is spatially downsampled on width and height but remains the same depth as the input. Similar to the convolutional layer, the output size depends on the choices of kernel size and stride. Figure 6.4 shows an example of performing max pooling on a 4×4 input with a filter size of 2 and stride size of 2. The maximum value of each window in the input is the output in the resulting feature map.

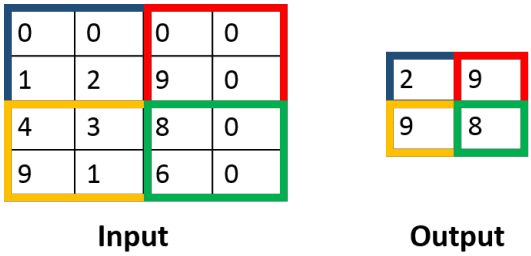


Figure 6.4: An example for max pooling. Different colors mean different pooling windows and the corresponding results.

6.2.3 Fully Connected Layer

The fully connected layer in CNN can be treated as the RBM used in DBN. An RBM is an energy-based generative model that consists of two layers: a layer of visible units v, and a layer of hidden units h. The units in different layers are fully connected with no connections among units in the same layer. Figure 6.5 shows a very small RBM with 4 units in the visible layer and 3 units in the hidden layer for illustration purposes. In total, there are 4×3 edges in the network. Weights associated with these edges are represented as a 4×3 weight matrix. In CNN, input of a fully connected layer is usually the output of a pooling layer. Each 3D volume from the pooling layer can be unrolled to a large 1D vector. The dimension of the 1D vector equals the visible layer size of the RBM. By unrolling all 3D volumes from the pooling layer, we are then able to represent them as a 2D matrix, which we then multiply with the weight matrix to derive the output of the fully connected layer.

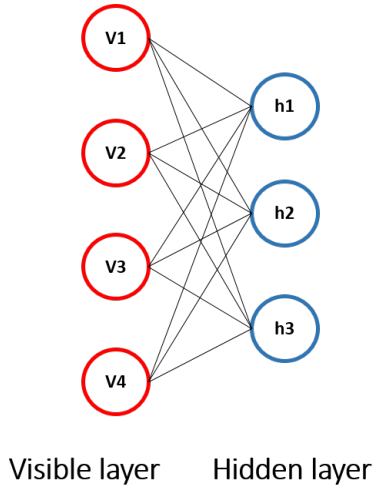


Figure 6.5: This figure shows a simple RBM with 4 units in the visible layer and 3 units in the hidden layer.

6.3 PIM Architecture

Figure 6.6 shows our system organization consisting of one host and four PIM stacks. Each PIM stack has a logic die containing the in-memory processor and memory (DRAM) dies on top of it.

In our system design, both the host and in-memory processors are Accelerated Processing Units (APU). Each APU consists of CPU and GPU cores on the same silicon die which enables the execution of both CPU- and GPU-oriented general-purpose code on either the host or PIM. Selecting an APU as the in-memory processor lowers the barrier to adoption and allows the use of existing rich sets of development tools for CPUs and GPUs. For this evaluation, we focus on the GPU execution units of the host and the PIM APUs.

The in-memory processor in each memory stack has high-bandwidth access to the memory stacked on it at a peak of 320 GB/s. The capacity of each memory stack is 4 GB. The host also has direct access to the memory stacked atop the PIM devices but at a reduced bandwidth, since those accesses must be over a board-level memory interface. We model the host memory interface on Hybrid Memory Cube (HMC) [62] at 160 GB/s peak bandwidth per memory stack (i.e., $1/2$ the internal bandwidth), which

results in aggregate host bandwidth of 640 GB/s across the four memory stacks. In order to model more mainstream hosts with lower memory bandwidth, we also evaluate designs where the host has 1/4 and 1/8 the internal bandwidth per memory stack.

We model a unified address space among the host and PIM devices that allows direct access from any PIM device to any memory within the system. Access to remote memory (i.e., memory in other PIM stacks) by PIM devices is modeled at 1/8 the intra-stack bandwidth per stack.

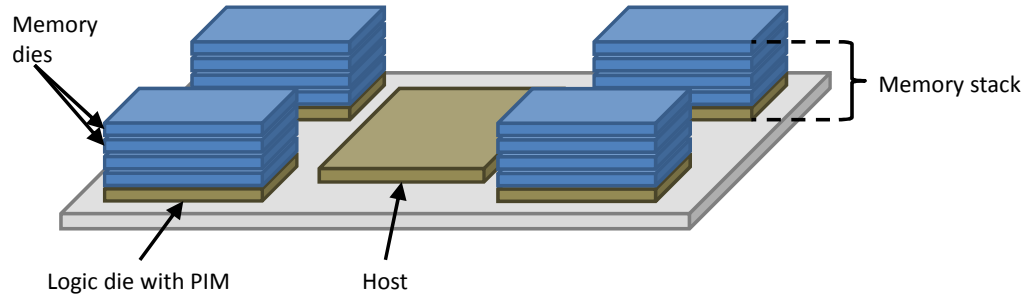


Figure 6.6: A node with four PIM stacks. Host can access all PIM stacks simultaneously. Each PIM stack can remotely access the other PIM stacks.

6.4 PIM Performance Model

A key challenge for memory systems research is the need for evaluating realistic applications with large datasets that can be prohibitive to run on cycle-level simulators. This issue is exacerbated as PIM expands the design space that must be explored. Therefore we perform our evaluations using a model that analyzes performance on existing hardware and uses machine learning techniques to predict the performance on future system organizations [92].

The model is constructed by executing a sufficiently large number of diverse kernels (the training set) on native GPU hardware and characterizing their execution through performance counters. Each hardware parameter that we are interested in scaling for future systems is varied to identify the performance sensitivity of each kernel to that hardware parameter. We then use a clustering algorithm to identify groups of kernels with similar scaling characteristics. Once the model is constructed, we can run

a new kernel at a single hardware configuration, and use its performance counters as a signature to map it to one of the clusters formed during model construction. The cluster identifies the scaling characteristics of the kernel, which is used to predict the performance for future machine configurations of interest, including PIM. The accuracy of this approach has been shown to be comparable to cycle-level models for exploring the design space of key architectural parameters such as compute throughput and memory bandwidth [92].

6.5 Deep Learning on Multiple PIMs

The key challenge in implementing deep learning algorithms on a system with multiple PIMs is partitioning the data among the memory stacks and dispatching the scoped compute kernels to the PIMs corresponding to the data partitions in order to exploit the high memory bandwidth available from each PIM to its local memory stack. Due to the high parallelism and throughput requirements of deep learning algorithms, we focus on the GPU execution units of the host and PIM APUs.

6.5.1 Data Parallelism and Model Parallelism

We explore two approaches to parallelize deep learning models on multiple PIM GPUs: data parallelism and model parallelism. In data parallelism, the input batch of images is partitioned across PIMs. Each PIM GPU gets a subset of the data and works on the full model. In model parallelism, the neural network model is partitioned across PIM GPUs. Each GPU works on one partition of the model using the full input batch.

The advantage of data parallelism is that each PIM gets a copy of the model, allowing each one to operate independently on its data without any inter-PIM communication. This is often desirable in cases where the model fits within the memory capacity of a single stack and the overhead of replicating the model on all PIM stacks is acceptable. Further, by increasing batch size, data parallelism can be made arbitrarily scalable and efficient. For example, if there are 8 PIMs and the batch size is 256, then

each PIM gets 32 input images which may result in low GPU usage. If we increase the batch size to 1024, then each PIM can get 128 input images and higher GPU usage. However, increasing the batch size can increase response time for latency-critical prediction tasks and adversely affect convergence rates in model training. Therefore, the batch sizes are typically set to be hundreds. For example, AlexNet uses a batch size of 128 and VGG nets [82] use 256. In this study, we evaluate batch sizes up to 1024.

