From Market to Device: Adaptive and Efficient Malware Detection for Android



FENG Ruitao

School of Computer Science and Engineering

A thesis submitted to the Nanyang Technological University
in partial fulfilment of the requirement for the degree of
Doctor of Philosophy (Ph.D)

2021

Statement of Originality

I hereby certify that the work embodied in this thesis is the result of original research, is free of plagiarised materials, and has not been submitted for a higher degree to any other University or Institution.

January 6, 2021

....

Date FENG Ruitao

Supervisor Declaration Statement

I have reviewed the content and presentation style of this thesis and declare it is free of plagiarism and of sufficient grammatical clarity to be examined. To the best of my knowledge, the research and writing are those of the candidate except as acknowledged in the Author Attribution Statement. I confirm that the investigations were conducted in accord with the ethics policies and integrity standards of Nanyang Technological University and that the research data are presented honestly and without prejudice.

January 6, 2021

....

Date LIN Shang-Wei

Authorship Attribution Statement

This thesis contains material from 5 papers published in or submitted to the following peer-reviewed international conference/journals in which I am listed as an author.

- Chapter 3 is accepted as Guozhu Meng, **Ruitao Feng**, Guangdong Bai, Kai Chen, Yang Liu, "DroidEcho: an in-depth dissection of malicious behaviors in Android applications", Cybersecurity, 2018.

  The contributions of the co-authors are as follows:

  - Prof. Guozhu Meng, Prof. Guangdong Bai, Dr. Kai Chen, Prof. Yang Liu provided the initial research directions.
  - Prof. Guozhu Meng suggested the analytical frameworks that can be explored in the manuscripts.
  - Prof. Guozhu Meng and I wrote project code and set the used tools.
  - Prof. Guozhu Meng and I analyzed and organized the experiment results.
  - Prof. Guozhu Meng, Prof. Guangdong Bai and I prepared the manuscript drafts. The manuscripts were revised together with Prof. Kai Chen, Prof. Yang Liu.

- Chapter 4 is accepted as **Ruitao Feng**, Yang Liu, Shangwei Lin, "A Performance-Sensitive Malware Detection System on Mobile Platform", ICFEM, 2019; **Ruitao Feng**, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, Shang-Wei Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform", ICECCS, 2019. and submitted as **Ruitao Feng**, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, Yang Liu, "A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices", IEEE Transactions on Information Forensics and Security (TIFS), 2020.

  The contributions of the co-authors are as follows:

  - Dr. Xiaofei Xie, Prof. Lei Ma, Prof. Guozhu Meng, Prof. Yang Liu and Prof. Shang-Wei Lin provided the initial research directions.

- Dr. Sen Chen, Dr. Xiaofei Xie, Prof. Guozhu Meng suggested the analytical frameworks that can be explored in the manuscripts.
- I wrote project code and set the used tools.
- Dr. Sen Chen and I analyzed and organized the experiment results.
- Dr. Sen Chen and I prepared the manuscript drafts. The manuscripts were revised together with Dr. Sen Chen, Dr. Xiaofei Xie, Prof. Guozhu Meng, Prof. Lei Ma, Prof. Yang Liu and Prof. Shang-Wei Lin.

- Chapter 5 is accepted as **Ruitao Feng**, Jing Qiang Lim, Sen Chen, Shang-Wei Lin, and Yang Liu, "SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices", ICECCS, 2020.

  The contributions of the co-authors are as follows:

  - I provided the initial research directions.
  - I suggested the analytical frameworks that can be explored in the manuscripts.
  - Jing Qiang Lim wrote project code and set the used tools.
  - Jing Qiang Lim and I analyzed and organized the experiment results.
  - Jing Qiang Lim and I prepared the manuscript drafts. The manuscripts were revised together with Dr. Sen Chen, Prof. Yang Liu and Prof. Shang-Wei Lin.

January 6, 2021

....

Date FENG Ruitao

Abstract

In the past few years, the market share ratio of Android System has been increased to a leading position. With that large user basis, the number of Android applications on Google Play has increased to 3 million till the year of 2018. However, not all of the applications in market can be surely prevented from security risks. API misuse and incorrect invocation by developers may cause significant data leakage or tangibly degrade user experience, etc. Meanwhile, due to the complexity of Android system and diversity of real usage scenarios, it is quite a challenge to solve all these problems within a strait forward way. Thus, we set our targets on providing solutions for the Android security problems towards different usage scenarios separately.

As we know, a precise representation for attacks can benefit the detection of malware in both accuracy and efficiency. However, it is still far from expectation to describe attacks precisely on the Android platform. In addition, new features on Android, such as communication mechanisms, introduce new challenges and difficulties for attack detection. Considering to solve the addressed problems by the side of service provider and security researcher, we propose abstract attack models to precisely capture the semantics of various Android attacks, which include the corresponding targets, involved behaviors as well as their execution dependency. Meanwhile, we construct a novel graph-based model called ICCG (Inter-component Communication Graph) to describe the internal control flows and inter-component communications of applications. The models take into account more communication channel with a maximized preservation of their program logics. With the guidance of the attack models, we propose a static searching approach to detect attacks hidden in ICCG. To reduce false positive rate, we introduce an additional dynamic confirmation step to check whether the detected attacks are false alarms. Experiments show that our integrated malware detection system, DroidEcho, can detect

attacks in both benchmark and real-world applications effectively and efficiently with a precision of 89.5%.

However, apart from the applications provided by the official market (i.e., Google Play Store), which can adopt a heavy and complicated detection approach (e.g., DroidEcho), apps from unofficial markets and third-party resources are always causing serious security threats to end-users. Meanwhile, it is a time-consuming task if the app is downloaded first and then uploaded to the server side for detection, because the network transmission has a lot of overhead. In addition, the uploading process also suffers from the threat of attackers. Consequently, a last line of defense on mobile devices is necessary and much-needed.

To address this problem, we propose an effective Android malware detection system, MobiTive, leveraging customized deep neural networks to provide a real-time and responsive detection environment on mobile devices. MobiTive is a pre-installed solution rather than an app scanning and monitoring engine using after installation, which is more practical and secure. Although a deep learning-based approach can be maintained on server side efficiently for malware detection, original deep learning models cannot be directly deployed and executed on mobile devices due to various performance limitations, such as computation power, memory size, and energy. Therefore, we evaluate and investigate the following key points: (1) the performance of different feature extraction methods based on source code or binary code; (2) the performance of different feature type selections for deep learning on mobile devices; (3) the detection accuracy of different deep neural networks on mobile devices; (4) the real-time detection performance and accuracy on different mobile devices; (5) the potential based on the evolution trend of mobile devices' specifications; and finally we further propose a practical solution (MobiTive) to detect Android malware on mobile devices.

Based on the evaluations and findings on MobiTive, we find that syntax features, such as permissions and API calls, lack the semantics which can represent the potential malicious behaviors and further result in more robust model with high accuracy for malware detection. We further propose an efficient Android malware detection system, named SeqMobile, which adopts behavior-based sequence features and leverages customized deep neural networks on mobile devices instead of the server end. Different from

the traditional sequence-based approaches on server end, to meet the performance demand on mobile devices, SeqMobile accepts three effective performance optimization methods to reduce the time of feature extraction and prediction. To evaluate the effectiveness and efficiency of our system, we conduct experiments from the following aspects 1) the detection accuracy of different recurrent neural networks (RNN); 2) the feature extraction performance on different mobile devices, and 3) the detection accuracy and prediction time cost of different sequence lengths. The results unveil that SeqMobile can effectively detect malware with high accuracy. Moreover, our performance optimization methods have proven to improve the performance of training and prediction by at least twofold. Additionally, to discover the potential performance optimization from the state-of-the-art TensorFlow model optimization toolkit for our sequence-based approach, we also provide an evaluation on the toolkit, which can serve as a guidance for other systems leveraging on sequence-based learning approach. Overall, we conclude that our sequence-based approach, together with our performance optimization methods, enable us to efficiently detect malware under the performance demands of mobile devices.

Acknowledgments

First and foremost, I would like to express my deep gratitude to my advisors Prof. LIU Yang and Prof. LIN Shang-Wei. Their invaluable advice, gentle care and continuous support encouraged me a lot on both research career and daily life throughout my Ph.D. study. I really appreciate the opportunity to study and work on those exciting research topics in NTU.

Meanwhile, I am grateful to my outstanding seniors, Dr. Chen Sen, Dr. Xie Xiaofei and Prof. Meng Guozhu. Their patient guidance on thinking and academic writing helped me a lot on improving my basic research skills. Their suggestions and comments in every discussion taught me too much on improving my research and challenging myself. I would also like to take this opportunity to thank Prof. Ma Lei, Dr. Su Ting, Prof. Chen Bihuan, Prof. Xue Yinxing, Dr. Chen Hongxu and Dr. Mahinthan Chandramohan for their suggestions and advice that helped me start my research career in the early time.

I would like to express appreciation for my friends and colleagues in the Cyber Security Lab (CSL) for their support and encouragement, especially, Dr. Zhou Yuan, Dr. Li Yuekang, Mr. Liu Shangqing, Dr. Du Xiaoning, Dr. Fan Lingling, Mr. Tang Yun, Mr. Xu Zhengzi, Prof. Fan Ming, Mr. David Berend, Dr. Wang Haijun, Dr. Wang Dongxia, Dr. Wu Zhimin, Mr. Peng Lunan, Prof. Wang Junjie, Dr. Wei Lei, Prof. Shi JianQi and many others. I take this opportunity to thank our laboratory executive Tan Suan Hai for providing technical support at CSL.

I would also like to take this opportunity to express appreciation for my old friends and roommates, Dr. Fang Zheng, Sun Yidan, Du jie and many others. Your friendship could always cheer me up, while I was upset and confused.

Finally, I want to express my utmost gratitude to my family. I am heavily indebted to my parents, Dr. Feng Zhiyong and Dr. Li Xiaohong, for their unconditional support,

understanding and trust. Their meticulously care and guidance will continue as one of my biggest sources of inspiration. Their care and company are precious treasures in my life. Meanwhile, I also want to thank my fiancee, Ms. Fu Qian, for her understanding, support and encouragement. Last but not least, I would also like to thank my little kittens, Mr. Pie and Tea, for the happiness they brought to us during coronavirus lockdown.

Contents

Abstract v
Acknowledgments viii
List of Figures xv
List of Tables xvii
List of Publications xix

1 Introduction **1**
1.1 Background 1
1.2 Thesis Motivation and Objective 3
1.2.1 Protection on Server 3
1.2.2 Protection on Device 3
1.3 Major Contributions 4
1.3.1 Chapter 3 6
1.3.2 Chapter 4 8
1.3.3 Chapter 5 9
1.4 Thesis Organization 11

2 Preliminaries and Literature Survey **12**
2.1 Preliminaries 12
2.2 Malware Detection 18
2.2.1 Static Analysis 18
2.2.2 Dynamic Analysis 19
2.2.3 Machine Learning 19
2.3 Chapter Summary 20

3 An in-depth dissection of malicious behaviors in Android applications on Server **22**

3.1 Introduction . . . 22
3.2 Semantic Model of Attack . . . 26
3.2.1 Building Blocks . . . 26
3.2.1.1 Assets . . . 26
3.2.1.2 Actions . . . 26
3.2.1.3 Triggers . . . 28
3.2.1.4 Flows . . . 29
3.2.2 Attack Models . . . 30
3.2.2.1 Attack Taxonomy . . . 30
3.2.2.2 Discussion . . . 32
3.2.2.3 Disclaimers . . . 33
3.3 The Inter-component Communication Graph . . . 33
3.3.1 Android Communication Medium . . . 33
3.3.2 Inter-component Communication Graph . . . 34
3.3.3 Sufficiency of ICCG . . . 36
3.4 System Design of DroidEcho . . . 36
3.4.1 Disclaimer Learning . . . 37
3.4.2 ICCG Construction . . . 38
3.4.2.1 Pointer Analysis . . . 39
3.4.2.2 Link Analysis . . . 40
3.4.2.3 Graph Assembling . . . 40
3.4.3 Attack Detection . . . 41
3.4.3.1 Action Recognition . . . 41
3.4.3.2 Reachability Analysis & Slicing . . . 43
3.4.3.3 Taint Analysis . . . 44
3.4.4 Dynamic Attack Confirmation . . . 45
3.5 Evaluation . . . 46
3.5.1 Evaluation on Malware Benchmark . . . 46
3.5.2 Evaluation on Real Applications . . . 47

3.5.3 Evaluation on Performance 49
3.6 Chapter Summary 49

4 A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices 51
4.1 Introduction 51
4.2 System Design of MobiTive 54
4.2.1 Overview of MobiTive 54
4.2.2 Feature Preparation 55
4.2.2.1 Feature Selection 55
4.2.2.2 Feature Updating 57
4.2.2.3 Feature Vector Extraction 57
4.2.3 DL Model Construction 58
4.2.3.1 DL Model Training 58
4.2.3.2 DL Model Migration and Quantization 59
4.2.4 Real-Time Detection System 60
4.3 Evaluation 62
4.3.1 Experiment Environment 62
4.3.2 Effectiveness of Feature Extraction, Feature Updating, Feature Category Selection, and Neural Network Selection 63
4.3.2.1 Performance evaluation of feature extraction 63
4.3.2.2 Accuracy evaluation of behavior-based feature updating method 65
4.3.2.3 Accuracy evaluation of feature category selection and deep neural network selection 68
4.3.2.4 Comparison between the existing learning-based Android malware detection systems and MobiTive 69
4.3.2.5 Accuracy evaluation of multi-class classification 71
4.3.3 Effectiveness Evaluation of MobiTive on Mobile Devices 73
4.3.3.1 Performance evaluation of feature preparation on real devices 73

4.3.3.2 Performance evaluation of RNN models on real device . 75
4.3.3.3 Accuracy and prediction time on different real devices . 76
4.3.3.4 Performance comparison between MobiTive and dynamic run-time detection 78
4.3.4 Analysis of Hardware Performance Evolution Trend of Android Mobile Devices 79
4.4 Limitations and Discussions 82
4.5 Chapter Summary 85

5 An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices 86
5.1 Introduction 86
5.2 System Design of SeqMobile 88
5.2.1 Overview 88
5.2.2 Feature Preparation 89
5.2.2.1 Feature Selection 90
5.2.2.2 Feature Dictionary Construction 92
5.2.2.3 Feature Extraction 92
5.2.3 Deep Learning Model Construction 93
5.2.3.1 Sequence Padding 94
5.2.3.2 Customized Deep Neural Network Architecture 94
5.2.4 Model Conversion and Quantization 95
5.2.5 Real Time Detection System 95
5.3 Evaluation 96
5.3.1 Experiment Environment and Dataset 96
5.3.2 Effectiveness Evaluation of Feature Selection, Deep Neural Networks, and Repetitive Pattern Removal in Feature Preparation and Network Training Phases 97
5.3.2.1 Accuracy Comparison Across Feature Categories and Deep Neural Networks 97
5.3.2.2 Effect of Repetitive Pattern Removal 99

5.3.3 Performance Evaluation and Optimization of Feature Extraction and Detection on Android Mobile Devices 101
5.3.3.1 Performance Comparison of Feature Extraction between Implementing with JNI and Java Language on Android Devices 102
5.3.3.2 Performance Optimizations in Prediction on Android Devices 103
5.3.4 Comparison between Previous Work and SeqMobile 107
5.4 Chapter Summary 108

6 Conclusions and Future Directions **109**
6.1 Conclusions 109
6.2 Future Directions 110
6.2.1 Adversarial Attack Against Learning Based Android Malware Detection 110
6.2.2 Automatic Code Re-construction and Manipulation Technique . . 111
6.2.3 Empirical Study of Interpretation Methods in Learning Based Android Malware Analysis 111
6.2.4 Robustness Evaluation for Android Malware Detection System . . 111
6.2.5 Effectiveness Study of Used Feature Representation Towards Usage Scenarios in Android Security Analysis 112

List of Figures

1.1 The overview of the Thesis 5

2.1 The overview of Android architecture[1] 13
2.2 Structure of the Dalvik executable file 15

3.1 An example of malicious behaviors and the corresponding ICCG 35
3.2 The architecture of the system 37
3.3 The learning process of disclaimers 38
3.4 Taint analysis across multiple domains 44