The advantage of model parallelism is to enable training and prediction with much larger deep learning models. For example, COTS HPC system trains a network with more than 11 billion parameters which requires about 82 GB memory. Such a model is too large to fit into one single node using data parallelism, and thus needs to be partitioned using model parallelism. However, inter-PIM communication is inherent in model parallelism. As the model is partitioned across PIMs, each PIM can only compute a subset of neuron activities. They require synchronization to get the full neuron activities. In Figure 6.7, we show how to partition the RBM example from Figure 6.5 across two PIMs. PIM1 gets visible unit $v1$ and $v2$ while PIM2 gets visible unit $v3$ and $v4$. When computing the neuron activities of $h1$, $h2$, and $h3$, PIM1 only computes the contributions from $v1$ and $v2$ and PIM2 only computes the contributions from $v3$ and $v4$. However, the full activities come from all units in the visible layer; therefore the contributions from PIM1 and PIM2 are summed together to generate the correct result.

6.5.2 Convolutional Layer Parallelization

In deep learning models, convolutional layers cumulatively contain most of the computation, e.g., 90% to 95%, but only a small fraction of the parameters, e.g., 5% [50]. As we focus only on the prediction phase of deep learning in this study (i.e., there is no backward propagation and no weights update) the two parallelization schemes result in the same amount of computation for convolutional layers. Suppose there are I input images, F filters, and the image size is S by S. With a stride of 1, the total number of convolutions is $I \times F \times S \times S$. For data parallelism across N

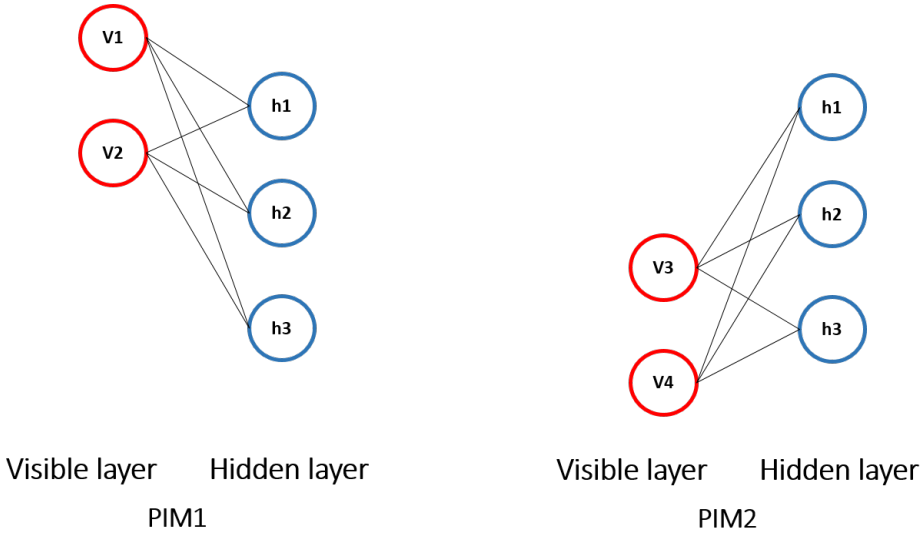


Figure 6.7: This figure shows model partitioning of the RBM example shown in Figure 6.5 across two PIMs.

PIMs, each PIM is responsible for $(I/N) \times F \times S \times S$ convolutions because input data is partitioned. For model parallelism, each PIM is assigned $I \times (F/N) \times S \times S$ convolutions because the set of filters is partitioned. Therefore, the amount of computation is the same for each PIM GPU no matter which parallelization scheme is applied. Further, due to the small size of the model parameter set, memory capacity pressure is not a factor in convolutional layers. Therefore, for simplicity, this study only evaluates data parallelism on convolutional layers. Given N PIM devices, the input batch of images is evenly partitioned to N mini batches. Each mini batch is assigned to one PIM and then propagates forward independently.

6.5.3 Pooling Layer Parallelization

For the pooling layers, there are no model parameters. Therefore, we can only apply data parallelism. However, depending on the parallelization scheme applied on the previous convolutional layer, the convolutional layer may have different groupings of the same output resulting in different input groupings to the pooling layer. This does not affect the correctness of the application. For example, consider an input to the previous convolutional layer with eight images and four filters. In model parallelism

across two PIM GPUs, each PIM outputs eight images where each image has two feature maps. In data parallelism, each PIM outputs four images where each image has four feature maps. Nevertheless, the total amount of computation for each PIM stays the same for the subsequent pooling layer.

6.5.4 Fully Connected Layer Parallelization

In contrast to the convolutional layers, fully connected layers contribute a small part of the computation, e.g., 5% to 10%, but the majority of the model parameters, e.g., 95% [50]. Fully connected layers can choose to deploy whichever parallelization scheme was used in previous layers. However, if the model is too large to fit into each PIM's memory, then model parallelism is required to train and predict at that large scale. Therefore, we evaluate both data and model parallelism on a fully connected layer. Please note that, by applying model parallelism on a fully connected layer, synchronization is needed at the end. As shown in Figure 6.7, hidden layer activities from PIM1 and PIM2 need to be summed together to get the correct hidden layer activities. In our implementation, this reduction across PIMs happens on the host. The host accesses each PIM's memory, performs the reduction, and writes the results back to, for example, PIM0. The other PIMs can then retrieve the results from PIM0.

6.6 Experimental Results

To show the potential of scaling deep learning algorithms on multiple PIMs, we evaluate three representative layers: the convolutional, pooling, and fully connected layers. We first run these layers with varying application parameters on an AMD Radeon^{TM} HD 7970 GPU with 32 compute units and 3 GB device memory. During each run, we profile and collect the performance counters of all kernels. We then scale the performance on the native hardware to multiple desired hosts and PIM configurations applying the methodology described in Section 6.4.

6.6.1 PIM configurations

In our experiments, we set the memory bandwidth to 320 GB/s and peak computation throughput to 650 GFLOPS for each PIM device. Two node organizations are explored. The first node organization has a host and four PIM stacks shown in Figure 6.6. The second one has a host and eight PIM stacks. The objective is to compare the performance of deep learning parallelization on multiple PIM devices against the performance of the host GPU. For accurate comparison, we set the peak FLOPS of the host GPU to be equal to the aggregate peak FLOPS of all PIM devices. The host accesses the memory stacks at lower bandwidth than in-stack memory access from the PIMs. Three host bandwidths are evaluated: 1/2, 1/4, and 1/8 of the in-stack PIM bandwidth per memory stack. However, the host can simultaneously access all the memory stacks.

Table 6.1 shows the configurations used for host and PIM. There are a total of six host configurations; three of them have four PIMs and the other three have eight PIMs. The host configurations are named in the pattern of *Host_N_B* where N is the number of PIM stacks and B is the total memory bandwidth available to the host. For example, *Host_4_640* means this host has 4 PIM stacks and 640 GB/s memory bandwidth in total. Two PIM configurations are listed as *4PIM* and *8PIM* in the table.

6.6.2 Results on Convolutional Layer

We first explore data parallelism on convolutional layer. The size of a single input image is 256 by 256 in width and height. The number of channels per image is 16. The number of images per input batch is 256. The number of filters is 16. The depth of each filter is set to be 16 but different filter sizes including 3, 5, 7, and 11 are evaluated. We select these filter sizes because they were used in state-of-the-art deep learning models. For example, AlexNet uses filter sizes 11, 5, and 3. VGG nets set the filter size to be 3. The stride size is set to be 1 for all cases for simplicity. For 4-PIM and 8-PIM configurations, the input batch is partitioned into mini batches, taking the

Table 6.1: This table shows different host and PIM configurations.

	Host_4_160	Host_4_320	Host_4_640	4PIM
Number of CUs	32	32	32	64
Engine Frequency (MHz)	1300	1300	1300	650
Total DRAM BW (GB/s)	160	320	640	1280
DRAM BW/stack (GB/s)	40	80	160	320
Number of DRAM stacks	4	4	4	4

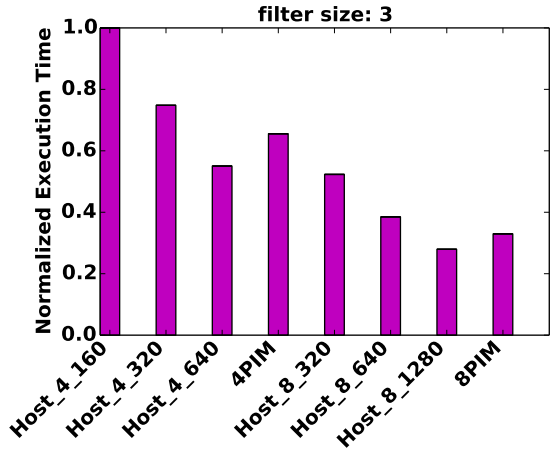
	Host_8_320	Host_8_640	Host_8_1280	8PIM
Number of CUs	64	64	64	128
Engine Frequency (MHz)	1300	1300	1300	650
Total DRAM BW (GB/s)	320	640	1280	2560
DRAM BW/stack (GB/s)	40	80	160	320
Number of DRAM stacks	8	8	8	8

data parallelism approach. Each PIM is assigned one mini batch and computes the convolution independently. For all host configurations, since there is only one GPU, no data partition is needed.

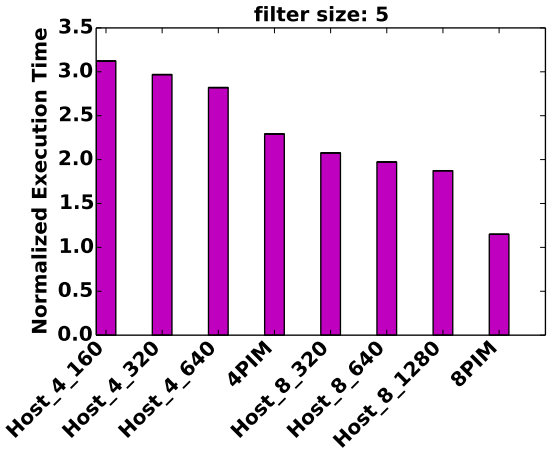
Figure 6.8 shows the normalized execution time for the convolutional layer. Each of the four subfigures shows results obtained using a particular filter size. In these subfigures, the Y-axis shows execution time normalized to *Host_4_160* with a filter size of 3. The X-axis lists the different configurations: six host design points, 4-PIM design, and 8-PIM design. When the filter size is 3, the execution times of *4PIM* and *8PIM* are slightly worse than the *Host_4_640* and *Host_8_1280* respectively. However, they do outperform the other host configurations. When the filter size is increased to 5, 7 or 11, running convolutional layer on multiple PIMs is faster than on all the host configurations. The observation fits our expectation because larger filter size means more memory access per convolution. The high memory bandwidth provided by PIM stacks makes it beneficial to run larger convolutions on PIM devices.

6.6.3 Results on Pooling Layer

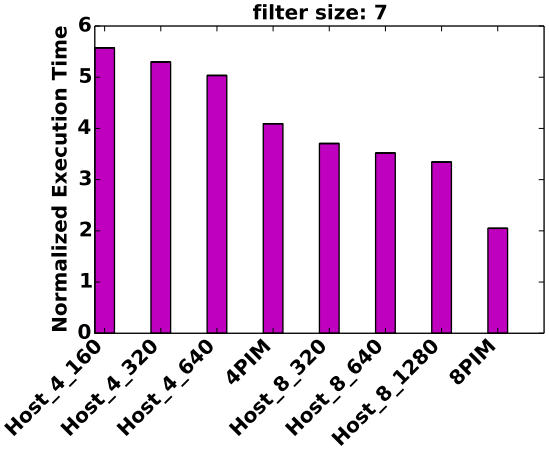
The pooling layer is also evaluated with data parallelism because it has a highly localized compute pattern and there is no neural network model. We again use the



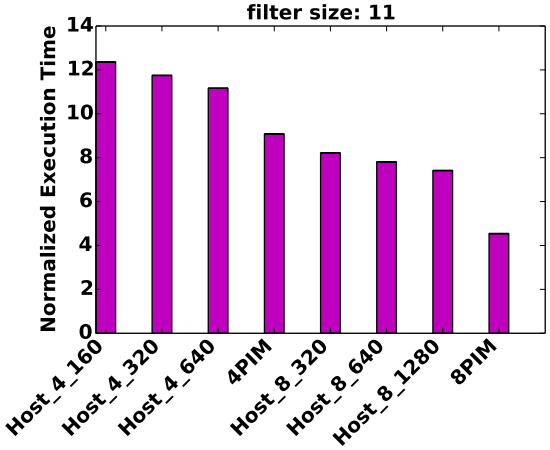
(a) Results from convolution filter size 3



(b) Results from convolution filter size 5

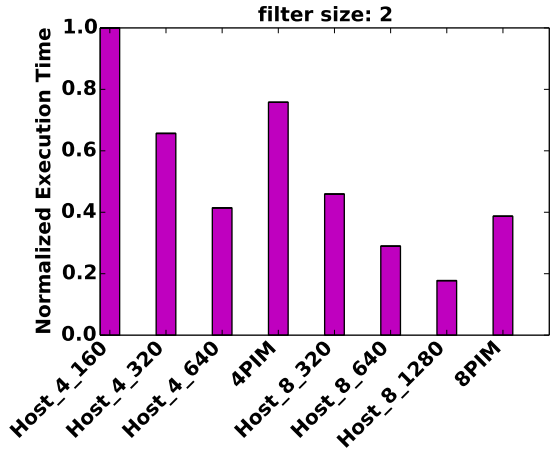


(c) Results from convolution filter size 7

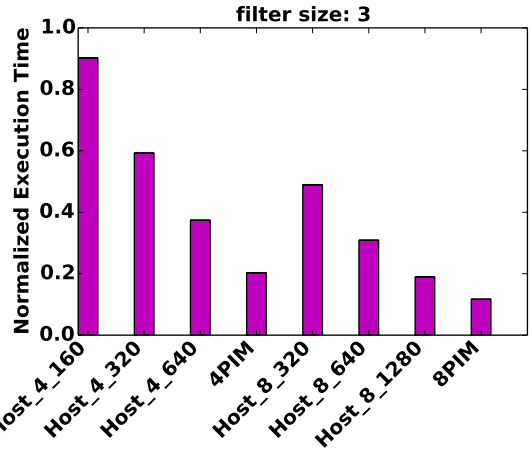


(d) Results from convolution filter size 11

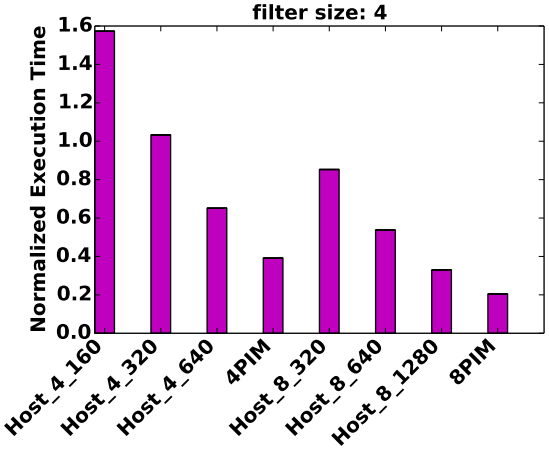
Figure 6.8: Convolutional layer results from different filter sizes. All results are normalized to *Host_4_160* with filter size 3 shown in Figure 6.8(a).