4.1 Overview of MobiTive 54
4.2 Extraction time of different raw data types 64
4.3 Extraction time of different feature types 66
4.4 Comparison of model size changes on migration and quantization 69
4.5 Unzipping time on different mobile devices 74
4.6 Extraction time on different mobile devices 75
4.7 The accuracy and prediction time of RNN models on Huawei P30 76
4.8 Benchmark score comparison among different chipsets from Exynos, Kirin, and Snapdragon 80
4.9 Top 21 chipsets assembled in Android mobile phones released in 2019 . . 81
4.10 RAM size of Android mobile devices released in 2019 81

5.1 Overview of SeqMobile 89
5.2 Distribution of extracted sequence length 97
5.3 Accuracy change across different sequence length on $N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API,Intent\}}$ 99

5.4 Training time across different sequence length and accuracy comparison between M_1 and M_2 101
5.5 Performance gain with JNI implementation 102
5.6 Feature extraction time cost for different APK sizes across different devices 103
5.7 Prediction time cost for different sequence lengths across different devices 104
5.8 Average prediction time between non-quantized dynamic Bi-LSTM and quantized static Bi-LSTM across different devices 106

List of Tables

3.1 The category of actions on Android 27
3.2 The category of attacks on Android 32
3.3 Comparison with FlowDroid 47
3.4 Privacy leakage via 3rd-libraries 49
3.5 Evaluation on Performance of DroidEcho 50

4.1 Selected Features 56
4.2 Deep Neural Network Architecture: GRU and LSTM 58
4.3 Deep Neural Network Architecture: Stacked GRU and LSTM 59
4.4 Deep Neural Network Architecture: Bidirectional GRU and LSTM 59
4.5 Deep Neural Network Architecture: CNN 60
4.6 Used Dataset 67
4.7 Detection results of feature updating 67
4.8 Detection results of feature categories and networks on server side 68
4.9 Comparison of MobiTive against existing approaches 70
4.10 Multi-Class Dataset 71
4.11 Detection result with multi-class dataset 72
4.12 Accuracy and performance of MobiTive on real mobile devices 77
4.13 Run-time performance comparison of MobiTive against dynamic Android analysis tool 78

5.1 An example of API mapping 90
5.2 Vocabulary size of each dictionary 92
5.3 Network architecture - Bi-LSTM 94
5.4 Dataset 96

5.5 Detection results of best feature category combinations across difference networks 98
5.6 Average sequence length 99
5.7 Non-quantized and quantized model size comparison 105
5.8 Comparison of SeqMobile against previous work 107

List of Publications

International Conferences

1. Ruitao Feng, Guozhu Meng, Xiaofei Xie, Ting Su, Yang Liu, Shang-Wei Lin, "Learning Performance Optimization from Code Changes for Android Apps", ICST Workshops, 2019.

2. Ruitao Feng, Yang Liu, Shangwei Lin, "A Performance-Sensitive Malware Detection System on Mobile Platform", ICFEM, 2019.

3. Ruitao Feng, Sen Chen, Xiaofei Xie, Lei Ma, Guozhu Meng, Yang Liu, Shang-Wei Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform", ICECCS, 2019.

4. Ruitao Feng, Jing Qiang Lim, Sen Chen, Shang-Wei Lin, and Yang Liu, "SeqMobile: An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices", ICECCS, 2020.

International Journals

1. Guozhu Meng, Ruitao Feng, Guangdong Bai, Kai Chen, Yang Liu, "DroidEcho: an in-depth dissection of malicious behaviors in Android applications", Cybersecurity, 2018.

2. Ruitao Feng, Sen Chen, Xiaofei Xie, Guozhu Meng, Shang-Wei Lin, Yang Liu, "A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices", IEEE Transactions on Information Forensics and Security (TIFS), 2020.

Chapter 1

Introduction

1.1 Background

In the past few years, Android system has dominated the mobile market. As a universal study shows, there would be more than 70% smart phones installed Android system. However, behind the large market share ratio, the Android system on different device from different manufacturers doesn't always locate in the same release version. In the year of 2018, Google already released 30 Android System Versions. On average, two release versions were released per year. That high updating frequency not only benefit on the bug fixing [1–3], vulnerability fixing [4–8] etc, but also make a lot of changes on Application Programing Interface (API) related features, which include both new features and deprecated usage. As a result, the hardware and customized system development from manufacturers will cause an unignorable diversity on the build-in Android system. Some study[1] shows that there are always more than 6 Android system version running in the market. From the release update log[2], there are always a lot API related updates between each pair of different versions.

In Android Application development, API gives developers access to resource features like Wifi, Location, etc, with required permissions. A series of problems, like security, performance, etc, may be caused. Security risks may come out when a sensitive resource is accessed by an API without any protection. Performance issues can be triggered if a bunch of APIs, which will occupy device computational resource like CPU, memory, etc,

[1] https://android.jlelse.eu/apple-vs-android-a-comparative-study-2017-c5799a0a1683/
[2] https://en.wikipedia.org/wiki/Android_version_history#Android_9.0_Pie_(API_28)

are invoked while it is unnecessary. Thus, using APIs in the correct way can surely avoid a lot of unpredictable security and performance problems. Otherwise, another difference between Android application and tradition software is that Android applications are executing within a life cycle, which mixed asynchronous and synchronous functionalities together. As a result, always invoking APIs in a correct way is quite a challenge for developers.

Also the Android mobile application market has expended to a quite large size, too. Based on the statistics in 2018, there are more than 3 million Android applications available on Google Play. With that such a large cardinal number, the security and privacy concerns are increasingly becoming the focus point to various Android mobile users and stakeholders. For example, more and more users are used to store their personal data in mobile devices through various popular apps such as shopping, banking, and social apps. However, it is always unable to guarantee the security and correctness of every application. Moreover, to seek profit from Android users illegally, the malware distributed in the markets may steal personal privacy (e.g., privacy leakage [9], privilege escalation [10, 11]) and even directly get money from users (e.g., financial charge [12], phishing [13], extortion [14]). Consequently, since the last decade, attackers shift their attention to mobile apps. That makes Android malware undoubtedly become one of the most important security threats in this security field [10, 15, 16].

Therefore, how to detect Android malware becomes a severe problem. On one hand, service providers of Android application markets and security researchers would prefer a end-to-end solution that can help them confirm malware with low false positive and further provide the relevant information (e.g., attack type, malicious behavior description, execution trace of malicious code). On the other hand, end-users always expect a secure environment which is maintained by the app markets. In other words, they consider their app sources are all trustable and secure enough. It is not surprising that the demands of Android malware detection approaches have been proposed, such as signature-based approaches [17–19], behavior-based approaches [20–23], data-flow analysis-based approaches [24–26], and machine learning-based approach [27–32].

1.2 Thesis Motivation and Objective

1.2.1 Protection on Server

The differences between diverse Android malware may locate in their attack targets, methods and the applied obfuscation techniques, etc. Some malware divides the their whole attack procedure into small parts and presents them within different malicious applications [33–36], performed periodically [10], and some of them are even able to be triggered while installing or receiving a broadcast message. As a result, the traditional malware detection techniques may be easily bypassed. For instance, the malware, which adopted the the techniques provided by ProGuard [37] and reflection [12], may become extremely risky.

Besides, greyware [38], which may request privileged permissions or hardware actions without being harmful like malware, is another challenge in detection. Therefore, the traditional detection systems, which used a imprecise and coarse-grained malicious behavior model, will lead to a high false positive rate in predicting greyware. Moreover, new execution paradigm, system libraries and rich communication features also make the detection harder, even a precise model of malicious behaviors is provided.

To overcome the security challenges, an integrated framework, which can automatically analyze Android applications with a desirable precision and scalability to benefit the further analysis/repairing, is necessary for Android detection task on server.

1.2.2 Protection on Device

With the available big data and hardware evolution over the past decade, deep learning has achieved tremendous success in many cutting-edge domains, including Android malware detection. Actually, the existing protecting solutions are mostly working on server for app markets/provider. However, when a new Android malware family is reported, not all the app markets are able to respond in a responsive time. The current analysis workflow always follows analyzing malicious behaviors within apps, building the detection models with the generated features and then performing the detection on the entire apps. Since the number of the real-world Android apps is extremely large, e.g., there are more than 3 million Android apps on Google Play Store, it is a time-consuming task

to perform the complete detection with that large number of apps. Moreover, the apps from unofficial markets and third-party resources like XDA [39] are more vulnerable in the wild. The security of these kinds of apps is indeed unpredictable and uncontrollable.

The traditional server-side based malware detection surely has unignorable drawbacks when detecting such apps, because (1) it is a time-consuming task to upload the apps to server before the installation, especially for large apps; (2) the uploading process via the Internet is not secure. For example, attackers may modify the malware during the uploading period such that an incorrect "benign" result is returned. As a result, the users will install the malware. Hence, a last line of defense on mobile devices is necessary and much-needed. To address the severe problem, we intend to conduct Android malware detection on mobile devices instead of server side.

Actually, machine learning-based approaches have achieved better performance compared with other approaches in Android malware detection [27, 29, 31, 40]. While a computationally intensive deep learning software could be executed efficiently on server-side with the GPU support, such deep learning models usually cannot be directly deployed and executed on other platforms supported by small mobile devices due to various computation resource limitations such as the computation power, memory size, and energy. Besides, some work [41] have been conducted the evaluation on the performance and robustness of models during training and inference phases, which revealed the potential compatibility issues on different platforms and frameworks. To guarantee the quality of the malware detection system when it is deployed on mobile devices, lots of deep learning techniques [42–51] have been proposed for testing the deep learning frameworks.

Thus, we intend to research the possible performance-sensitive Android malware detection solutions which directly work on mobile devices.

1.3 Major Contributions

The major contributions of this thesis are summarized below.

- To face the increasing threats from Android malicious applications, we propose DroidEcho as a server-end solution, which can search potential attacks using ICCG (i.e., inter-component communication graph) and further confirm the detected potential

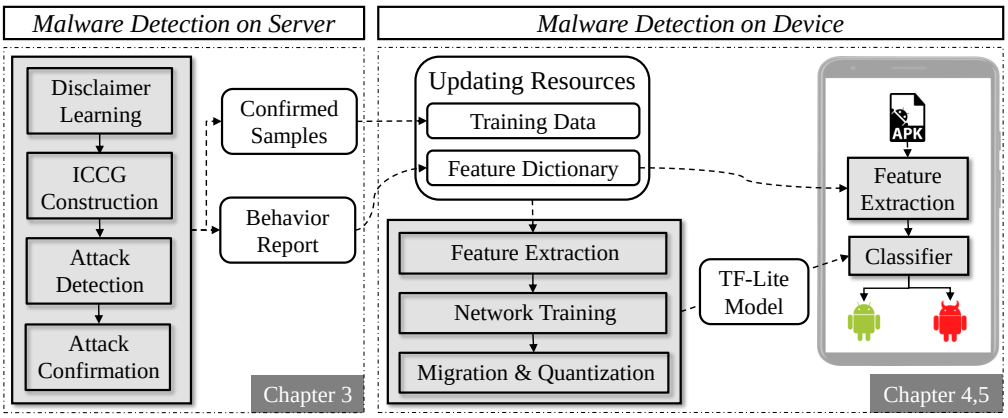


Fig. 1.1: The overview of the Thesis

attacks with novel attack models abstracted from malicious behaviors. We have evaluated DroidEcho on the malware benchmarks (i.e., Genome and DroidBench), and 7,643 real world applications. It shows that DroidEcho outperforms the state-of-the-art tool. Moreover, we have found out 444 applications with malicious behaviors in Google Play, and have a competitive edge in precision of 89.5% to the counterpart approaches and tools.

- To overcome the challenges brought by the high performance demand of Android mobile devices' user together with the uncertain situations in existing on-device protections, we propose MobiTive, which is a performance-sensitive malware detection system using deep learning on mobile devices directly. We have evaluated MobiTive on over 100,000 real world applications. It shows that MobiTive achieves an high accuracy at 96.75% with little time cost at 0.46s in detection.

- To discover more possibilities on adopting features with more semantic informantion to DL-based on-device malware detection systems, we further propose SeqMobile, which adopts behavior-based sequence features and customized deep neural networks to provide an effective and efficient malware detection service on Android devices. The evaluations of SeqMobile shows that SeqMobile achieves a relatively higher classification accuracy (i.e., 97.85%) as well as lower feature extraction and prediction time cost (i.e., <5 seconds). Moreover, we propose a novel optimization method to

enhance both the training (i.e., 53%) and testing (i.e., 57% on real device) time cost in learning approaches, when adopting sequential features as its input.

To further elaborate on the above points, the main contributions of each chapter are summarized below.

1.3.1 Chapter 3

To overcome the security challenges on PC/server, we propose an integrated framework called DroidEcho to analyze Android applications. First, we summarize the features of attacks happening on the Android platform, and propose a novel attack model. The model illustrates a variety of system API-based attack types at an abstract level, which is platform-independent. In particular, an attack is composed of: *assets*, which are the targets of attacks; *actions*, the execution operations with the API invocation performed on assets, and *triggers*, of which one entrance to the app that leads to the attack behaviors. Then we specialize the attack model into attack instances which are close to the Android platform, and can be utilized to guide our detection of attacks in a precise way.

Meanwhile, we transform Android applications into a comprehensive graph, incorporating call graphs between methods, and control flow graphs as per method. We conduct an in-depth static analysis through the graph with the guidance of attack model, and generate a full path with the trigger and the predicates that guarantee the occurrence of these behaviors. The detected malicious behaviors will be filtered by two conditions: if a seemingly malicious behavior is triggered by the user, it is likely that the behavior is user-intended, which we regard it as being harmless; presence of suspicious behaviors does not mean there is a real attack. It happens because some applications indeed need to carry out several seeming "malicious" behaviors to fulfill their tasks with good purposes. This is learnt and induced by investigating a group of applications under the same category or being similar. We make use of the mined social knowledge to filter out these harmless behaviors with a high level of confidence, i.e., these behaviors are likely a necessary part for applications. It does not only facilitate the efficiency of detection, but also reduce false positive in practice.

After the identification of malicious behaviors, we propose an approach to confirm the detected attacks with the dynamic execution. Our dynamic analysis is driven by the

attack traces generated previously, and provides a satisfied condition to guarantee the program to proceed along the trace. The dynamic execution reproduces the occurrence of attacks, and makes the attack detection more precise.

Different from the existing research on static analysis based approaches [24, 52–54], our work starts from the comprehension of Android malware by constructing semantic models. To reduce the false positive rate, we propose an approach to confirm attacks complying with the identified executed traces. To sum up, we make the following contributions:

- **Attack model.** We propose a novel representation, to characterize malicious behaviors. An attack in the model is constituted of target assets, execution actions, triggers, execution flows and apps' declaimers. It can facilitate the understanding of the essential features of attacks, and the detection of malware.

- **Accurate attack detection approach.** We propose a richly descriptive representation, named ICCG, to depict an Android application, with a maximal preservation of information. Based on ICCG, we design a synthetic approach to identify a malicious application by considering both the engineering aspect and the social aspect. A reduced but sufficient static analysis is to prove the presence of suspicious behaviors, then confirmed with the help of the learnt social knowledge.

- **Attack confirmation.** After the identification of malicious behaviors, we conduct a confirmation process to prove the existence of a real attack with dynamic execution. The dynamic execution is fed with the traces of malicious behaviors generated by DroidEcho, and further identifies the satisfiable conditions. Then it drives the application to execute along the traces, and thereby reproduces the attacks for confirmation.

- **Evaluation.** We have evaluated DroidEcho on the malware benchmarks (i.e., Genome and DroidBench), and 7,643 real world applications. It shows that DroidEcho outperforms the state-of-the-art tool. Moreover, we have found out 444 applications with malicious behaviors in Google Play, and have a competitive edge in precision of 89.5% to the counterpart approaches and tools.

1.3.2 Chapter 4

In this work, we intend to deploy the trained deep learning (DL) models from server-side to mobile devices. While a computationally intensive deep learning software could be executed efficiently on server-side with the GPU support, such deep learning models usually cannot be directly deployed and executed on other platforms supported by small mobile devices due to various computation resource limitations such as the computation power, memory size, and energy.

According to the evaluation metrics of accuracy and time cost from different features and neural networks, we propose an effective and efficient Android malware detection system on mobile devices, named MobiTive. MobiTive leverages (1) a newly-proposed feature extraction method from binary code; (2) a performance-based feature type selection mechanism; (3) a novel feature updating method through malicious behavior mining and understanding; (4) a customized deep neural network for classification. So that, MobiTive can provide a real-time and fast responsive environment on mobile devices.

In our comprehensive experiments, (1) we first divide the feature preparation procedure into two steps, which are raw data extraction and feature extraction, and evaluate the performance (time cost) separately to decide the feature selection. (2) With the selected features, we then provide an accuracy comparison between different feature categories. (3) The behavior-based feature updating method performs around 1%~5% accuracy increase. (4) We provide a comprehensive comparison between seven different neural networks (e.g., CNN, LSTM, and GRU) to show the potential improvement of our customized DL models on network definition. (5) We further evaluate the performance and accuracy of MobiTive on different real mobile devices by using our customized RNN model and compare with dynamic device-end solutions. (6) In the last part of our experiments, we perform an analysis of the performance trend on mobile devices from three different aspects and integrate the results to provide a strong evidence on the potential of MobiTive in practice. Specifically, MobiTive achieves a relatively higher classification accuracy (i.e., 96.78% accuracy) on real testing data in the wild and mobile devices with relatively lower overhead (i.e., less than 3 seconds on average for one app).

In summary, we make the following main contributions.

- **MobiTive.** We propose a device-end solution to protect mobile devices from malware threats in real-time efficiently by leveraging customized deep neural networks and binary features. This research work aims to detect malware directly on mobile devices as a pre-installed and run-time solution rather than detecting them on common servers or monitoring them after installation.

- **Binary level feature extraction.** We propose a new feature extraction method from binary code, as well as a feature updating method based on the understanding of malicious behaviors. Due to the high performance demand of mobile devices, we evaluate the different performance (time cost) and accuracy with various feature types and neural networks, and further provide a comparison against 4 existing Android malware detection approaches. Besides, we also investigate the accuracy on multi-class classification task.

- **Evaluation.** We evaluate and investigate the different performance on multiple devices from different manufacturers, and further provide insights of the current quality and potential for our approach according to the feature extraction and prediction time cost on six real mobile devices. Meanwhile, an additional comparison on run-time efficiency and discussion on effectiveness is provided to show the advantages against dynamic malware detection system based on behavior analysis.

1.3.3 Chapter 5

In this work, we propose SeqMobile, which adopts behavior-based sequence features and customized deep neural networks to provide an effective and efficient malware detection service on Android devices. To enhance the performance of SeqMobile, we propose a series of performance optimization methods that can effectively reduce the training and prediction time for sequence-based approaches. In the experiments, we first summarize and propose 8 feature categories (e.g., combination of permissions, intent filters, API sequence and intent sequence) and investigate their corresponding performance with different deep neural networks. We then perform an evaluation using accuracy and prediction time as metrics to decide on a suitable network configuration. After that, we accept the pre-trained model that yields the best results and deploy it onto Android

devices. To ensure that our pre-trained models are compatible with Android device, we convert our pre-trained models into lightweight TensorFlow Lite models. In our proposed system, we prioritize time cost over accuracy such that lower-end devices can choose to trade off less than 1% of the classification accuracy for lower prediction time. Overall, through our performance optimization methods, SeqMobile can achieve a relatively higher classification accuracy (i.e., 97.85%) as well as lower feature extraction and prediction time cost (i.e., <5 seconds).

In this work, we make the following contributions:

- **SeqMobile.** We propose an efficient sequence-based malware detection system, which adopts behavior-based sequence feature and customized deep neural network to provide an effective and efficient malware detection service on Android devices.

- **Binary level sequence-based feature extraction.** We present a systematic approach to directly extract the semantic feature sequence, which can provide information of certain malicious behaviors, from binary files under a certain time constraint. Thereby, achieving a relatively higher classification accuracy (i.e., 97.85%).

- **Repetitive elements removal.** We propose a method to remove repetitive elements in sequences and further evaluate how it can affect the overall performance of our malware detection system. Results has shown that our removal method significantly enhances the training and prediction performance with insignificant effects on the accuracy. To our best knowledge, this is the first comprehensive study on how removing repetitive elements in sequences can affect training and prediction performance in sequence-based learning approach on mobile devices.

- **Evaluation.** We conduct an evaluation on the state-of-the-art mobile-end model optimization toolkit provided by TensorFlow for our proposed sequence-based learning approach. The evaluation results can serve as a guidance for other mobile-end sequence-based learning approaches.

1.4 Thesis Organization

The remaining part of this thesis is organized as follows. First, the relevant preliminaries and background literature reviews are discussed in Chapter 2. Second, we propose a searching approach (DroidEcho) to detect potential attacks in Android applications using a graph based static program analysis technique. To capture the semantics of various Android attacks with their corresponding target and execution dependency, we propose attack models to represent the discovered attacks' behaviors. Meanwhile, to describe the internal control flows and inter-component communications within the applications, we construct a novel graph-based model called ICCG (i.e., inter-component communication graph) in Chapter 3. Third, we propose a device-end solution (MobiTive) to protect mobile devices from malware threats in real-time efficiently by leveraging customized deep neural networks and binary features in Chapter 4. To satisfy the high performance demand of mobile devices, we propose a new feature extraction method from binary code, as well as a feature updating method based on the understanding of malicious behaviors. Next, in Chapter 5, to discover the potential usage of semantic information in certain malicious behaviors under high performance demand, we propose an optimization method, which can significantly enhance the performance for both training and testing periods on sequence based Android Malware detection. Moreover, based on the optimization method, we propose an efficient sequence-based malware detection system, which adopts behavior-based sequence feature and customized deep neural network to provide an effective and efficient malware detection service on Android devices. Finally, the thesis is concluded in Chapter 6, with discussions on relevant future directions.

Chapter 2

Preliminaries and Literature Survey

2.1 Preliminaries

Android architecture. Android consists of four layers as shown in Fig. 2.1. All Android apps are deployed at the top layer *APPLICATIONS*. Android provides abundant APIs for Android apps to call, which are located on the layer *APPLICATION FRAMEWORK*. Usually, the application framework is written in Java, managing activities, notifications, resources, packages, and etc. On the next layer, *Android RUNTIME*, there are deployed with many core libraries and the runtime environment (e.g., Dalvik, ART). All the applications will run in an individual instance of Dalvik which has isolated memory, security policies, and a thread pool. At the bottom of the architecture, it is Linux kernel with some slight modifications by Google. Programs on this layer provide the drivers to the hardware, such as display, camera, power and audio. **Android apps**. There are four types of components defined in Android[2]. Each type of components can act as an entry point of a complete behavior when certain conditions are satisfied.

- **Activities.** An activity has an individual screen with a user interface. It shows content through GUI and receives input events through the pre-defined views in Android. An Android app may contains many activities, and one of them are specified as the main activity which is firstly started when running the app. And there exists dependency relationship between two activities, that is, one activity can start another activity.

[2]http://developer.android.com/guide/components/fundamentals.html

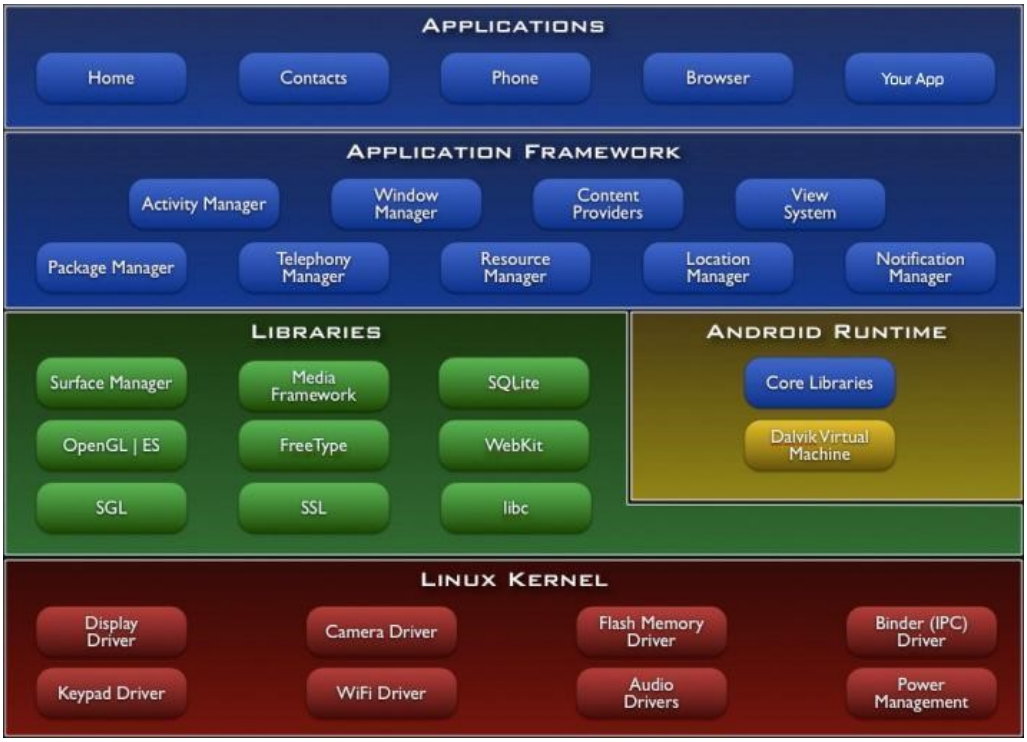


Fig. 2.1: The overview of Android architecture[1]

- **Services.** A service runs in the background to perform long-running operations or perform periodically work. The difference from activities is that a service does not contain an interface, which could not interact with users directly. However, services can be controlled or directed by associated activities or other system events, such as a broadcast message and a timer.

- **Content providers.** A content provider manage a set of data shared by apps. Although it is allowed to store data through various ways, for example, file system, SQLite database, a content provider provides a more flexible place to store and manage the data. Usually, one app needs first apply a permission to read or write the content provider. For example, the contact list in a phone is managed with a content provider, and one app needs acquire the permission READ_CONTACTS to access the contact, and the permission WRITE_CONTACTS to write a piece of contact information into contacts.

[2]https://elinux.org/Android_Architecture

- **Broadcast receivers.** A broadcast receiver is a component which can respond to broadcast message issued by system. For example, the system will announce a message reminding apps that the battery is low or the system has been rebooted. In that case, a broadcast receiver can capture these system events, and act correspondingly. One app can shutdown several consuming services if the battery is low, and initiate its services once the system is booted up.

In addition, there is a kind of messengers commonly existing in the communications amongst these components—Intent. It is an abstract description of an operation to be performed. An activity can be easily started by invoking an Android api startActivity with an Intent object as a parameter. This Intent object specifies the target activity, and data transmitted to the activity and so on. Besides, an Intent object can be used in the communication between different apps, for example, it is used in Android Interface Definition Language (AIDL) which allows remote invocation.

Android application package (APK). To distribute and execute Android applications on devices, developers will compile and compress the source code, XML files and other relevant components (e.g., application structure files and other resources) into an Android application package (APK). Considering each APK, it always contains a binary format manifest file (i.e., AndroidManifest.xml), one or more Dex files, used assets, linked resources and the certificates.

The manifest file defines the meta-data of Android apps. It contains application ID, package name, together with other app components (e.g., Intent filters, activities, and services), used permissions, device compatibility (e.g., uses-feature and uses-sdk). The Dalvik executable files (i.e., .dex), which is compiled from the Java or Kotlin source code, can be executed in Android OS with the help of Dalvik virtual machine. As a result of the difficulties in manually analyzing the binary files, in previous research, the Dex files are often disassembled back to smali code level by reverse engineering technique. Thus, researcher can obtain the smali files that have same meaning contents, but have a better syntax format than the binary files. However, as a result, the disassembling procedures from Dex to smali will definitely cause a considerable time cost during analysis.

Dalvik executable. Dalvik executable file contains 21 kinds of contents, which can be mainly divided into metadata and program information, etc as shown in Fig. 2.2. The

Header	
Checksum	
Signature	
File Size	Header size
Endian tag	Link Size
Link Offset	Map Offset
String IDs Size	String IDs Offset
Type IDs Size	Type IDs Offset
Proto IDs Size	Proto IDs Offset
Field IDs Size	Field IDs Offset
Method IDs Size	Method IDs Offset
Classdef IDs Size	Classdef IDs Offset
Data Size	Data Offset

Fig. 2.2: Structure of the Dalvik executable file

metadata information of Dalvik executable file is provided by the header, checksum, signature, etc. After them, it follows with the size and offset values of program information, like class definition identifiers, method identifiers, type identifiers, string identifiers, etc. Besides the above program information, the map offset is also an important component. It provides concrete mappings between static information, like strings and method names, etc. With the given offsets, we can easily access the defined static program information without decompiling the binary executable into human readable format. Other than the static information, we can also get the compiled code contents with the offset of each method in different classes.

Security mechanisms for Android. By investigating the existing security mechanisms for Android, we categoried them by their working environments into application market, Android OS platform, and device-end in practice.

From the aspect of Android application market, the Android official market (i.e., Google Play Store) provides their protection by performing a security verification on the uploaded APKs. For example, a machine learning based detection technique is announced to be provided by the Google Play Store. Besides, considering the third-party markets, some of them also present high-quality security protection for users. For instance, ApkMirror [55] provides a protection service powered by GuardSquare besides the signature verification. However, reliable third-party markets only occupy a small portion. Currently, most of them provide very simple and limited security check services (e.g., signature verification), which can be bypassed easily. Therefore, users of the

insecure third-party markets have to take their security risk, while install and use the applications downloaded from those markets.

On the mobile devices, the most famous antivirus applications (e.g., AVG and Avast) mainly provide their protection services by monitoring the privacy-sensitive components (e.g., permission requests granted in real-time), and searching the suspicious apps with their local or on-cloud malicious signature database. Besides the protection techniques out of Android OS, some strong built-in security mechanisms (e.g., application sandbox) are also provided by Google. The sandbox mechanism provides the protection by allocating an independent execution environment for every application. Thus, it restricts the triggered attack to its own requested components in the malicious application. For example, the malicious application will be forbidden to access Bluetooth with API calls, if the corresponding permissions and actions are not required in the application.

Deep learning model migration and quantization. While deploying a pre-trained deep learning model to a target device, it often goes through platform migration before deployed to user-end applications. The reason is that the training phase requires a vast amount of computation and energy resources. As the model size and the complexity of the tasks grow, more data are needed to train the network till reaching optimality, which could spend a long time on high-performance GPU clusters. Meanwhile, the deployment of the DNN models is usually faced with the resource-constrained environment with limited computation, storage, and power.

As a result of the environment differences of target platforms (e.g., embedded systems, mobile phones) and training platforms (e.g., GPUs), a deep learning model often goes through a customization phase to cater specific hardware and software that constraints of a specific target platform. Mostly, quantization technique can improve the computation efficiency, reduce memory consumption and storage size with little precision lose, which has become a common practice when migrating a large deep learning model to a low power computation system (e.g., mobile or IoT devices).

Recently, the rapid development of system-on-chip (SoC) acceleration (e.g., Qualcomm Snapdragon, Samsung Exynos9, and Kirin 970) provides the foundation and hardware support for universal deployment across multiple platforms, especially on mobile device, edge computing device and so forth. With the mobile platforms supporting, such

as TensorFlow Lite [56], Caffe2 Mobile [57], CoreML [58] and PyTorch Android [59], the lightweight solutions, which using the DL-based malware detection techniques, become possible to be deployed on a mobile device directly.

Sequence representation of application behavior. Mostly, Android applications provide their functionalities with basic behaviors that are represented using permissions, intent, API calls and etc. However, the analysis of Android malware shows that there are high risks that those basic behaviors inside applications may be accepted as a part of malicious functionalities. For example, considering a spyware, no matter how much it hides its malicious functionality, there will still be necessary basic behaviors existing to access the private information from those devices. Thus, a semantic representation of the basic behaviors, like sequence-based feature, will be beneficial in providing the corresponding potential malicious information unlike the traditional syntax feature in learning approaches.

Native code implementation. Considering the architecture of an Android system, the functionalities in Java implementation will be executed on Dalvik Virtual Machine. Different from traditional operating systems, when facing computationally intensive tasks, significant performance problems may occur on mobile devices. To meet certain performance criteria on Android devices, developers often investigate the bottleneck of their code and attempt improve the performance by re-implementing it using native code (C and C++), which can then be invoked through the Java Native Interface (JNI) [60]. As a result, Google recommends developers to perform the heavy operations on the native-end and return the results to the Java-end through JNI.