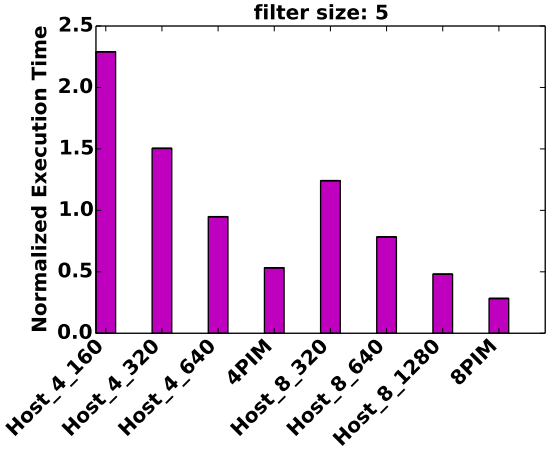
(a) Results from pooling filter size 2



(b) Results from pooling filter size 3



(c) Results from pooling filter size 4



(d) Results from pooling filter size 5

Figure 6.9: Pooling layer results from different filter sizes. All results are normalized to *Host_4_160* with filter size 2 shown in Figure 6.9(a).

single image size of 256 width by 256 height in pixels with 16 channels. Batch size is set to 256. There are various pooling operations used in deep learning applications. However, their computational and memory access characteristics are very similar. Hence, we pick the commonly used max pooling for our evaluation. The max pooling operation is performed using a 2D window at each channel of the input image. Large filters are typically not used in pooling because too much information can be lost. For example, AlexNet uses a filter size of 3 for pooling, VGG nets use 2 as the filter size, and a filter size of 5 was used in the COTS HPC system. As a result, we evaluate filter sizes 2, 3, 4, and 5. So we can evaluate how the performance changes as the filter size increases. The stride size is set to the filter size for simplicity.

Figure 6.9 shows the normalized execution time of the pooling layer using different filter sizes on the proposed configurations. We pick the execution time from *Host_4_160* using a filter size of 2 as the baseline for normalization. When filter size is small (e.g., 2), the performance of multiple PIM stacks is competitive with the host. As the filter size increases, more significant performance improvement is observed on the two PIM configurations. This observation is similar to convolutional layer results as it also benefits from the high memory bandwidth of the PIMs. As filter size increases, more memory access is needed for each pooling operation.

6.6.4 Results on Fully Connected Layer

We evaluate both data and model parallelism on a fully connected layer. We first compute the memory consumption per PIM for these two parallelization schemes. The results are shown in Figure 6.10. In this figure, the solid lines are for data parallelism and the dashed lines represent model parallelism. The red lines are for the 4-PIM configurations and the green lines are for the 8-PIM configurations. Different markers correspond to different input batch sizes. This figure shows that data parallelism consumes substantially more memory per PIM for one fully connected layer. Our evaluation shows that varying the number of PIMs or batch sizes does not change the memory consumption significantly for data parallelism, as the large number of model

parameters in the fully connected layer consumes most of the memory and is replicated on each memory module. However, in model parallelism, the model parameters are partitioned among the memory modules, resulting in moderate growth in memory capacity demand with both model size and input batch size. Further, with model parallelism, adding more PIMs reduces the memory pressure per PIM. In theory, if the batch size is large enough, model parallelism can consume a similar amount of memory as data parallelism. However, as batch size is typically small (e.g., 128 or 256), model parallelism is preferred in a fully connected layer due to lower memory consumption.