Static and dynamic RNN. Known for their recurrent structure and the internal mechanism that stores the information on the previous state and forward it to the next state, RNNs are preferred when it comes to sequence-based approaches. There are two types of RNN implementations namely, static and dynamic RNNs. The main difference between them is that dynamic RNN is configured to accept variable length input while static RNN only accepts fixed length input. Having said that, when a static RNN is used, padding or truncation needs to be perform on the input sequence to ensure it matches the defined input length requirement of the model. In contrast, only truncation is needed when for dynamic RNN so that the input sequence length does not exceed the defined required length.

2.2 Malware Detection

Generally, we will discuss the relevant literature from 3 aspects to provide an brief on the existing malware detection approaches according to their used techniques.

2.2.1 Static Analysis

Techniques using static features are widely used in Android malware detection area. Basically, the approaches can be categorized into three aspects according to their used features, as follows:

Approaches based on analyzing the information in XML files.

E. Chin et al. [61] proposed ComDroid, which detects application communication vulnerabilities by analyzing the manifest file. ComDroid found 34 exploitable vulnerabilities within an analysis of 20 applications. C.-Y. Huang et al. [62] classified the benign data and malware data using the permission information in manifest and files structure as features. It was able to detect more than 81% of malicious samples in their dataset. As well, Z. Aung et al. [63] considered the permission in a different way. They considered the permission requests in the code together with the defined permission information in manifest.

Approaches based on analyzing the API calls.

Y. Zhongyang et al. [64] introduced DroidAlarm, which analyzes the inter-procedural call graphs constructed by the relationship between permissions and the interface to identify attacks. D.-J. Wu et al. [65] proposed the technique, DroidMat, to detect malware with API traces, intent, communication and some other the life-cycle information. L. Deshotels et al. [66] classified the benign apps and malware based on the frequency of API calls. M. Zhang et al. [67] developed a classifier, DroidSIFT, which is based on the API dependency graphs.

Approaches based on analyzing the program analysis result (e.g., call graph).

Narayanan et al. [68] proposed DroidOL, which uses the generated control flow graph as the input of an online SVM classifier. In a large-scale analysis with a dataset of more than 87,000 apps, DroidOL achieved an accuracy of 84.29%. Narayanan et al. [69, 70] proposed graph kernel based approaches to represent the Android malware and further

evaluated their approaches in real detection task. Narayanan et al. [71] proposed MKL-Droid that used the Multiple Kernel Learning (i.e., MKL) system to capture the malicious information in malware. G. Z. Meng et al. [72] proposed SMART, a deterministic symbolic automaton (DSA) based detection system, in which DSA contains the corresponding components of the target app. SMART identified 4583 new malware missed by most anti-virus tools in their experiments. G. Z. Meng et al. [73, 74] proposed Mystique and Mystique-S, which are two frameworks to automatically generate malware by adopting the software product line engineering approach. G. Z. Meng et al. [75] performed a study on the spread of malware between markets, and thereby helps security practitioners to take measures against malware instantly.

2.2.2 Dynamic Analysis

In some special scenarios (e.g., malicious behaviors triggered in real-time), the false positive rate of static analysis techniques may suffer from its mechanism. As a result, some dynamic analysis based techniques are adopted in area as alternative solutions.

A. Shabtai et al. [76] and A.-D. Schmidt et al. [77] provided the abnormalities identification systems, which use the information profiled from device in real-time, such as CPU usage etc. M. Grace et al. [78] proposed RiskRanker, which detect malware by profiling and analyzing app's behaviors with defined rules. L. K. Yan et al. [20] proposed DroidScope, which abstracts semantic information from Dalvik opcode and API calls traces. W. Enck et al. [79] proposed a taint analysis tool, named TaintDroid,, for Android apps. It can analyze the potential malicious behaviors by monitoring the communications between functions. K. Basu et al. [80] used performance counter to monitoring the malicious behaviors while doing real-time analysis in malware/bug detection task.

2.2.3 Machine Learning

Since computing power increased significantly in past years, machine learning has achieved great success in malware detection. There exist also a lot of learning-based approaches as follows:

D. Arp et al. [27] proposed a classifier based on machine learning technique, named Drebin. It uses predefined features generated from XML files and API calls as its input.

W. Yu et al. [32] presented a malware detection solution whose input features are the defined permission and API call traces. Z. Yuan et al. [40] provided DroidDetector, which is an Android malware detection engine based on a deep belief network. It was able to achieve 96.76% detection accuracy. S. Chen et al. [29] introduced a streaminglized machine learning-based malware detection framework. L. L. Fan et al. [81] proposed Begonia, a malware detection system through Pareto ensemble pruning. They convert the malware detection problem into the bi-objective Pareto optimization, aiming to trade off the classification accuracy and the size of classifiers as two objectives. S. Chen et al. [31, 82] proposed KuafuDet, a two-phase learning enhancing approach that adversarially detects the Android malware. N. McLaughlin et al. [83] provided a CNN based Android malware detection system, which uses opcode sequences of target apps as the input feature. K. Xu et al. [84] proposed DeepRefiner, which is an efficient two layer malware detection system. They adopted XML features as the first layer to perform a fast detection. At the end of the first layer, if result cannot be promised with a high rate, some more complicated features, like bytecode information, etc., will be involved in the second layer to determine whether the target application is malicious. The result shows that DeepRefiner achieved an accuracy of 97.74% and a false positive rate of 2.54% on their dataset, which has 62,915 malicious applications and 47,525 benign. Kim et al. [85] presented a malware detection framework based on the detection results of multiple neural networks with different features as input. Every network has a single feature input and output score. The final detection result is a combination of all the models. S. Chen et al. [86] investigated the ability of poisoning attack on the four most recent machine-learning detection systems in academia, and further proposed a general threat model to characterize attackers in different scenarios. B. Z. Wu et al. [87] proposed a novel and interpretable ML-based approach (named XMal) to classify malware with high accuracy and explain the classification result meanwhile.

2.3 Chapter Summary

In this chapter, we first introduced the preliminaries on the architecture of Android OS, the mechanism of Android application, the structure of Android application package

(APK) and Dalvik executable file (i.e., .dex files), existing security mechanisms, the techniques for deep learning model migration and quantization, the basic concept of sequence representation of application behavior, the executing mechanism of native code implementation and the concept of static and dynamic RNN. Then, we reviewed the existing approaches in Android malware detection area and further organized them along their based techniques and used features.

Chapter 3

An in-depth dissection of malicious behaviors in Android applications on Server

3.1 Introduction

As one of the most serious among different security problems on Android devices, Android malware varies in many aspects such as attack targets, attack methods, and applied obfuscation techniques. For example, Android malware may steal users' sensitive information [24, 88], elevate their privilege [89, 90], deplete device resources [91, 92], and remote control users' devices [10]. Malware may accomplish attack missions either individually or collaboratively [33, 34], perform attacks only once or periodically [10], and be triggered by the installation or a broadcast message. In addition, malware may adopt several mechanisms to bypass the detection of security analysts and antivirus software, such as ProGuard [37] and reflection [12]. All of these raised challenges for the existing detection approaches to reach a desirable precision and scalability simultaneously.[1]

On the other hand, it is challenging to eliminate greyware from malware [38], especially when they invoking APIs, which can request privileged permissions for accomplishing specific functionalities. For instance, WeChat, one of the top-ranked applications in Google Play, requests permissions of reading SMS messages and accessing network simultaneously. It may raise the concern of security analysts since it is speculated as

[1]The work in this chapter has been published in [93].

a potentially malicious behavior which sends SMS messages out to the network. However, the fact is that it only reads the SMS messages from its remote server for the two-factor authentication use. Similar cases are pervasive on Android: weather applications show the weather situation and forecast to users, and thereby, need to read and send out users' location information; social applications may ask for users' contacts to find friends quickly; fitness applications sometimes access the sensors in order to measure users' exercise. Therefore, the detection based on an imprecise and coarse-grained malicious behavior model would lead to a high false positive rate.

Even with a precise model of malicious behaviors, malware searching in applications with static approaches is not easy. New execution paradigm, system libraries and rich communication features provided by Android have facilitated the development of rich-functionality applications. On the other hand, however, they also make static analysis of application more complicated and difficult, which are summarized below.

- **Implicit Execution Sequence.** Android framework provides a variety of program execution environments, callbacks and control frameworks for each Android component[2]. It is known as *lifecycle*. For example, after an activity is started by the system, it will execute the methods `onCreate()`, `onStart()` and `onResume()` in proper order, which cannot be observed from the application code;

- **Various Triggers for an Application.** There are many ways for an application to interact with the external environment. The application can be triggered or impacted by users' GUI operations (e.g., clicking a button). It can register a broadcast receiver to respond once a broadcast message arrives. In addition, local sensors can drive the application to run in a pre-defined way. On the other hand, an application can be started and driven via remote messages, such as Google Cloud Messaging (GCM), HTTP response, and an incoming SMS or phone call;

- **Complicated Communication Mechanisms.** Although each application is running in a separated sandbox, Android provides them various ways to communicate with each other. For instance, the Intent model [33] is the most compelling method

[2]It is regarded as Template Method design pattern in Java, which is designed to solve a particular problem by specifying an execution sequence of a set of methods or classes.

for component communication. Additionally, applications can define bound services, for example, an AIDL (Android Interface Definition Language) interface, and implement a *Binder* or a *Messenger* to accomplish the communication even between different processes or applications.

To overcome the security challenges on PC/server, we propose an integrated framework called DroidEcho to analyze Android applications. First, we summarize the features of attacks happening on the Android platform, and propose a novel attack model. The model illustrates a variety of system API-based attack types at an abstract level, which is platform-independent. In particular, an attack is composed of: *assets*, which are the targets of attacks; *actions*, the execution operations with the API invocation performed on assets, and *triggers*, of which one entrance to the app that leads to the attack behaviors. Then we specialize the attack model into attack instances which are close to the Android platform, and can be utilized to guide our detection of attacks in a precise way.

Meanwhile, we transform Android applications into a comprehensive graph, incorporating call graphs between methods, and control flow graphs as per method. We conduct an in-depth static analysis through the graph with the guidance of attack model, and generate a full path with the trigger and the predicates that guarantee the occurrence of these behaviors. The detected malicious behaviors will be filtered by two conditions: if a seemingly malicious behavior is triggered by the user, it is likely that the behavior is user-intended, which we regard it as being harmless; presence of suspicious behaviors does not mean there is a real attack. It happens because some applications indeed need to carry out several seeming "malicious" behaviors to fulfill their tasks with good purposes. This is learnt and induced by investigating a group of applications under the same category or being similar. We make use of the mined social knowledge to filter out these harmless behaviors with a high level of confidence, i.e., these behaviors are likely a necessary part for applications. It does not only facilitate the efficiency of detection, but also reduce false positive in practice.

After the identification of malicious behaviors, we propose an approach to confirm the detected attacks with the dynamic execution. Our dynamic analysis is driven by the attack traces generated previously, and provides a satisfied condition to guarantee the

program to proceed along the trace. The dynamic execution reproduces the occurrence of attacks, and makes the attack detection more precise.

Different from the existing research on static analysis based approaches [24, 52–54], our work starts from the comprehension of Android malware by constructing semantic models. To reduce the false positive rate, we propose an approach to confirm attacks complying with the identified executed traces. To sum up, we make the following contributions:

- **Attack model.** We propose a novel representation, to characterize malicious behaviors. An attack in the model is constituted of target assets, execution actions, triggers, execution flows and apps' declaimers. It can facilitate the understanding of the essential features of attacks, and the detection of malware.

- **Accurate attack detection approach.** We propose a richly descriptive representation, named ICCG, to depict an Android application, with a maximal preservation of information. Based on ICCG, we design a synthetic approach to identify a malicious application by considering both the engineering aspect and the social aspect. A reduced but sufficient static analysis is to prove the presence of suspicious behaviors, then confirmed with the help of the learnt social knowledge.

- **Attack Confirmation.** After the identification of malicious behaviors, we conduct a confirmation process to prove the existence of a real attack with dynamic execution. The dynamic execution is fed with the traces of malicious behaviors generated by DroidEcho, and further identifies the satisfiable conditions. Then it drives the application to execute along the traces, and thereby reproduces the attacks for confirmation.

- **Evaluation.** We have evaluated DroidEcho on the malware benchmarks (i.e., Genome and DroidBench), and 7,643 real world applications. It shows that DroidEcho outperforms the state-of-the-art tool. Moreover, we have found out 444 applications with malicious behaviors in Google Play, and have a competitive edge in precision of 89.5% to the counterpart approaches and tools.

3.2 Semantic Model of Attack

In this section, we first give an in-depth discussion on the attacks happening on the Android platform, and then provide a formal description of these attacks.

3.2.1 Building Blocks

An attack on the Android platform has its unique features and characteristics. It has a variety of attack targets, and includes a sequence of actions that often leverage the APIs provided by Android. In order to depict these elements of an attack, we start with introducing the building elements of attacks and their representative examples on Android, in order to construct a general and formal definition of attacks.

3.2.1.1 Assets

Assets are referred to hardware, software and information on Android devices, which are the targets of attacks. For example, contact information is an important asset, which attackers aim to steal and make use of for malicious purposes; front light is a battery-consuming hardware such that some malicious applications may acquire it without releasing to exhaust battery quickly. On the Android platform, all the assets we concern about can be accessed by invoking certain system APIs. We list the representative examples of these assets on Android as follows.

- **Information Assets**: Identity code, Contact, SMS messages, File system, Location, System setting, etc.
- **Software Assets**: Phone service, SMS service, Package Manager, Download Manager, Broadcast service, etc.
- **Hardware Assets**: Camera, Media, Sensor, etc.

3.2.1.2 Actions

An attack action is an operation performing on a certain asset with the purpose of acquisition, tampering and interception, e.g., to fetch the IMEI code of the mobile phone.

Table 3.1: The category of actions on Android

Category	Operation	Action Example	Corresponding Implementation
Information-based	acquire	get SMS message	ContentResolver.query(Inbox)
	insert	insert a contact	ContentResolver.insert(Contact)
	edit	change system setting	Wallpaper.setBitmap(Image)
	delete	delete local files	File.delete()
Software-based	invoke	call a number	startActivity(Intent{tel:num})
	interrupt	block SMS messages	abortBroadcast()
	stop	uninstall an app	startActivity(Intent{pkg:app})
Hardware-based	occupy	hold the wakelock	WakeLock.aquire()
	release	release the wakelock	WakeLock.release()
Communication	e_send	send data to environment	sendTextMessage(SMS)
	e_recv	receive data from environment	getMessagesFromIntent(Intent)
	i_send	send data to other component	startService(Intent)
	i_recv	receive data from other component	getIntent(Intent)

Category According to the type of the target assets, actions can be categorized into several classes. For example, an action can acquire, edit, or delete some information stored on device; invoke, interrupt or stop a service provided by Android; and occupy or release a hardware resource. Therefore, the semantics of actions can be uniquely specified by the association of the action type and the target assets. In addition, there is a unique kind of actions on Android which are used for communication (see § 3.3 for more details). Within communication, there must be at least one sender and one receiver, and the communication can occur between an application and the external environment, or between two components in one application. As a result, we summarize four actions related to communication in the scope of application. Table 3.1 shows the categories of actions covered in this work.

Parametrization An action is often implemented by invoking a set of system APIs. These APIs are organized with a certain dependency relationship. For example, the action of retrieving data stored in a content provider can be described as: obtaining an instance of `ContentResolver`; specifying the URI of the target asset; and retrieving the data stored in this content provider. Every action of retrieving data in content provider follows the above processes. And we provide more details about this in § 3.4.3.1.

As a functional unit in the attack model, an action usually has an input, an output or both. Let α be an action, and β be an asset, then $\alpha(\beta)$ denotes the input of the action α is the asset β, and $\alpha \xrightarrow{\beta}$ denotes the output of the action α is the asset β (refer to § 3.2.1.4). A variety of concrete actions are derived from parameterizing these actions with assets. For instance, when acquiring the content of a content provider, we can

specify some assets as the target, such as `ContactsContract.Contacts.CONTENT_URI` and `CalendarContract.Events`. As a consequence, two actions are generated to fetch the contact list and events in the calendar, respectively. Table 3.1 list 9 basic kinds of actions, based on which more actions can be generated by parametrization with explicit target assets.

3.2.1.3 Triggers

Triggers are events which are taken as inputs to an application and lead to the occurrence of a behavior. Although triggers, which occur during runtime, are unpredictable for applications, the application can provide handlers to subscribe and capture these triggers. Once the application receives a subscribed trigger, it will go into the life cycle and execute specific methods. In light of the awareness of users, we present two sorts of triggers in the following:

- **User Interaction**. This kind of triggers are usually GUI-related, which are visible to the operating users. For example, when the user clicks a button drawn on the screen, the behavior is triggered and starts to execute. From this, the user can learn that the behavior is caused by his/her click operation, and we call it user-awareness. For simplicity, we assume that users can know the behaviors from the context which the user interaction causes.

- **Environmental Inputs**. There is another kind of triggers which can drive the execution of an Android application. The trigger could be the initialization of the application, a broadcast message or registered listeners to sensors. The whole process is free from the involvement of the user, which means that the user is likely unware of the execution of behaviors. As a consequence, we classify malicious behaviors triggered by environmental input as potential attacks for a further analysis.

As suggested by [94,95], behaviors that would never been executed until they are triggered by the user interaction reflect the "intention" of the user. Therefore, in this work, we assume that user interactions will not trigger any malicious behaviors, i.e., potential attacks that are triggered by user interactions are false positive. However, environmental

input triggers can proceed stealthily, preventing users from knowing them. This kind of triggers usually bring in many security risks, which are our main concern in this work.

Since triggers are external objects that cause the execution of attacks, we can instead recognize *e_recv* (see Table 3.1) to observe the arrivals of triggers. Specifically, the listeners can be categorized in terms of types of triggers. For example, `onClick(View)`, `onDrag(View,...)` and `onKey(View,...)` are the entry points of program when a user interaction trigger comes. While `onCreate()` and `onReceive()` are the entry points for the boot of applications and a broadcast message, respectively, which are regarded as environmental inputs.

3.2.1.4 Flows

Actions have a flow relationship in between. It is a kind of dependency relationship which is either directional or contextual. The directional relationship indicates the certain order of execution, which has been defined in the program logic for a specific task[3]; and the contextual relationship can be described as a semantic connection between two actions, for example, the input of an action is the output of the other action[4]. Generally, the contextual relationship needs a transition of the negotiated data from one participant to the other.

A flow can exist between the environment and an action, and triggers are their negotiated data between them. Take an incoming SMS message for example, if an application registers a *BroadcastReceiver* for SMS messages, once an incoming SMS message arrives, the application will start to execute from the listener, and it can also get the content of the message as input. Therefore, there exists a directional and contextual relationship between the environment and the action $acquire(SMS)$, i.e.,

$$\overset{SMS}{\dashrightarrow} \; e_recv$$

A flow can also exist between two actions. After an application gets an incoming SMS message, it can send the message to a remote server via the Internet. In such a case,

[3]From the aspect of program analysis, it is control flow, which defines the execution order of statements.

[4]For the aspect of program analysis, it is data flow, which depicts a flow between two variables.

it is a contextual flow between these two actions. The flow guarantees the two actions perform on the same SMS message. Therefore, we present the flow as:

$$e_recv \overset{\text{SMS}}{\dashrightarrow} e_send \overset{\text{SMS}}{\dashrightarrow}$$

3.2.2 Attack Models

Based on the aforementioned building blocks for an attack, we define different attacks in this section. In the remainder of this section, we use the following notations. E is the set of Environmental Input triggers; t is the trigger of the attack and $t \in E$; $Asset$ is the set of assets involved in the attack. Let α be an action or a trigger, β be an action, and γ be an asset. A flow is either a control flow denoted as $\alpha \rightarrow \beta$, or a data flow denoted as $\alpha \overset{\gamma}{\dashrightarrow} \beta$.

3.2.2.1 Attack Taxonomy

We conduct a comprehensive investigation of existing attacks of malicious behaviors [10, 12, 96–98], and propose a taxonomy of attacks in terms of these building blocks and semantic information as follows.

Privacy Leakage. Privacy leakage [79, 88, 99] refers to the exposure of sensitive information on devices. As discussed in the action part, such kind of information can be acquired by specifying an *acquire* action, which is regarded as *source* in the attack of privacy leakage. If there exists a data flow from the return value of the acquire action to the data sent out to the external environment by a communication action, usually called *sink*, privacy leakage happens. In addition, the attack needs to happen without users' awareness, and it is not necessary for the trigger to have a dataflow relationship with these two actions. As a result, the formal attack model of privacy leakage can be defined as:

$$PL =_{\rightarrow}^{t} e_recv \rightarrow acquire \overset{\gamma}{\dashrightarrow} e_send(\gamma) \overset{\gamma}{\dashrightarrow}$$

Information Interception. Mobile devices can interact with the external environment in many ways. However, malicious applications intercept the communication, suspend, or even break off the communication. The common attacks include blocking an incoming SMS messages and phone calls. For such kind of attacks, malicious applications

need to register a listener (i.e., *e_recv*) for broadcast messages of incoming messages and calls, which stops the spreading (i.e.,*intercept*) to avoid the messages from reaching to other applications or the user.

$$II =\overset{t}{\dashrightarrow} e_recv \rightarrow intercept(\gamma)$$

Content Tampering. Malicious applications may tamper content on mobile devices, such as contact, SMS, account, and system settings. It can cause severe damages to the user. Usually, an application can insert, update and delete an item in a content provider with specific permissions. In addition, it can change system settings such as network connection, wallpaper and sleep time. We use *insert*, *edit* and *delete* to describe such kind of behaviors. The trigger of this attack will not give rise to users' attention and does not have any data flow relationship with these actions. The attack is defined as follows:

$$CT =\overset{t}{\rightarrow} e_recv \rightarrow \alpha(\gamma),\ where\ \alpha \in \{insert, edit, delete\}.$$

Service Abuse. Malicious applications may abuse the services provided by Android [100]. According to our investigation, the most prevailing services which are abused include *phone service*, *SMS service*, *package manager*, and *download manager*. For example, if an application possesses the permission of sending SMS messages, it can subscribe a premium-rate mobile service which causes users' financial charge. Let α be the kind of actions which abuses services, and the attack model can be presented as:

$$SA =\overset{t}{\rightarrow} e_recv \rightarrow \alpha(\gamma),\ where\ \alpha \in \{invoke, stop\}.$$

Resource Depletion. Due to portability and simplicity, mobile devices usually carry low-frequency CPU, RAM of limited size and small capacity battery. Mobile devices thereby can only provide a limited computation capability, storage and energy. It would make worse if any installed applications occupy these resources immoderately, which can influence other applications, and even the battery life of the device. Either intentionally or unintentionally, applications keep consuming resources [91, 92, 101] or carry on useless and endless works [102], while never release or stop them. Let $occupy$ be the kind of actions which exhausts resources, and $release$ be the kind of actions which releases resources. And we use $\nrightarrow$ to show a missing flow between these two actions. The attack model is given in the following:

$$RD =\overset{t}{\rightarrow} e_recv \rightarrow occupy(\gamma) \nrightarrow release(\gamma)$$

Table 3.2: The category of attacks on Android

Attack	Percent	Supported by DroidEcho
Privacy Leakage	31.4	✓
Information Interception	11.6	✓
Content Tampering	13.4	✓
Service Abuse	31.4	✓
Resource Depletion	1.8	✓
Fishing	1.7	✗
Adware	2.3	✗
Privilege Escalation	6.4	✗

3.2.2.2 Discussion

The taxonomy of attacks is based on the 102 malware families we have studied. However, there are some attacks out of detection of our approach, such as *fishing*, *adware* and *privilege escalation*. Fishing is a kind of attacks in which one application disguises an authentic and legitimate application, and induces users to enter their credentials of, for example, bank account [103]. Adware is a program that displays advertisements to its users, which is annoying rather than harmful at most of time [104]. Some applications may exploit the vulnerabilities of Android, such as Exploid, RATC/Zimperlich and Ginger Break [11], to elevate the privilege once installed on device; *Pilup* [89] is a newfound flaw in Package Management Service which can be exploited by malicious applications only during the phase of upgrading the Android OS. At last, *side channel* attacks [17, 105, 106], which collect memory information or timing information, are not our scope of attack detection.

The insufficiency of DroidEcho comes from two aspects: 1) our static analysis is carried on Java code, and does not go inside the native code. Many of malware of privilege escalation utilize native code to elevate the privilege; 2) we try to avoid to make a subjective judgement, but prefer to detect an objective existence of malicious behaviors. That is, fishing and adware just deceive and bother users respectively, which do not violate security policies [96] of Android precisely. We give the statistics of attacks mentioned previously in Table 3.2, indicating that our approach can detect up to 90.4% of attacks in theory.

3.2.2.3 Disclaimers

There is a significant exception for determining an attack - disclaimers. A disclaimer is a white list for an application in which some behaviors are excluded from consideration for the determination of attacks. The violation of certain security properties cannot imply the occurrence of attacks. Some applications may need to carry on some suspicious looking behaviors which they already claimed the potential security violation explicitly. We conclude that the users who install their applications would like to undertake the introduced risks by default. Therefore, in this work, we filter out the "attacks" that are allowed by the users, and remove them from the generated attack report.

3.3 The Inter-component Communication Graph

For an accurate representation of Android applications and the convenience of attack detection, this section presents the proposed the inter-component communication graph (ICCG) to capture all possible communications between components and threads inside Android applications.

3.3.1 Android Communication Medium

Medium is a special data structure used for communications. The communications can occur between either two components (i.e., activity, service, broadcast receiver and content provider), or two isolated processes. Medium is playing a critical role in the behavior of Android applications. Besides the frequently-talked *Inter-Component Communication*(ICC) [17,107], which is based on the Intent medium, there are three other mediums which can be also used during the communication. Here we provide the different types of mediums existing on the Android platform.

Intent. Intents are the main vehicle for communication. One intent can be either explicit or implicit. Explicit intents have a specific class to start, while implicit intents do not specify the corresponding class, and the system will select the most well-suited class or application to execute. An explicit intent can only invoke a specific component, which is defined in the constructor, or by calling `setComponent(ComponentName)` or

`setClass(Context, Class)`; an implicit intent can be received by many well-suited components. It appoints potential receivers by setting an action in the constructor or `setAction(String)` (Meanwhile, it can be instrumented with a data type to restrict its receivers) [108]. Intent can influence the execution order (a.k.a., control flow) of the application, and also impact on the data flow if enclosed with *extras*.

Message. *Message* is a concise data structure for arbitrary data. Two isolated processes or threads can communicate with each other by transferring a message. In general, the message receiver has to create a *Messenger* to handle the received messages. On the sender side, it needs to obtain the reference to this Messenger, and sends its crafted message by invoking `send(Message message)` of the *Messenger*. In order to send a message, for example, to a daemon service, the component can first bind to this service via `bindService()`, and then fetch the reference to the *Messenger* from the returned *Binder* object. **Binder**. Binder is used for a component to talk to a daemon service. The component which attempts to bind to a service needs to invoke `bindService()` and implement `ServiceConnection`, which establishes the connection with the service. On the service side, it needs to provide an inherited class of Binder, exposing public methods to customers; or design an AIDL interface as well as the implementation. After that, the component can obtain a binder object, which is a remotable object for a lightweight remote procedure call. In addition, AIDL can be exposed to other applications for remote invocations.

Persistent Storage. On Android, applications may exchange data through persistent storage. There are three types of persistent storage: *File, Shared Preferences and SQLite database*. They can be used for applications or components to exchange data, that is, they provide an implicit data flow from one component to another.

3.3.2 Inter-component Communication Graph

Definition 1 *Let M be the communication mediums existing on Android. An ICCG is a directed graph defined as $G = \{V, E_f, E_c\}$,where V is a set of nodes; $E_f : V \times V$ is a set of flow edges; and $E_c : V \times M \times V$ is a set of communication edges.*

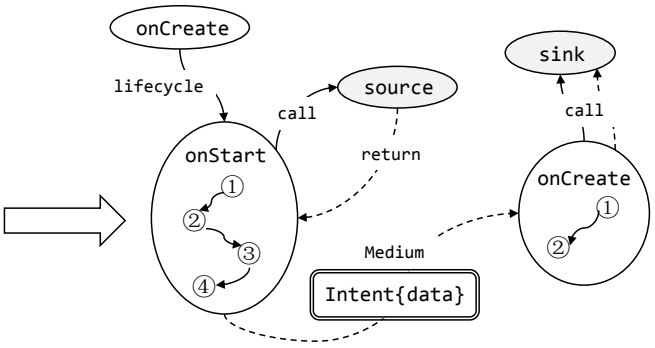
The nodes of a graph are the methods contained in the application, which come with two levels of granularity. The coarse-grained nodes only represent the signature of the

```
Activity:

onCreate(){}
onStart(){
    ① String data = source();
    ② Intent intent = (Service.class);
    ③ intent.putExtras(data);
    ④ startService(Intent);
}
-----------------------------------
Service:

onCreate(){
    ① String data = getIntent().getData();
    ② HttpClient.send(data);
}
```

(a) The snippet code of malicious behavior



(b) The corresponding ICCG of the code

Fig. 3.1: An example of malicious behaviors and the corresponding ICCG

functions, and help to express the relationship between functions in the system level. We can learn the method invocation relationship and possible communications between different functions. In the fine-grained level of granularity, a node is in-depth dissected and shows the internal logic, i.e., control flow. When we are identifying the elements of attacks, especially behaviors, we need to go in deep at the code level, and recognize the different patterns of behaviors.

We employ two different kinds of edges to denote the relationship between nodes - *call* relationship and *communication* relationship. Flow edges reflect the call relationship among nodes. This is the primary concept in the program analysis, which consists of *explicit calls* and *implicit calls*. Here we emphasize the unique implicit calls, i.e., *Android Lifecycle*, existing on Android. An Android lifecycle indicates an implicit function invocation between different methods or classes. The implicit calls are either callbacks passed to a concrete method, or control frameworks specifying a call sequence. Besides the lifecycle features of standard Java, e.g., the method `void start()` of one thread instance will implicitly call the override method `void run()`, Android has included many libraries to support an amount of implicit calls. For each component of Android, it has a unique call sequence pre-defined by Android. In addition, all GUI components on Android allow developers to pass a callback to execute functionalities when the corresponding event occurs.

The communication edges are connecting between nodes and mediums. As defined previously, there are four kinds of mediums used for communication, and it is worth

mentioning that the communications are not only showing the logic order of execution, some of them also enclose data which can be transferred from one node to another node.

We use the DroidKungFu malware[5] as an example to explain the ICCG. As shown in Figure 3.1 (a), there are an activity and a service, which communicate via an Intent medium. The activity obtains sensitive data (refer to ① in `onStart`), and passes the data to the service. Then the service sends the data out at ② in `onCreate`. Figure 3.1 (b) shows the constructed ICCG based on the code. As discussed in the previous section, each node represents a method of the application, and contains a control flow graph. The nodes are connected by two kinds of edges: Android mediums (e.g,. the Intent object) and method invocations either implicit invocations (e.g., lifecycle) or explicit invocations.

3.3.3 Sufficiency of ICCG

We construct ICCG for representing the overall structure of functions in the application, and search if any attack model is hidden in the graph. As the attack model proposed in § 3.2 is general and platform-independent, we show the sufficiency of ICCG to detect attacks below.

As modeled in § 3.2, an attack is a set of operations which the attacker performs to achieve a certain objective, and it is composed of 5 essential elements. ICCG retains almost all program information, and we can extract a number of call sequences from it. By checking each call sequence, we can recognize actions which are attack related, identify the trigger of it, and perform data flow analysis on the call sequence. Hence, we could find a mapping from the attach model to the ICCG, which means that ICCG contains sufficient information to detect an attack inside.