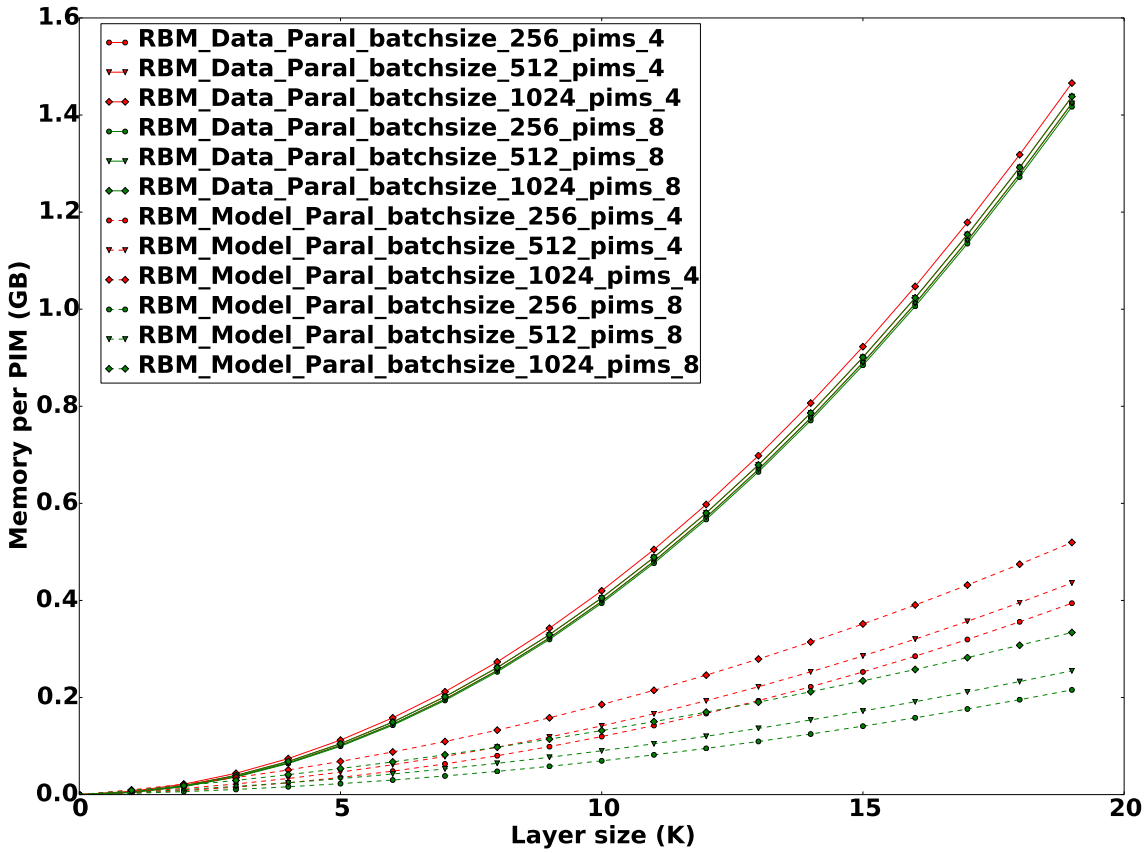


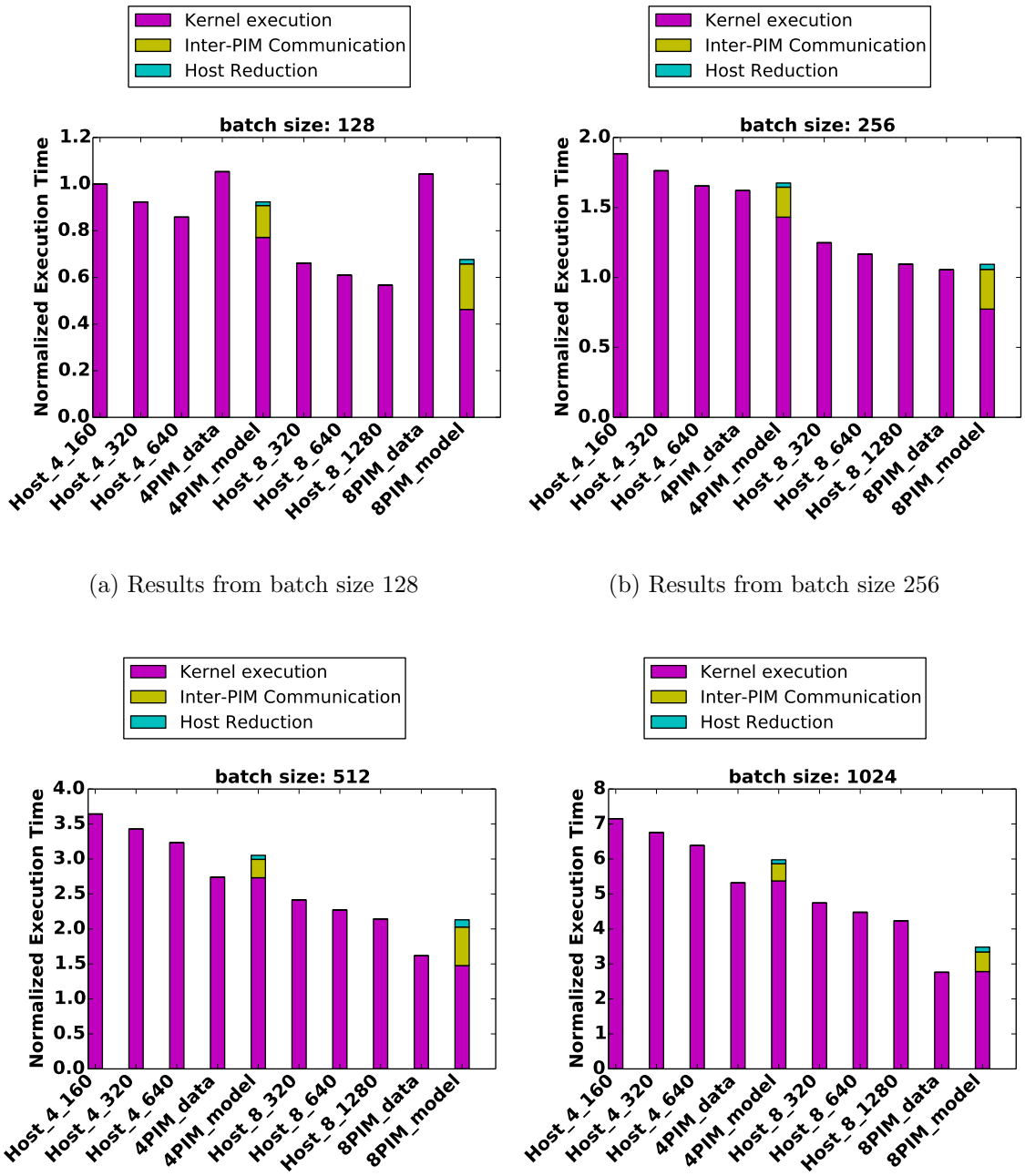
Figure 6.10: This figure shows memory consumption per PIM for data parallelism and model parallelism on fully connected layer.

We then evaluate the execution time for both data and model parallelization schemes using 4-PIM and 8-PIM configurations and compare them with host executions. Figure 6.11 records the results, normalized to *Host_4_160* with a batch size of

128. The four subfigures correspond to four different batch sizes: 128, 256, 512, and 1024. The layer size is fixed at 4096 which was used in AlexNet and VGG nets. For each subfigure, the Y-axis records normalized execution times and the X-axis lists different configurations. Please note that, *4PIM_data* means data parallelism on 4-PIM configuration. Similarly, *8PIM_model* stands for model parallelism on 8-PIM configuration. Because synchronization is needed in model parallelism, we use different colors to represent different components of execution time in the *4PIM_model* and *8PIM_model*. Purple represents OpenCL kernel execution time on PIM GPU, which excludes the reduction procedure that runs on the host GPU. The green segment represents the reduction across PIMs performed on host GPU, including memory access to all PIM stacks and writing the final results back to the first PIM in the configuration (PIM0). Yellow represents the time that all the other PIMs copy the reduction result from PIM0 to their local memory stacks. Please note that for PIM to PIM memory copy, we assume the memory bandwidth is 1/8 of the local in-stack memory access bandwidth. Therefore, for remote PIM memory access, the bandwidth is configured at 40 GB/s. This figure shows that model parallelism performs better than data parallelism and is comparable to host executions when batch size is small, e.g., 128. However, when we increase the batch size, model parallelism loses its advantage to data parallelism due to synchronization cost. However, with large batch sizes, both data and model parallelism on multiple PIM stacks outperform host executions.

6.7 Conclusion

In this chapter, we evaluate the performance of deep learning models on PIM devices. We study three types of layers from CNN and DBN, which are two of the most popular forms of deep learning models. The fully connected layer is parallelized across multiple PIM devices using data parallelism, which partitions the input set, and model parallelism, which partitions the model parameter set. Our results show that memory capacity requirements of data parallelism increase much more rapidly than for model parallelism as the model size increases. Further, we show that model



(a) Results from batch size 128

(b) Results from batch size 256

(c) Results from batch size 512

(d) Results from batch size 1024

Figure 6.11: Fully Connected layer results from different batch sizes. All results are normalized to *Host_4_160* with batch size 128 shown in Figure 6.11(a).

parallelism performs better at small input batch sizes while data parallelism performs better as input batch size increases. We parallelize convolutional and pooling layers across multiple PIM devices using data parallelism. We also vary key parameters for each of the layers over commonly used ranges of values. Our results show that PIM achieves competitive or superior performance compared to a high-performance host GPU across a variety of system and model parameter ranges.

Chapter 7

CONCLUSION

In this dissertation, we present a novel hybrid Android malware classification method named HADM. It consists of a static analysis method using multiple static features and a dynamic analysis method using novel graph-based representations. For the graph-based dynamic analysis, we use computationally expensive graph kernels to calculate graph similarities. Therefore, we parallelize the Shortest Path Graph Kernel (SPGK) using multiple-core CPUs and GPUs. For vector-based static and dynamic analysis methods, the bottleneck is deep learning computation. Since its parallelization has been studied on existing CPU and GPU architectures, we explore the potential for an emerging architecture design named PIM.