3.4 System Design of DroidEcho

This section presents the design of DroidEcho. As shown in Figure 3.2, DroidEcho takes as input an Android application, which contains the class files, the manifest file and the description of its functionality. DroidEcho will generate an attack report which contains identified malicious behaviors and the corresponding traces of these behaviors

[5]http://www.f-secure.com/v-descs/trojan_android_droidkungfu_c.shtml

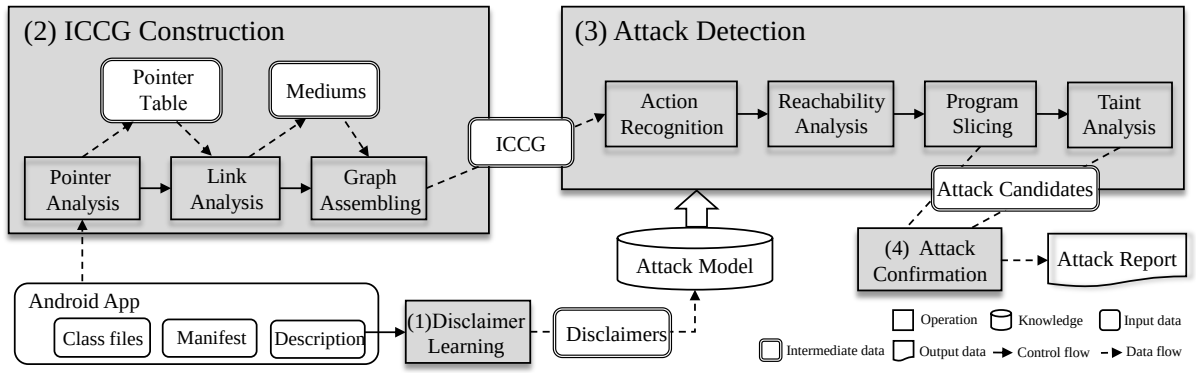


Fig. 3.2: The architecture of the system

for forensic use. DroidEcho leverages the *attack model* which is presented in § 3.2 as the guidance for attack detection, and proceeds in four phases: *disclaimer learning*, *ICCG construction*, *attack detection* and *attack confirmation*. The first phase *disclaimer learning* receives the descriptive text of applications as input, and generates a white list of "necessary" behaviors (a.k.a., disclaimer of the application) in a supervised manner. The white list will be used to exclude the detection for the claimed functionality of the application. Second, *ICCG construction* takes class files and the manifest file of the application as input, and constructs an ICCG, which is then passed to the third phase. *Attack detection* can find out, if any, existing attacks and the corresponding traces which cause these attacks in the application. At last, *attack confirmation* receives the candidate attacks, and determines whether one attack candidate is a false positive or not by a trace-guided dynamic execution.

3.4.1 Disclaimer Learning

Some Android applications may perform seemingly suspicious behaviors while they are actually demanded to accomplish the functionality. The demanded functionality and the risks it may bring are usually claimed in their descriptive text. We regard this as a benign behavior (henceforth disclaimer), and it will not be considered as an attack candidate. For example, *TripAdvisor* is a travel application, which can provide the nearby restaurants and hotels when the user is travelling. For ease of use, it acquires the permission *FINE_LOCATION* to learn the user's location such that it can provide the

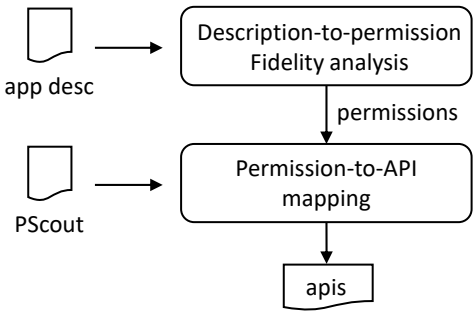


Fig. 3.3: The learning process of disclaimers

most suitable information for the customers. Although we detect that TripAdisor has a privacy issue, which sends the user's location to a remote server from time to time, we regard this as being benign and harmless.

As shown in Figure 3.3, we obtain the descriptions of applications and perform a description-to-permission fidelity analysis [109]. The fidelity analysis builds a description-to-permission relatedness model in which one permission is associated with a list of noun phrases. For the description of a given application, we can leverage this model to produce a list of requested permissions. Then, we employ PScout [110] to elicit the corresponding APIs that request permissions. For example, the sentence "Your location: These permissions are needed to obtain your location so we can help you discover hotels, restaurants, and attractions around you" in app TripAdvisor implies that it requests for recognizing users' current location the permission android.permission.ACCESS_COARSE_LOCATION and android.permission.ACCESS_FINE_LOCATION. Therefore, 21 Android APIs (e.g., void requestLocationUpdates(float, LocationListener) and Location getLastKnownLocation(String)) are regarded as being necessary to invoke by permission-to-api mapping.

The produced Android APIs serve as disclaimers to refine the attack model. During attack detection (see Section 3.4.3), these APIs will not be considered as attack actions.

3.4.2 ICCG Construction

The construction of ICCG takes class files and the manifest file of the application to be checked as inputs. Primarily, DroidEcho employs *Soot* [111] to generate a rough call graph of the whole application and a control flow graph for each method. Given that, DroidEcho proceeds in three steps successively: *pointer analysis*, *link analysis* and

graph assembling. The first two steps can provide all auxiliary information to assemble an ICCG.

3.4.2.1 Pointer Analysis

Pointer analysis is a static analysis to infer which variables are pointed to by pointer references or heap references. In this step, we want to identify all references which are pointing to variables in the application, and all possible values which the variables can be assigned to. The result of this step is a *PointerTable*, which contains mappings from variables to concrete values: $\mathbf{Set}(variables) \rightarrow \mathbf{Set}(values)$. $\mathbf{Set}(variables)$ denotes a set of variables which are pointed to with the same reference at a time, and $\mathbf{Set}(values)$ denotes a set of possible values to which the variables can be assigned. ***PointerTable*** plays a critical role in the step of *link analysis* and *action recognition*. During the step link analysis, ***PointerTable*** is used to infer the actions and classes of an Intent object, thereby DroidEcho can identify which components are able to receive this Intent. And DroidEcho needs the ***PointerTable*** to recognize the semantics of actions during the action recognition. For example, when DroidEcho encounters an operation to query a content provider, it needs to learn the value of the argument URI, to distinguish different content providers.

Parts of our pointer analysis are based on SPARK [112], which is a pointer analysis framework. It can cluster the variables into several sets, i.e., $\mathbf{Set}(variables)$, where all variables in the same set have been pointed to with same reference at a time. Since we have got a rough call graph and control flow graphs of all methods, we traverse the call graph and go inside control flow graphs to perform value inference. We evaluate each node in a control flow graph, and infer the possible values of the variables. The value inference can handle basic arithmetic and ***String*** operations. In addition, we do not evaluate all types of variables, which are both computation expensive and useless to our attack detection. We only pay attention to the valuation of primary types (e.g., ***boolean***, ***int***, ***double***), ***String***, ***ComponentName***, URI/URL and Intent. It is worth mentioning that the values of ***ComponentName*** and URI/URL objects can be expressed by a ***String***, while we construct a more complicated structure for Intent objects, which basically contains four fields: action, class, data and category.

The pointer analysis used in this work is type-sensitive, however, flow-insensitive. That is, every variable in the same set needs to share the same data type with others. In order to reduce the expense of storage and computation, we store all possible values which the set of variables can be assigned to rather than only parts of them after a certain statement.

3.4.2.2 Link Analysis

Link analysis is to establish all links between methods or components in an application, i.e., the edges in ICCG. Primarily, the call graph generated by Soot only contains the call relationship between Java methods. As introduced in § 3.3, there are implicit invocations and a variety of communication mechanisms on Android. On the basis of the call graph, we analyze all links between methods and build a complete communication graph for the application.

There are two kinds of links between two methods, invocation links (either explicit or implicit) and communication links via Android medium (e.g., Intent and message). We first build call chains for the lifecycle of Android components. For example, one of the call chains of Android Activity is `onCreate` → `onStart` → `onResume`, which shows the implicit invocations after the start of the Activity. As a result, the above methods in the call graph will be linked with an invocation edge, respectively. For communication links, we recognize the mediums as well as their attributes existing in the methods, and identify which components or methods can receive these mediums. Take the Intent medium as an example, if we find an action which starts activities, like `startActivity(Intent)`, we retrieve the attributes (e.g., class and action) of the Intent object and identify which activities can be triggered by this Intent object. As a result, we add a new link between the method which sends out the Intent and the constructor method of the target activities.

3.4.2.3 Graph Assembling

By far, we have obtained the control flow graph for each method of the application, and all links between these methods. We take the control flow graphs as nodes, the links as edges, and assemble them into an ICCG. The graph depicts the execution order and communications between different methods at the system level, and illustrates the control

flow at the method level. Combined with *PointerTable*, ICCG is passed to the attack detection phase. Attack detection will search the graph and find out any existing attack.

3.4.3 Attack Detection

To reduce the search space of attack detection, we will not analyze the program from its entry points. In converse, we first recognize attack-related actions existing in the program in a fast way, and perform a bidirectional flow analysis from behaviors, which can effectively speedup the search process.

Algorithm 1 shows the whole process to check whether one attack is contained by the application or not. The algorithm takes ICCG of an application, and one attack model as the input, and outputs whether the attack model exists in the ICCG. Line 1-3 show that it recognizes all actions existing in the ICCG. If any of actions in the attack is not contained in the ICCG, DroidEcho concludes that the application does not contain this attack. In our implementation, we conduct an one-time retrieval of the ICCG for each application and store all recognized actions. By comparing the included actions in each attack, we can quickly eliminate some attacks which will definitely not happen.

If all actions in the attack model are found in ICCG, we proceed the reachability analysis and program slicing. Since there are two kinds of flows (referred to control flow and data flow in program analysis, respectively) defined in our attack model, we carry on *ForwardControlFlowAnalysis* (Line 10) and *TaintAnalysis* (Line 6) to determine whether the flows are satisfied or not. At last, we get the trigger causing this attack (Line 13), and check if it is a kind of *environmental input*, e.g., the initialization of application, system broadcast message and a timer task. In the following, we will give a more detailed description for each step.

3.4.3.1 Action Recognition

We use actions to describe the basic elements in an attack, which is semantic but domain-independent. However, we need to define a system of notations in a specific domain (here Android), to capture these actions and triggers in ICCG. On Android, we recognize an action by the corresponding constraints. Here we define three kinds of predicates to express APIs and constraints in these actions we met in the code: **sig**(*api*), **type**(*arg*),

Algorithm 1: Model-based attack detection

```
Input: ICCG of the application
Input: Attack model {(action_i, action_{i+1}, data|control)}, where 0 ≤ i < n − 1
Output: if ICCG contains attack
for action ∈ attack do
    if !(ICCG contains action) then
        return false;
for i = 0 to #actions − 2 do
    if flow(action_i, action_{i+1}) == data then
        data_flow = TaintAnalysis(action_i, action_{i+1}, asset);
        if data_flow is not satisfied then
            return false;
    if flow(action_i, action_{i+1}) == control then
        control_flow = ForwardControlFlowAnalysis(action_i, action_{i+1});
        if control_flow is not satisfied then
            return false;
trigger := BackwardControlFlowAnalysis(action_0);
if trigger ∈ {Environmental Input} then
    return true;
else
    return false;
```

and **value**(arg), where api is an Android API, arg is a variable, and these predicates will return a comparable constant value. As a consequence, action recognition can be transformed into a satisfiability problem,

$$action \models \mathbf{sig}(api) \tag{3.1}$$

$$\mathbf{sig}(api) \models \mathbf{type}(arg) \cap \mathbf{value}(arg) \tag{3.2}$$

One action is recognized if we detect some APIs which satisfies the above constraints progressively. Equation 3.1 shows the action can be recognized with an API with the specific signature, and moreover, the arguments or the base, if any, need to satisfy two kinds of predicates, **type** and **value**. As shown in Equation 3.2, arg is either the base of the API (static methods do not have a base), or the arguments. Specially, arg may be another invocation of API, i.e., **sig**. Therefore, we will recursively solve the constraints until the action is recognized. Taking the example of obtaining contacts, the essential code at language level of this action can be described as follows:

$$\frac{sig(api) = obj.query(uri, *)}{obtain\ contact} \tag{3.3}$$

$$\frac{\begin{array}{c} type(obj) = ContentResolver, \\ type(uri) = Uri, \\ value(uri) = \text{“}content://contacts\text{”} \end{array}}{sig(api) = obj.query(uri, *)} \tag{3.4}$$

As shown in Equation 3.3, we first need to find a pivotal function whose signature matches `obj.query(uri, *)`, and the methods need to meet three constraints: the base of the invocation *obj* needs to be an object of the class `android.content.ContentResolver`, the type of *uri* needs to be an object of `android.net.Uri`, and its value needs to be `content://contacts` as shown in Equation 3.4. The code statements, which together form a behavior, might have dependency relationship or follow an execution order in between. We deal with it as a constraint satisfaction problem, and recognize a behavior with reasoning. The benefits are that we do not need to care about the execution order of code in a behavior, and hence our approach is more general so as to identify more variations.

3.4.3.2 Reachability Analysis & Slicing

If the ICCG contains all necessary elements for one attack, we start to do program slicing from these elements. The slicing consists of backward and forward control flow analysis. The backward control flow analysis aims to complete three tasks: 1) find the root cause that lead to such action, i.e., its entry points. Based on the entry points, we can infer the type of the triggers. Then we know whether the attack is triggered by a user interaction or environmental inputs; 2) obtain all conditions in a trace from the entry points to the action. The conditions are used in *attack confirmation* to guide the dynamic execution of the application; 3) identify the search space for potential taint analysis.

The forward control flow analysis aims to complete two tasks: 1) determine the occurrence of the subsequent actions in an attack model; 2) similar to the backward control flow analysis, identify the search space for the taint analysis. As a result, we will not search the entire ICCG during the taint analysis, which is computationally expensive.

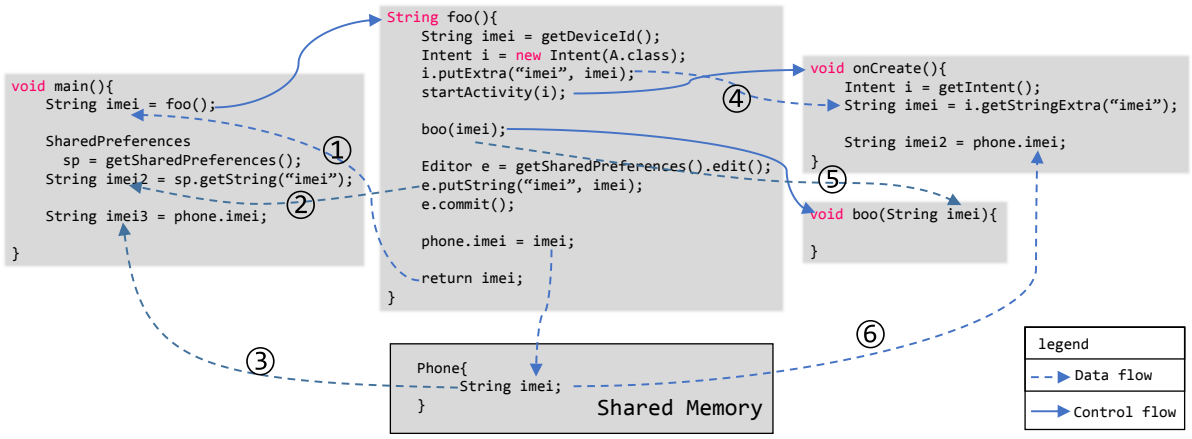


Fig. 3.4: Taint analysis across multiple domains

3.4.3.3 Taint Analysis

Taint analysis can track the flow of data during detection. Taking privacy leakage as an example, we need to carry on taint analysis to track the flow of data, and if the data is flowed to a sink action and sent out eventually. During the taint analysis, we get a domain set in a control-flow order $SearchDomain = D_1 \rightarrow D_2, ... \rightarrow Dn$, and the source action is located at D_{sr} after the above steps. Then we perform a forward data flow analysis on the domain set $SearchDomain$. Figure 3.4 illustrates the ways how the data can be tainted cross domains. First of all, data in the domain D_s can influence the data in its previous domain by three methods: return the data at the call site in the previous domains, referring to ①; the data flow ② shows how the data in the latter domain influences the data in its previous domains; and we can assign the data to one commonly shared variable between the domain D_s as shown in ③. There are three possible ways for the data in domain D_s to influence the data in the successive domains: enclose communication medium with data and pass it to the next domains as shown by the data flow ④; pass the data as an argument to its successive domains, which are used in these domains, referring to ⑤; assign the data to a commonly shared variable in between as shown by the data flow ⑥. In addition, we take a coarse-grained aliasing analysis in this work, i.e., if for example a string variable is passed to a function, and this function will encrypt the string and return a new encrypted value with a cryptographic scheme. Although we do not know how to convert the original string to the encrypted

one (we do not infer the meaning of cryptographic schemes), we can definitely ensure the operation is reversible, and the returned data is also of sensitive information.

3.4.4 Dynamic Attack Confirmation

As discussed before, DroidEcho's ICCG construction and attack detection are based on static program analysis, which is less precise than dynamic analysis. As a result, the attacks reported by DroidEcho may be false positives. Therefore, we introduce a confirmation step to reduce false positives, and the attack confirmation is based on the technique of dynamic testing.

An attack candidate, which is passed from the *attack detection* phase to the *attack confirmation* phase, contains an attack trace and the conditions that guarantee the occurrence of attacks. Given that, we simulate the inputs to drive the dynamic execution of the application and check whether the attack trace can occur in the real execution. In order to activate the attack candidate and capture malicious behaviors, we first instrument Android OS by hooking specific Android APIs which are included in our attack model, and then generate the triggers which are used to activate the contained malicious behaviors.

- **Instrumentation.** Since the actions in attack model are recognized as the invocations of specific Android APIs, we instrument Android OS to monitor the invocation behaviors. In this work, we leverage TaintDroid [79] to determine whether these APIs are invoked.

- **Triggers.** We leverage IntelliDroid [26] to generate all triggers leading to specific malicious behaviors, and subsequently schedule these triggers to drive the execution of the application. We simply feeds the application with all possible trigger sequences, and in order to eliminate the impossible sequences (which never occur during the real executions), we exploit the "*happen-before*" relations among these triggers to generate sequences.

Obtaining these inputs, DroidEcho is able to execute the suspicious applications to determine if the attack is reachable. In order to make the exploration faster, DroidEcho prunes the paths which rarely lead to the attack trace, which can significantly reduce the search space of the program.

3.5 Evaluation

We implement an automatic platform DroidEcho to facilitate the detection, accordingly. DroidEcho is written with 17,038 lines of Java, and 163 lines of scripts (Python and Shell). The dynamic confirmation is implemented based on TaintDroid [79] and IntelliDroid [26]. TaintDroid enables us to track the information flows of applications. In addition, we customized TaintDroid for two purposes. First, we intercept the APIs in our *Action* set to monitor whether they are invoked by the tested applications. Second, we intercept the APIs providing the applications with environmental inputs, such as location and time information, where we can return the applications values that would activate the target behaviors. During the confirmation, we employ IntelliDroid to generate the call paths for specific Android APIs as well as conditions that enable the paths. Then the driver script takes them as input to automatically drive the execution of the suspicious applications. To estimate the overall performance of DroidEcho, we conduct the experiments from three aspects: *evaluation on malware benchmark*, *evaluation on real apps* and *evaluation on performance*.

3.5.1 Evaluation on Malware Benchmark

To evaluate the performance of our approach on the infamous malware, we conduct an experiment on 1260 samples of malware of the collection [12]. According to the types of malware, we filter out 108 of them (e.g., Asroot, DroidCoupon and DroidDeluxe) which only use native code to launch attacks. At last, we successfully detect 940 (89.5%) samples, and also show the attack type. There are mainly two reasons for the missing malware: 1) some malware use reflection to dynamically invoke malicious code. For example, AnserverBot loads an executable file in its asset folder, retrieves the included classes and runs the code. 2) some of them leverage complicated obfuscation and encryption to confuse AV tools. For example, Geinimi leverages several cryptographic schemes (e.g., DES) to encrypt the communication and strings.

In addition, we conduct an experiment to compare DroidEcho's capability of attack detection with FlowDroid [24], which is a static tool in detecting privacy leakage. The

Table 3.3: Comparison with FlowDroid

App	DroidEcho	FlowDroid
ArrayAccess-1&2	TP	FP
HashMapAccess1	TP	FP
ListAccess1	TP	FP
Ordering1	TP	FP
Unregister1	TP	FP
Exception-1&4	TP	FP
PrivateDataLeak-1&2	FN	TP
ImplicitFlow-1&2&3&4	FN	FN
Reflection-3&4	FN	FN

subjects of this experiments include a set of open-source Android applications named DroidBench[6], of which the applications may contain the attacks of privacy leakage.

DroidEcho successfully detects 34 samples of malware, while fails to find 8 malicious samples. We provide Table 3.3 to illustrate the comparison results, actually only the different results, with FlowDroid. As shown in Table 3.3, DroidEcho has an edge in detecting the first six kinds of privacy leakage, but cannot detect the last three kind of privacy leakage. PrivateDataLeak-1&2 are two applications which steal the text in a password field of an Android GUI view. Since the data on GUI components are hard to be determined to be sensitive, in addition, applications which need authentication have to send credentials, such as user input from keyboard, to the remote server for authentication. As a result, DroidEcho does not track the flow of the data on GUI components. And last, DroidEcho and FlowDroid both cannot cope with the last two kinds of applications, where ImplicitFlows are samples which leverage obfuscation techniques to confuse the analysis, and Reflections are two samples which use reflection to dynamically invoke methods or fetch fields to complete the process of privacy leakage.

3.5.2 Evaluation on Real Applications

We have collected 7,643 applications from Google Play, which are hot and free application in their respective categories. By running DroidEcho, we find out 444 applications which have malicious behaviors. In addition, we have done a statistics of

[6]http://sseblog.ec-spride.de/tools/droidbench/

behaviors which are user-awared or already claimed by the description of applications. We compare DroidEcho with other anti-virus (AV) tools, by uploading *apk* files into VirusTotal (www.virustotal.com). Although AV tools have detected 1,541 (20.2%) samples of malware, most of them are Adware, of which the number is up to 1,217 (79.0%). Due to the restriction of our approach, we do not provide a detection for Adware. By filtering these applications of Adware, we can also find 149 more applications which have malicious behaviors.

We investigate the 149 applications which contain malicious behaviors, of which 131 applications have privacy leakages, while the remaining applications have other four kinds of malicious behaviors. In particular, 10 applications contain service abuse attacks, i.e., sending SMS messages without users' consent; 6 applications contain content tampering attacks, i.e., deleting SMS messages from the inbox; 2 applications are depleting battery by holding Screen lock for a long time. By investigating the code of these applications, we find that many of them are employing a third-party library which has exposed sensitive information. The third-party libraries may do a measurement for the usage of applications, e.g., Flurry and Crittercism, diagnose the crash of applications, e.g., Crashlytics, or advertise, e.g, Umeng and Google Ads. Table 3.4 shows third-party libraries that are contained in the applications.

False positive analysis. To evaluate DroidEcho's accuracy, we randomly selected 50 samples, and manually identified 4 false positives. Two false positives are because DroidEcho cannot well handle collection objects such as array, list, and map. If any element in a collection is tainted, DroidEcho determines the whole collection object is tainted. One false positive is due to the ignorance of execution conditions of flows. The execution condition may not be satisfied during runtime leading the malicious behaviors cannot be practically triggered. The last false positive is attributed to the insufficient modelling of persistent storage. As an alternative communication channel, persistent storage (e.g., file, database) might contains multiple dimensional data. It is non-trivial to track the flow of data in the persistent storage, which will be further studies in future.

Table 3.4: Privacy leakage via 3rd-libraries

Library	Description	Num	Behaviors
Adobe	Measurement of Usage	1	Identity Code, etc.
Flurry	Measurement of Usage	20	Identity Code, Location, etc.
Conversant	Measurement of Usage	1	Identity Code, Location, etc.
Crashlytics	Diagnosis of Crash	8	Identity Code, Sys. Info, etc.
Map Service	Map Service	5	Location, etc.
Crittercism	Optimization Tool	1	Identify Code, etc.
Umeng	Advertisement	4	Identity Code, Location, etc.
Google Ads	Advertisement	3	Identity Code, Location, etc.
Amazon Ads	Advertisement	1	Identity Code, Locatoin, etc.
Millennialmedia	Advertisement	2	Identity Code, Location, etc.

3.5.3 Evaluation on Performance

In order to evaluate the efficiency and scalability of DroidEcho, we measured runtime parameters in the previous experiments. The runtime parameters consist of the complexity of applications and runtime for each phase of DroidEcho. And the experiments are conducted on a Linux Ubuntu 14.04 machine, carrying 12 cores of Intel Xeon(R) CPU E5-16500, and 16G Memory. We depict the complexity of applications from four aspects: the file size of application, the number of nodes, edges and mediums of the ICCG. We have measured the runtime for pointer analysis, link analysis, action recognition and attack detection, respectively. The detailed data can be found in Table 3.5. As shown in Column *Runtime(ms) of DroidEcho*, DroidEcho is very effective in detecting attacks, with the average time of about 35s to complete the analysis of a real application. In addition, since we leverage Soot to generate the rough call graph and control flow graphs for each method of applications, the runtime of Soot should also be considered to complete the whole detection. Soot performs a heavy work of reverse engineering, i.e., converting Android *.dex* code into Java bytecode, the time spent on that is hence much larger than the runtime of DroidEcho. As a result, due to our optimized approach, like pruning the paths which have lower possibility to trigger attacks, the performance of DroidEcho is much better than Soot.

3.6 Chapter Summary

In this work, we introduce a novel attack model to depict the essential characteristics and features. In addition, we build a transformation from an Android application to

Table 3.5: Evaluation on Performance of DroidEcho

	Size (K)	ICCG			Runtime(ms) of DroidEcho						Runtime(ms) of Soot
		#N	#E	#M	Pointer	Link	Assembling	Recognition	Detection	Total	
DroidBench	186	15	1	0	46	3	14	11	40	114	24,702
Malware	893	1,327	6,070	5	4,818	108	55	747	2,358	8,086	65,241
Real Apps	5,392	3,900	75,117	10	17,114	611	453	3,742	13,301	35,221	135,763

a directed graph, called the inter-component communication graph. ICCG captures all structure information of application, including call relationships and communication between different methods, and it contains all control flow information for each method. Then we propose an effective algorithm to search attacks in ICCG. The approach is proved to be feasible and effective in the experiments. In future, we expect to extend our detect algorithm to handle more complicated obfuscation or encryption techniques, and will continue enriching the attack model in order to handle more variants or new attacks.

Chapter 4

A Performance-Sensitive Malware Detection System Using Deep Learning on Mobile Devices

4.1 Introduction

With the currently increasing number of Android devices and applications (apps), plenty of Android users are benefited from that. The security and privacy concerns are also increasingly becoming the focus point to various mobile users and stakeholders. For example, more and more users store their personal data in mobile devices through various popular apps such as shopping, banking, and social apps. Consequently, since the last decade, attackers shift their attention to mobile apps. That makes Android malware undoubtedly become one of the most important security threats in this security field [10, 15].[1]

Therefore, how to detect Android malware becomes a severe problem. End-users always expect a secure environment which is maintained by the app markets. In other words, they consider their app sources are all trustable and secure enough. It is not surprising that the demands of Android malware detection approaches have been proposed, such as signature-based approaches [17–19], behavior-based approaches [20–23], data-flow analysis-based approaches [24–26]. We note that machine learning-based approach [27–32] is one of the most promising techniques in detecting Android malware.

[1]The work in this chapter has been published in [113–115].

With the available big data and hardware evolution over the past decade, deep learning has achieved tremendous success in many cutting-edge domains, including Android malware detection. Actually, all of the above protecting solutions are mostly on server side for app markets. However, when a new Android malware family is reported, not all the app markets are able to respond in a responsive time. The current analysis workflow always follows analyzing malicious behaviors within apps, building the detection models with the generated features and then performing the detection on the entire apps. Since the number of the real-world Android apps is extremely large, e.g., there are more than 3 million Android apps on Google Play Store, it is a time-consuming task to perform the complete detection with that large number of apps. Moreover, the apps from unofficial markets and third-party resources like XDA [39] are more vulnerable in the wild. The security of these kinds of apps is indeed unpredictable and uncontrollable.

The traditional server-side based malware detection surely has unignorable drawbacks when detecting such apps, because (1) it is a time-consuming task to upload the apps to server before the installation, especially for large apps; (2) the uploading process via the Internet is not secure. For example, attackers may modify the malware during the uploading period such that an incorrect "benign" result is returned. As a result, the users will install the malware. Hence, a last line of defense on mobile devices is necessary and much-needed. To address the severe problem, we intend to conduct Android malware detection on mobile devices instead of server side.

Actually, machine learning-based approaches have achieved better performance compared with other approaches in Android malware detection [27, 29, 31, 40]. In this work, we intend to deploy the trained deep learning (DL) models from server-side to mobile devices. While a computationally intensive deep learning software could be executed efficiently on server-side with the GPU support, such deep learning models usually cannot be directly deployed and executed on other platforms supported by small mobile devices due to various computation resource limitations such as the computation power, memory size, and energy.

According to the evaluation metrics of accuracy and time cost from different features and neural networks, we propose an effective and efficient Android malware detection system on mobile devices, named MobiTive. MobiTive leverages (1) a newly-proposed

feature extraction method from binary code; (2) a performance-based feature type selection mechanism; (3) a novel feature updating method through malicious behavior mining and understanding; (4) a customized deep neural network for classification. So that, MobiTive can provide a real-time and fast responsive environment on mobile devices.

In our comprehensive experiments, (1) we first divide the feature preparation procedure into two steps, which are raw data extraction and feature extraction, and evaluate the performance (time cost) separately to decide the feature selection. (2) With the selected features, we then provide an accuracy comparison between different feature categories. (3) The behavior-based feature updating method performs around 1%~5% accuracy increase. (4) We provide a comprehensive comparison between seven different neural networks (e.g., CNN, LSTM, and GRU) to show the potential improvement of our customized DL models on network definition. (5) We further evaluate the performance and accuracy of MobiTive on different real mobile devices by using our customized RNN model and compare with dynamic device-end solutions. (6) In the last part of our experiments, we perform an analysis of the performance trend on mobile devices from three different aspects and integrate the results to provide a strong evidence on the potential of MobiTive in practice. Specifically, MobiTive achieves a relatively higher classification accuracy (i.e., 96.78% accuracy) on real testing data in the wild and mobile devices with relatively lower overhead (i.e., less than 3 seconds on average for one app).

In summary, we make the following main contributions.

- We propose MobiTive, a device-end solution to protect mobile devices from malware threats in real-time efficiently by leveraging customized deep neural networks and binary features. This research work aims to detect malware directly on mobile devices as a pre-installed and run-time solution rather than detecting them on common servers or monitoring them after installation.

- We propose a new feature extraction method from binary code, as well as a feature updating method based on the understanding of malicious behaviors. Due to the high performance demand of mobile devices, we evaluate the different performance (time cost) and accuracy with various feature types and neural networks, and further provide a comparison against 4 existing Android malware detection approaches. Besides, we also investigate the accuracy on multi-class classification task.

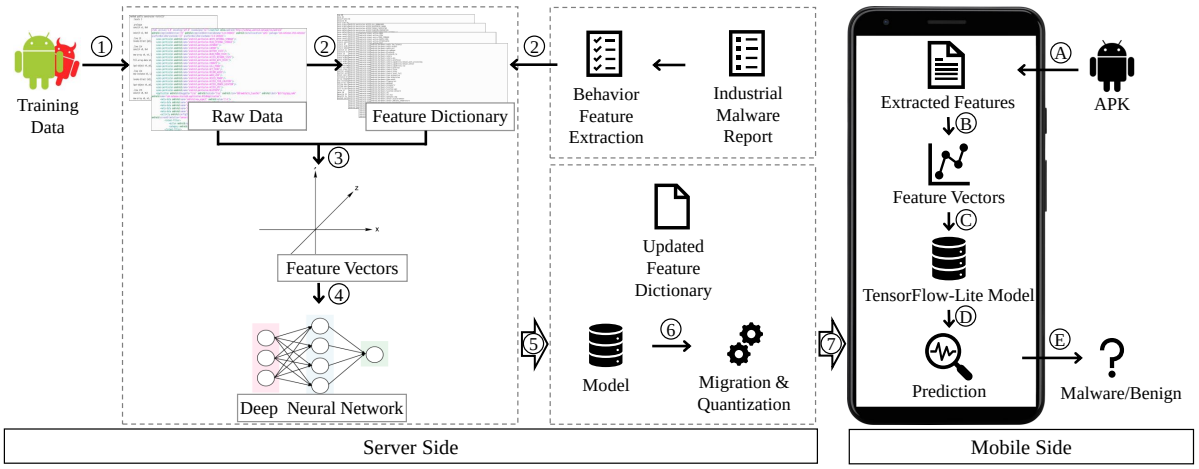


Fig. 4.1: Overview of MobiTive

- We evaluate and investigate the different performance on multiple devices from different manufacturers, and further provide insights of the current quality and potential for our approach according to the feature extraction and prediction time cost on six real mobile devices. Meanwhile, an additional comparison on run-time efficiency and discussion on effectiveness is provided to show the advantages against dynamic malware detection system based on behavior analysis.

4.2 System Design of MobiTive

In this section, we first introduce the overview of our approach, and then detail each of the key phases.

4.2.1 Overview of MobiTive

To achieve our target, we propose MobiTive, whose functionality could be divided into two main parts (i.e., parts of server side and mobile side), as shown in Fig. 4.1. The first part of our system contains feature preparation, DL model training, model migration and quantization. The second part is the deployment phase on mobile devices by using the migrated/quantized models.