First, we propose a novel dynamic analysis method using graph representations converted from system call invocations recorded during executions of Android applications. We implement three traditional feature vector representations including histogram, n-gram, and the Markov Chain. These methods represent each Android application as one flat feature vector which is unable to capture rich structural information of a malicious application. Therefore we propose to augment previously studied feature vector representations with graph structure of the application's system call graph to improve the classification accuracy. Three graph representations including Histogram System Call Graphs (HSCG), N-gram System Call Graphs (NSCG), and Markov Chain System Call Graphs (MCSCG) are introduced in which each process is treated as a vertex and labeled with a feature vector. The graphs are then constructed by connecting parent/children processes. In HSCG, each vertex is labeled with a system call histogram generated from the invocations belonging to the corresponding process. Similarly, vertices in NSCG and MCSCG are associated with n-gram and Markov Chain

vectors. Graph kernels are then applied on the graphs to compute graph similarities that are subsequently classified with an SVM algorithm. To compare the performance of vector and graph representations, we collect thousands of Android applications from Google Play and VirusShare. In our dataset, 4002 samples are categorized as benign applications and 1886 samples are categorized as malware by VirusTotal. Experiments on this dataset show, for flat feature vector representations, the best classification accuracy that can be achieved is 83.3% by 4-gram vector. Experiments also show, for graph representations, NSCG-4 reaches the best classification accuracy at 87.3%. On average, graph representations are capable of improving the classification accuracies of the corresponding feature vector representations by 5.2%.

Second, since dynamic analysis has its drawbacks, we augment it with a static analysis method using multiple features and propose a hybrid Android malware classification method named HADM, in which both dynamic and static features are extracted including requested permissions, permission request APIs, used permissions, advertising networks, intent filters, suspicious calls, network APIs, providers, instruction sequences, and system call sequences. These features are represented as 16 feature vector sets and 4 graph sets. For each feature vector set, we train one DNN which is constructed by stacking four layer-by-layer pre-trained RBMs. The DNN learned features are concatenated with the original features to construct DNN vector sets. Different kernels are then applied to the DNN vector sets. Similarly, different graph kernels are applied to the graph sets. The learning results from different sets are combined using a two-level MKL to build a final hybrid classifier. In the first level, learning from different vector or graph kernels on the same vector or graph set are combined. In the second level, all learning results from the first level are combined. Experiments on the same dataset of benign and malicious Android applications show that the best classification accuracy that can be achieved using static analysis is 93.5% by a 4-gram instruction vector, and that the best accuracy using dynamic analysis is 87.3% by a 4-gram system call graph. Experiments also show that, by combining different features using hierarchical MKL, our final hybrid classifier is able to yield the best classification

accuracy of 94.7%.

Third, we evaluate the parallelization of shortest path graph kernel on multi-core CPUs and GPUs. We first analyze different drawbacks of the original Shortest Path Graph Kernel (SPGK) and propose Fast Computation of Shortest Path kernel (FCSP). We then parallelize FCSP on a multi-core CPU using two OpenMP methods. One focuses on the parallelization of FCSP on a single pair of graphs while the other focuses on the parallelization of kernel matrix construction. We also parallelize FCSP on the GPU using four OpenCL implementations. The first one applies 1D domain decomposition to the walk kernel while the second uses 2D decomposition. The third and the fourth implementations overlap communication with computation for the first and the second implementations. To evaluate the performance of different implementations, synthetic datasets, scientific datasets, and malware graphs are collected. Experiments on these datasets show FCSP reaches 76x speedup over the naive SPGK when running them sequentially on a single-core CPU. Experiments also show for small graphs, the OpenMP implementations on the CPU perform best while for large graphs the GPU implementations are better than the CPU implementations. Therefore, we propose a hybrid scheme that selects the CPU or GPU implementation based on graph size. Experiments show the hybrid algorithm works best for all of our datasets.

Last, we evaluate the parallelization of deep learning models on multiple PIM devices. Deep learning has been studied in existing architectures including CPUs and GPUs. However, little research has been done to evaluate the potential of emerging architecture design, such as PIM, which utilizes a 3D die stacking technology to move memory close to logic die and therefore reduces data movement overhead. We select three representative layers including the convolutional layer, pooling layer, and fully connected layer from popular deep learning models and then parallelize them across multiple PIMs. Two parallelization schemes are studied including data parallelism and model parallelism on a fully connected layer. Data parallelism partitions the input data across multiple PIMs, but each PIM keeps a full copy of the model. In contrast, model parallelism partitions the network model across PIMs. Each PIM works on one

model partition using the full input batch. Our results show that data parallelism consumes much more memory than model parallelism. Furthermore, we show model parallelism performs better at small input batch sizes while data parallelism performs better at large batch sizes due to the absence of synchronization. For convolutional and pooling layers, we parallelize them across multiple PIM devices using data parallelism because convolutional layers have a small model and pooling layers have no model. Our results show PIM achieves competitive or superior performance compared to a high-performance host GPU across a variety of system and model parameter ranges.

BIBLIOGRAPHY

[1] Contagio mobile malware. http://contagiominidump.blogspot.com. Accessed on Jan-21-2016.

[2] High bandwidth memory dram. https://www.jedec.org/standards-documents/docs/jesd235. Accessed on Jan-21-2016.

[3] Ltrace linux man page. http://linux.die.net/man/1/ltrace. Accessed on Jan-21-2016.

[4] Strace linux man page. http://linux.die.net/man/1/strace. Accessed on Jan-21-2016.

[5] N. Acosta-Mendoza, A. Morales-Gonzalez, A. Gago-Alonso, E. Garcia-Reyes, and J. Medina-Pagola. Image classification using frequent approximate subgraphs. In *Progress in Pattern Recognition, Image Analysis, Computer Vision, and Applications*. 2012.

[6] E. Aldea, J. Atif, and I. Bloch. Image classification using marginalized kernels for graphs. In *Proceedings of the 6th International Conference on Graph-based Representations in Pattern Recognition (GbRPR)*, pages 103–113, Berlin, Heidelberg, 2007.

[7] B. Amos, H. Turner, and J. White. Applying machine learning classifiers to dynamic android malware detection at scale. In *Proceedings of the 9th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1666–1671, July 2013.

[8] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 7(4):247–258, 2011.

[9] I. Arel, D. C. Rose, and T. P. Karnowski. Deep machine learning - a new frontier in artificial intelligence research. *IEEE Computational Intelligence Magazine*, 5(4):13–18, Nov 2010.

[10] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck. Drebin: Effective and explainable detection of android malware in your pocket. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2014.

[11] N. Bell and M. Garland. Efficient sparse matrix-vector multiplication on CUDA. NVIDIA Technical Report NVR-2008-004, NVIDIA Corporation, December 2008.

[12] F. Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the USENIX Annual Technical Conference (ATC)*, pages 41–41, Berkeley, CA, USA, 2005.

[13] Y. Bengio. Learning deep architectures for ai. *Foundations and trends in Machine Learning*, 2(1):1–127, 2009.

[14] T. Blasing, L. Batyuk, A. D. Schmidt, S. A. Camtepe, and S. Albayrak. An android application sandbox system for suspicious software detection. In *Proceedings of the 5th International Conference on Malicious and Unwanted Software (MALCON)*, pages 55–62, Oct 2010.

[15] K. M. Borgwardt and H. P. Kriegel. Shortest-path kernels on graphs. In *Proceedings of the IEEE International Conference on Data Mining (ICDM)*, pages 74–81, 2005.

[16] L. Brun and W. Kropatsch. Contains and inside relationships within combinatorial pyramids. *Pattern Recognition*, 39(4):515–526, April 2006.

[17] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: behavior-based malware detection system for android. In *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM)*, pages 15–26, 2011.

[18] D. Canali, A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis (ISSTA)*, pages 122–132, New York, NY, USA, 2012.

[19] G. Canfora, F. Mercaldo, and C. A. Visaggio. A classifier of malicious android applications. In *Proceedings of the 8th International Conference on Availability, Reliability and Security (ARES)*, pages 607–614, Sept 2013.

[20] R. Canzanese, S. Mancoridis, and M. Kam. Run-time classification of malicious processes using system call analysis. In *Proceedings of the 10th International conference on Malicious and Unwanted Software (MALCON)*, October 2015.

[21] A. Coates, B. Huval, T. Wang, D. Wu, B. Catanzaro, and N. Andrew. Deep learning with cots hpc systems. In *Proceedings of the 30th International Conference on Machine Learning (ICML)*, pages 1337–1345, 2013.

[22] N. Cristianini and J. Shawe-Taylor. *An introduction to support vector machines and other kernel-based learning methods*. Cambridge university press, 2000.

[23] O.E. David and N.S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *Proceedings of the International Joint Conference on Neural Networks (IJCNN)*, pages 1–8, July 2015.

[24] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, and Q. V. Le. Large scale distributed deep networks. In *Proceedings of the Neural Information Processing Systems (NIPS)*, pages 1223–1231, 2012.

[25] A. K. Debnath, R. L. Lopez de Compadre, G. Debnath, A. J. Shusterman, and C. Hansch. Structure-activity relationship of mutagenic aromatic and heteroaromatic nitro compounds. correlation with molecular orbital energies and hydrophobicity. *Journal of Medicinal Chemistry*, 34(2):786–797, 1991.

[26] J. Demme, M. Maycock, J. Schmitz, A. Tang, A. Waksman, S. Sethumadhavan, and S. Stolfo. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture (ISCA)*, pages 559–570, New York, NY, USA, 2013.

[27] L. Deng, M. L. Seltzer, D. Yu, A. Acero, A. R. Mohamed, and G. E. Hinton. Binary coding of speech spectrograms using a deep auto-encoder. In *INTERSPEECH*, pages 1692–1695, 2010.

[28] L. Deng and D. Yu. Deep learning: Methods and applications. *Found. Trends Signal Process.*, 7:197–387, June 2014.

[29] A. Desnos. Androguard - reverse engineering, malware and goodware analysis of android applications. http://code.google.com/p/androguard/. Accessed on Jan-21-2016.

[30] M. Dimjaševic, S. Atzeni, I. Ugrina, and Z. Rakamaric. Android malware detection based on system calls. Technical report, University of Utah, 2015.

[31] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2):6:1–6:42, March 2008.

[32] W. Enck, P. Gilbert, B. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI)*, 2010.

[33] A. Feizollah, N. B. Anuar, R. Salleh, and A. W. A. Wahab. A review on feature selection in mobile malware detection. *Digital Investigation*, 13(0):22 – 37, 2015.

[34] R. Floyd. Algorithm 97: Shortest path. *Communications of the ACM*, 5:345+, 1962.

[35] H. Gascon, F. Yamaguchi, D. Arp, and K. Rieck. Structural detection of android malware using embedded call graphs. In *Proceedings of the 2013 ACM Workshop on Artificial Intelligence and Security (AISec)*, pages 45–54, New York, NY, USA, 2013.

[36] W. Glodek and R. Harang. Rapid permissions-based detection and analysis of mobile malware using random decision forests. In *Proceedings of the IEEE Military Communications Conference (MILCOM)*, pages 980–985, Nov 2013.

[37] Mehmet Gonen and Ethem Alpaydin. Multiple kernel learning algorithms. *J. Mach. Learn. Res.*, 12:2211–2268, July 2011.

[38] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: Scalable and accurate zero-day android malware detection. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys)*, 2012.

[39] Y. J. Ham and H. Lee. Detection of malicious android mobile applications based on aggregated system call events. *International Journal of Computer and Communication Engineering*, 3, 2014.

[40] Y. J. Ham, D. Moon, H. Lee, J. D. Lim, and J. N. Kim. Android mobile application system call event pattern analysis for determination of malicious attack. *International Journal of Security and Its Applications (SERSC)*, 8(1):213–246, 2014.

[41] Z. Harchaoui and F. Bach. Image classification with segmentation graph kernels. In *Proceedings of the IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR)*, pages 1–8, 2007.

[42] P. Harish and P. J. Narayanan. Accelerating large graph algorithms on the gpu using cuda. In *Proceedings of the 14th international conference on High performance computing (HiPC)*, pages 197–208, Berlin, Heidelberg, 2007.

[43] G. E. Hinton. A practical guide to training restricted boltzmann machines. In *Neural Networks: Tricks of the Trade*, Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2012.

[44] G. E. Hinton, S. Osindero, and Y. Teh. A fast learning algorithm for deep belief nets. *Neural Computation*, 18(7):1527–1554, July 2006.

[45] S. Hong, T. Oguntebi, and K. Olukotun. Efficient parallel graph exploration on multi-core cpu and gpu. In *Proceedings of the International Conference on Parallel Architectures and Compilation Techniques (PACT)*, pages 78–88, 2011.

[46] IDC. Smartphone os market share, q1 2015. Technical report.

[47] A. Jain, S.V.N. Vishwanathan, and M. Varma. Spg-gmkl: Generalized multiple kernel learning with a million kernels. In *Proceedings of the 18th ACM International Conference on Knowledge Discovery and Data Mining (KDD)*.

[48] C. Jiang and F. Coenen. Graph-based image classification by weighting scheme. In *Applications and Innovations in Intelligent Systems XVI*, pages 63–76. Springer London, 2009.

[49] G. J. Katz and J. T. Kider, Jr. All-pairs shortest-paths for large graphs on the gpu. In *Proceedings of the 23rd ACM SIGGRAPH/EUROGRAPHICS symposium on Graphics Hardware (GH)*, pages 47–55, 2008.

[50] A. Krizhevsky. One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997, 2014.

[51] A. Krizhevsky and G. E. Hinton. Using very deep autoencoders for content-based image retrieval. In *Proceedings of the European Symposium on Artificial Neural Networks (ESANN)*, 2011.

[52] A. Krizhevsky, I. Sutskever, and G. E. Hinton. Imagenet classification with deep convolutional neural networks. In *Proceedings of the Neural Information Processing Systems (NIPS)*, pages 1097–1105. 2012.

[53] N. Le Roux and Y. Bengio. Representational power of restricted boltzmann machines and deep belief networks. *Neural Computation*, 20(6):1631–1649, June 2008.

[54] S. Lee, J. Lee, and H. Lee. Screening smartphone applications using behavioral signatures. In *Security and Privacy Protection in Information Processing Systems*, volume 405 of *IFIP Advances in Information and Communication Technology*, pages 14–27. 2013.

[55] M. Lindorfer, M. Neugschwandtner, and C. Platzer. Marvin: Efficient and Comprehensive Mobile App Classification Through Static and Dynamic Analysis. In *Proceedings of the 39th Annual International Computers, Software and Applications Conference (COMPSAC)*, 2015.

[56] M. Lindorfer, M. Neugschwandtner, L. Weichselbaum, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)*, 2014.

[57] B. D. Lund and J. W. Smith. A multi-stage cuda kernel for floyd-warshall. *CoRR*, abs/1001.4108, 2010.

[58] P. Mahé, N. Ueda, T. Akutsu, J.L. Perret, and J.P. Vert. Extensions of marginalized graph kernels. In *Proceedings of the International Conference on Machine Learning (ICML)*, pages 552–559, 2004.

[59] N. Mawston. Android captured record 85 percent share of global smartphone shipments in q2 2014. Smartphone report, Strategy Analystics, 2014.

[60] A. Morales-Gonzalez, N. Acosta-Mendoza, A. Gago-Alonso, E. B. Garcia-Reyes, and J. E. Medina-Pagola. A new proposal for graph-based image classification using frequent approximate subgraphs. *Pattern Recognition*, 47(1):169 – 177, 2014.

[61] NVIDIA. *Nvidia cuda programming guide: Version 3.2*. NVIDIA Corporation, 2010.

[62] J. T. Pawlowski. Hybrid memory cube: breakthrough dram performance with a fundamentally re-architected dram subsystem. In *Proceedings of the 23rd Hot Chips Symposium (HC)*, 2011.

[63] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proceedings of the 2012 ACM Conference on Computer and Communications Security (CCS)*, pages 241–252, New York, NY, USA, 2012.

[64] PulseSecure. 2015 mobile threat report. Technical report, 2015.

[65] M. Ranzato, Y. Boureau, and Y. L. Cun. Sparse feature learning for deep belief networks. In *Proceedings of the Neural Information Processing Systems (NIPS)*, pages 1185–1192. 2007.

[66] V. Rastogi, Y. Chen, and X. Jiang. Catch me if you can: Evaluating android anti-malware against transformation attacks. *IEEE Transactions on Information Forensics and Security*, 9(1):99–108, Jan 2014.

[67] D. K. S. Reddy, S. K. Dash, and A. K. Pujari. New malicious code detection using variable length n-grams. In *Information Systems Security*, volume 4332 of *Lecture Notes in Computer Science*, pages 276–288. 2006.

[68] A. Reina, A. Fattori, and L. Cavallaro. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. In *Proceedings of the 6th European Workshop on Systems Security (EuroSec)*, 2013.

[69] K. Rieck, P. Trinius, C. Willems, and T. Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, Dec 2011.

[70] J. Sahs and L. Khan. A machine learning approach to android malware detection. In *Proceedings of the European Intelligence and Security Informatics Conference (EISIC)*, pages 141–147, Aug 2012.

[71] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. Bringas, and G. lvarez. Puma: Permission usage to detect malware in android. In *International Joint Conference CISIS12-ICEUTE12-SOCO12 Special Sessions*, volume 189 of *Advances in Intelligent Systems and Computing*, pages 289–298. 2013.

[72] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, J. Nieves, P. G. Bringas, and G. Alvarez. Mama: Manifest analysis for malware detection in android. *Cybernetics and Systems - Intelligent Network Security and Survivability*, 44(6-7):469–488, October 2013.

[73] R. Sarikaya, G. E. Hinton, and A. Deoras. Application of deep belief networks for natural language understanding. *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, 22(4):778–784, April 2014.

[74] J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. *CoRR*, abs/1508.03096, 2015.

[75] J. Schmidhuber. Deep learning in neural networks: An overview. *CoRR*, abs/1404.7828, 2014.

[76] A. D. Schmidt, R. Bye, H. G. Schmidt, J. Clausen, O. Kiraz, K. A. Yuksel, S. A. Camtepe, and S. Albayrak. Static analysis of executables for collaborative malware detection on android. In *Proceedings of the IEEE International Conference on Communications (ICC)*, pages 1–5, June 2009.

[77] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[78] B. Schölkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, 2001.

[79] I. Schomburg, A. Chang, C. Ebeling, M. Gremse, C. Heldt, G. Huhn, and D. Schomburg. Brenda, the enzyme database: updates and major new developments. *Nucleic Acids Research*, 32(suppl 1):D431–D433, 2004.

[80] A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss. Andromaly: a behavioral malware detection framework for android devices. *Journal of Intelligent Information Systems*, 38(1):161–190, 2012.

[81] N. Shervashidze and K. Borgwardt. Fast subtree kernels on graphs. In *Proceedings of the Neural Information Processing Systems Conference (NIPS)*, pages 1660–1668, 2009.

[82] K. Simonyan and A. Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[83] M. Spreitzenbarth, T. Schreck, F. Echtler, D. Arp, and J. Hoffmann. Mobile-sandbox: combining static and dynamic analysis with machine-learning techniques. *International Journal of Information Security*, pages 1–13, 2014.

[84] K. Tam, S. J Khan, A. Fattori, and L. Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *Proceedings of the Symposium on Network and Distributed System Security (NDSS)*, 2015.

[85] F. Tchakounté and P. Dayang. System calls analysis of malwares on android. *International Journal of Science and Technology*, 2(9), 2013.

[86] C. Wagner, G. Wagener, R. State, and T. Engel. Malware analysis with graph kernels and support vector machines. In *Proceedings of the 4th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 63–68, Oct 2009.

[87] N. Wale and G. Karypis. Comparison of descriptor spaces for chemical compound retrieval and classification. In *Proceedings of the 6th International Conference on Data Mining (ICDM)*, pages 678–689, Washington, DC, USA, 2006.

[88] Y. Wei, H. Zhang, L. Ge, and R. Hardy. On behavior-based detection of malware on android platform. In *Proceedings of the IEEE Global Communications Conference (GLOBECOM)*, pages 814–819, Dec 2013.

[89] L. Weichselbaum, M. Neugschwandtner, M. Lindorfer, Y. Fratantonio, V. van der Veen, and C. Platzer. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Techical Report, TRISECLAB-0414-001*, 2014.

[90] B. Wolfe, K. Elish, and D. Yao. Comprehensive behavior profiling for proactive android malware detection. In *Information Security*, volume 8783 of *Lecture Notes in Computer Science*, pages 328–344. 2014.

[91] D. Wu, C. Mao, T. Wei, H. Lee, and K. Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Proceedings of the 7th Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69, Aug 2012.

[92] G. Wu, J. Greathouse, A. Lyashevsky, N. Jayasena, and D. Chiou. GPGPU performance and power estimation using machine learning. In *Proceedings of the IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, 2015.

[93] L. K. Yan and H. Yin. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *Proceedings of the 21st USENIX Security Symposium*, 2012.

[94] C. Yang, Z. Xu, G. Gu, V. Yegneswaran, and P. Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *Computer Security - ESORICS 2014*, Lecture Notes in Computer Science. 2014.

[95] Z. Yuan, Y. Lu, X. Wang, and Y. Xue. Droid-sec: deep learning in android malware detection. In *Proceedings of the ACM conference on SIGCOMM*, 2014.

[96] Z. Yuan, Y. Lu, and Y. Xue. Droiddetector: android malware characterization and detection using deep learning. *Tsinghua Science and Technology*, 21(01):114–123, 2016.

[97] D. Zhang, N. Jayasena, A. Lyashevsky, J. L. Greathouse, L. Xu, and M. Ignatowski. Top-pim: Throughput-oriented programmable processing in memory. In *Proceedings of the 23rd International Symposium on High-performance Parallel and Distributed Computing (HPDC)*, pages 85–98, New York, NY, USA, 2014.

[98] M. Zhang, Y. Duan, H. Yin, and Z. Zhao. Semantics-aware android malware classification using weighted contextual api dependency graphs. In *Proceedings of the 2014 ACM Conference on Computer and Communications Security (CCS)*, 2014.

[99] S. Zhao, X. Li, G. Xu, L. Zhang, and Z. Feng. Attack tree based android malware detection with hybrid analysis. In *Proceedings of the IEEE 13th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2014.

[100] C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou. Smartdroid: An automatic system for revealing ui-based trigger conditions in android applications. In *Proceedings of the 2nd ACM Workshop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 93–104, New York, NY, USA, 2012.

[101] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy (SP)*, pages 95–109, May 2012.

[102] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative Android markets. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, Feb 2012.