In our previous work [114], we involved multiple features (i.e., manifest properties, API calls, and opcode sequences) extracted from decompiled apps. In this work, to

improve the performance of MobiTive, we propose a new feature extraction method. Instead of decompiling APK into source code, like smali code, we extract and vectorize the manifest properties and API calls from binary code directly (step ①). We combine a performance-based feature selection mechanism and behavior-based feature updating method to generate the feature dictionary (step ②). With the customized deep neural networks and extracted feature vectors (steps ③ and ④), the first part allows to provide a trained DL model and a feature dictionary for the second part (step ⑤). To make the model adaptive to mobile devices, we then migrate the pre-built DL model to a TensorFlow Lite model. Also, a quantization phase [116], which is a general technique to reduce model size while also providing lower latency with little degradation in accuracy, is presented as a performance optimization for the mobile devices (step ⑥).

Fig. 4.1 shows that the second part loads the quantized DL model and feature dictionary into mobile devices. After that, when an application is downloaded from market or third-party market, MobiTive can extract feature vectors from it and deliver the result to MobiTive (steps Ⓐ→Ⓒ). After predicting with the loaded DL model, we obtain a certain level of confidence based on predictive output to know whether the downloaded Android app is a malware or not. (steps Ⓓ→Ⓔ).

4.2.2 Feature Preparation

To determine the features used in MobiTive, we perform a comparison of the extracting performance for most commonly-used features in previous malware detection approaches [27, 29, 31, 68, 83]. Based on the performance-based feature selection method, manifest properties and API calls are selected in our device-end scenario (*Feature Selection*). Also, to update the feature dictionary and improve the representatives, we propose a behavior-based method based on industrial malware reports (*Feature Updating*). To get the features from the APK, we first unzip the package instead of decompiling it to reduce time cost. Among the unzipped binary files, we can extract the two feature types from the raw data (*Feature Extraction*).

4.2.2.1 Feature Selection

Manifest properties such as used permissions, intents, and hardware features are widely-used features to detect Android malware [27, 29, 31]. AndroidManifest.xml file can be

Table 4.1: Selected Features

	#API calls	#Manifest properties
Collected from samples or by Android documentation	2,989,011	613
Pruning by manual	1,509	613
Updated by behavior-based analysis	2,290	625

Note: Feature lists can be found on [117].

easily decoded from APK file through existing tools, which benefits the feature extraction procedure. It is belonging to a lightweight feature type, which would be adopted by the performance-sensitive system, like MobiTive. In terms of the usefulness, API calls are more representative and important feature types because almost all malicious behaviors would be demonstrated by API callings. Apart from the individual API call, API call sequence may contains more semantics, such as opcode code sequence. However, the extraction procedure of these two feature types causes a lot of time due to analyzing source code or smali code. A novel feature extraction method for API calls is much-needed due to the energy limitation of mobile devices. Besides the above two widely-used feature types, we also evaluate other potential structural features by their performance behaviors, such as inter-procedural control-flow graph (ICFG) and call graph (CG). ICFG not only provides the control-flow graph but also contains the inter-procedural between the components within apps. CG represents the calling relations between different methods. By evaluating the performance (time cost) of all these potential feature types based on the two different extraction steps, which are raw data and feature vector extraction (steps ① and ③), we select unzipping APK to extract raw data first and select API calls and manifest properties as our feature types due to the better extraction performance compared with others (details in §4.3.2).

To get the feature vectors (step ③), we build a feature dictionary (step ②) according to the two types of features selected by performance comparison. Specifically, we build the manifest property dictionary by following the official Android documentation. As shown in Table 4.1, the manifest properties contain 613 features in total, including 324 used permissions, 213 intents, and 76 hardware features. In terms of the API call dictionary, we conduct a data-driven analysis to determine the feature lists. Specifically, by parsing the API calls from more than 60k real-world Android apps collected from Google Play

Store and malware, we collect 2,989,011 unique API calls in total. We summarize three rules to reduce the size of API calls through manual analysis. Firstly, we remove the obfuscated API calls. Secondly, we delete the API calls that are not related to potential malicious behaviors, such as View loading API. Last, we remove the third-party API calls, because these API calls exist and customize in an app, may rarely appear in other apps. As shown in Table 4.1, after pruning, the number of selected API calls is only 1,509. The details of feature lists can be found on our website [117]. We build a feature dictionary based on the 1,509 API calls and 613 manifest properties for matching on the features of permission, intent, hardware, and API calls (step ②).

4.2.2.2 Feature Updating

The quality of machine learning-based detection approaches highly depends on the selected features, which means that a more comprehensive feature coverage of malicious behaviors makes more benefits MobiTive. To enrich the feature coverage of malicious behaviors, we collect hundreds of industrial malware reports from Symantec Threats [38]. With the collected text-based reports, we perform a manual analysis and summarize 23 kinds of basic potential malicious behaviors as a supplement for the selected features. Note that, the malware reports detail the malicious behaviors and the core code level features, including both API calls and manifest properties. Also, a behavior-based feature understanding and verification by three co-authors are performed to ensure the manual results. As a result, except the features in the original feature dictionary, there are 46 new API calls and 12 new manifest properties in total, which are updated for a new feature dictionary. We also extend the new API calls with their package name. For example, if a new API call has package name as "android/net/Uri", we extract all the API calls under this package. As shown in Table 4.1, there are 781 API calls extended according to the 46 new API calls. Finally, we supplement our feature dictionary and update to 2,290 API calls and 625 manifest properties. The new feature dictionary is used to get the feature vector of each app for model training.

4.2.2.3 Feature Vector Extraction

As we mentioned in feature selection, the traditional feature vector extraction methods cause a lot of time due to the cost of decompiling and extracting from source code such

Table 4.2: Deep Neural Network Architecture: GRU and LSTM

Input		
Reshape	input	(None, 2915,)
	output	(None, 1, 2915)
GRU/LSTM	input	(None, 1, 2915)
	output	(None, 128)
Dropout		
Softmax Classification		

as Java code and smali code. To improve the extraction performance, we propose a novel feature vector extraction method from binary code instead of source code. Specifically, by analyzing the inner architecture of Dalvik binary file (classes.dex), we find there exists an API table, which is used to match the executable symbols and API strings. We extract API calls by parsing the API table in classes.dex file based on the address and offset defined in the metadata. Meanwhile, to get access to the information in binary format AndroidManifest.xml, we firstly generate a standard output with a XML decoder, Axmldec [118]. By analyzing the decoded manifest file, the manifest properties can be extracted.

4.2.3 DL Model Construction

4.2.3.1 DL Model Training

To discover the potential accuracy improvement and usability for different deep neural networks, we present seven widely-used networks to train the classifier, with the input feature vectors generated by step ③. As shown in Table 4.2, 4.3 and 4.4, we customize the RNN models to adopt the device-end scenario and improve performance. For simple RNNs in Table 4.2, the first computational layer is a LSTM/GRU layer with 128 neural units. After the computation, the dimension of input tensor will reduce to 128 from (1, 2,915). Then, there will be a dropout layer, the dropout rate is 0.5. At last, the result is passed to a softmax classifier function to get the final training result. For stacked RNNs in Table 4.3, there will two stacked LSTM/GRU with dropout layers instead. For bidirectional RNNs in Table 4.4, we apply a bidirectional LSTM/GRU layer instead of the original LSTM/GRU layer.

Additionally, we build the convolutional neural network (CNN) with reference to the conference version [114]. As shown in Table 4.5, the first layer of the CNN model is

Table 4.3: Deep Neural Network Architecture: Stacked GRU and LSTM

Input		
Reshape	input	(None, 2915,)
	output	(None, 1, 2915)
GRU/LSTM	input	(None, 1, 2915)
	output	(None, 1, 128)
Dropout		
GRU/LSTM	input	(None, 1, 128)
	output	(None, 128)
Dropout		
Softmax Classification		

Table 4.4: Deep Neural Network Architecture: Bidirectional GRU and LSTM

Input		
Reshape	input	(None, 2915,)
	output	(None, 1, 2915)
Bidirectional (GRU/LSTM)	input	(None, 1, 2915)
	output	(None, 256)
Dropout		
Softmax Classification		

Zero Padding Layer. With input feature vectors, we need to fit it to the the training part. Hence, we add two nonsense dimensions to the end of input since the kernel size of our convolutional layer is 3. Then, the resulting vector is reshaped to a matrix, whose horizontal dimension is 3, and send to the next layer. The second layer is the convolution layer with a 3 kernel, which receives the embedded matrix as its input and applies the convolution filter to produce activation maps for each batch. Before delivering the batches to the hidden layer, a global max pooling is used after activation to reduce the dimensions. Finally, the vector is passed to a hidden full layer, which is a multi-layer perception, for classification. To detect the relation between the result vector, we construct two sub-layers in the hidden layer, each of them contains a Rectified Linear Unit activation function. At last, the result from the hidden layer is passed to a softmax classifier function to get the final training result.

4.2.3.2 DL Model Migration and Quantization

To deploy our pre-trained DL model on mobile devices, we convert and migrate the pre-trained model to a TensorFlow Lite model, which is supported by Android operating system (step ⑥). Specifically, we migrate the TensorFlow model to a mobile readable

Table 4.5: Deep Neural Network Architecture: CNN

Input		
Reshape	input	(None, 2915,)
	output	(None, 1, 2915)
Zero Padding Layer	input	(None, 1, 2915)
	output	(None, 1, 2916)
Reshape	input	(None, 1, 2916)
	output	(None, 3, 708)
Convolutional Layer	input	(None, 3, 708)
	output	(None, 64, 706)
Relu		
Global Max Pooling	input	(None, 64, 706)
	output	(None, 64)
Linear Dense Layer	input	(None, 64)
	output	(None, 16)
Relu		
Linear Dense Layer	input	(None, 16)
	output	(None, 2)
Softmax Classification		

TensorFlow Lite model with a TensorFlow Lite converter [56]. Apart from the model migration, we also quantize our pre-trained model to improve the performance on mobile devices, which does not affect the accuracy of detection much. In the experiments, we measure the performance of accuracy and time cost affected by the model migration and quantization (details in §4.3.3.3).

4.2.4 Real-Time Detection System

Before conducting a real-time detection, the quantized TensorFlow Lite model and feature dictionary should be deployed to the detection system in advance (step ⑦). There are three main steps before completing the prediction. The first step of MobiTive is feature preparation. When an APK file is received in step Ⓐ, MobiTive first unzips it into original assembly files such as AndroidManifest.xml, classes.dex, and other resources. Features of API calls and manifest properties will be extracted accordingly. We implement an API parser to extract the API calls from classes.dex directly based on the understanding of Dalvik binary code. Since the raw binary AndroidManifest.xml cannot be analyzed directly, we use a third-party decoder library, AXML [119], to get the decoded manifest file. By analyzing the decoded manifest file, the three kinds of manifest properties will be extracted from the XML tag. Hence, we can get both the

manifest property vector and API call vector in step Ⓑ. All the two types of features are transformed into a feature vector, we connect them together as the input of TensorFlow Lite model (step Ⓒ). With the quantized model, MobiTive can perform the prediction in step Ⓓ and show the final prediction result as a feedback in step Ⓔ. With the prediction result, the system can raise a warning to help users blocking the installation of the detected malware and further save its information (e.g., name, release version, provider, and checksum) in local database. Also, besides the actions on local devices, reporting the malicious applications' information to the corresponding market and synchronizing the malware information to the updating server can be another two options.

To deploy an update for the MobiTive in practice, the service provider firstly need to collect the new detected malware and update the training dataset. After the dataset updated, it will be able to obtain a new pre-trained model on the server. Then, the new model can be packed as a system patch and deployed to devices within an update directly. As a result, the updated system surely will improve the effectiveness and robustness of the protection on device based on the new delivered model.

More implementation details. The AXML version used in MobiTive is v1.0.1. The API parser used in MobiTive on Android devices is implemented based on the Dex2jar [120] (2.1-nightly-28). Unlike the original Dex2jar project, we do not decompile the Dalvik executable files (i.e., .dex files) back into .smali files or .class files. Instead, we only involve the binary formatting functions in Dex2jar and collect the API calls from the decoded API table. The API parser is served as an external lib file in the MobiTive. Technically, the classification functionality of MobiTive on Android devices is consist of 3 main parts. Different from the well established high level API provided in Keras (2.2.4), the basic data structure used in the computation with TensorFlow Lite (0.0.0-nightly) on Android devices is bytebuffer. Thus, firstly, there will be a step to convert the input vector and model into bytebuffer format. Secondly, by loading the model into a TensorFlow Lite interpreter, we can feed the input bytebuffer into the interpreter and get the result matrix. At last, by using an argmax function on the result matrix, the final prediction result can be obtained.

4.3 Evaluation

In this section, our experiments are technically organized into three subsections based on the model deployment environments (i.e., PC/server and mobile devices). First, the goals of our experiments on PC/server are to investigate: (1) the performance of extraction time of different raw data (techniques) and feature types; (2) the effectiveness of behavior-based feature updating method; (3) the detection accuracy of different deep neural networks; (4) the comparison with other existing learning-based Android malware detection solutions; (5) the accuracy of multi-class classification on malware families.

Second, based on the observed findings and obtained results, we further evaluate: (1) the performance of feature preparation on six different real devices with six different app sizes from 5MB to 50MB; (2) the efficiency of detecting with different RNN models on real devices; (3) the usability (i.e., performance and accuracy) of MobiTive on six different real mobile devices; (4) the efficiency of MobiTive by comparing to dynamic behavior-based run-time detection systems.

In the end, we conduct a study on the hardware performance trend of Android mobile devices to provide insights into the future usability of MobiTive.

4.3.1 Experiment Environment

The experiments on server side are run on a Ubuntu 16.04 server with two Intel Xeon E5-2699 V3 CPUs, 192GB RAM, and NVIDIA GeForce 2080Ti GPU. To evaluate our approach, we select six different Android mobile devices to evaluate the performance and accuracy of our approach on real mobile devices. Among them, there are four common specification devices (Nexus 6P, Huawei Mate 10, HTC U11, and LG G6), a flagship device (Huawei P30), and a low-profile device (Samsung Galaxy J7 Pro) (detailed specifications provided on our website [117]). The implementation language of our system on server is Python 3. To get access to the raw data and features, we use seven different kinds of existing tools, which are axmldec [118], AXML [119], ApkTool [121], AndroGuard [122], Dex2jar [120], Soot [123], and FlowDroid [124]. axmldec is a C++ project which can be used to decode binary manifest file into readable XML format file. AXML is a library designed to parse binary Android XML files. It is written in Java

and can be used in an Android app as an external library. ApkTool is a tool for reverse engineering, which can decompile the apk file and generate the resources, which contains manifest, smali files, and etc. AndroGuard is a Python tool, which cannot only decode the resources but also disassemble bytecode to Java code. Also, with the help of AndroGuard, we can easily generate the call graphs (CG) and data-flow graph for an Android app. Dex2jar is a project which contains tools to work with Android .dex and Java .class files. Soot is a Java optimization framework, which can be used to extract the call graph (CG). FlowDroid is a static taint analysis tool for Android apps. It is applied to generate inter-procedural control-flow graph (ICFG). Apart from the above existing tools for feature extraction, JitPack is a novel package repository for JVM and Android projects, which can build the project to a ready-to-use artifacts (i.e., jar and aar). The deep neural networks and training projects are implemented with Keras [125], Numpy [126], Scikit-learn [127], and TensorFlow [128] libraries.

4.3.2 Effectiveness of Feature Extraction, Feature Updating, Feature Category Selection, and Neural Network Selection

4.3.2.1 Performance evaluation of feature extraction

In this experiment, we split the feature extraction time into two parts respectively (i.e., APK→raw data→feature) along with the technical procedures in feature preparation phase to show the performance advantages of our selected features.

Dataset. To mitigate the uncertain influence from apps' size on the time cost of feature extraction, we randomly collect 60 Android apps in total among 6 different sizes (i.e., 5MB, 10MB, 20MB, 30MB, 40MB, and 50MB) to provide a clear performance comparison between different extraction methods such as feature extraction from source code and binary code.

Setup. In this experiment, we first evaluate the extraction time (APK→raw data) of 3 different raw data types, which are widely-used in the existing static analysis based malware detection work (i.e., ICFG extracted by FlowDroid, CG extracted by Soot and AndroGuard, decompiled files obtained by ApkTool), together with our selected extracting method (i.e., binary code obtained by unzipping).

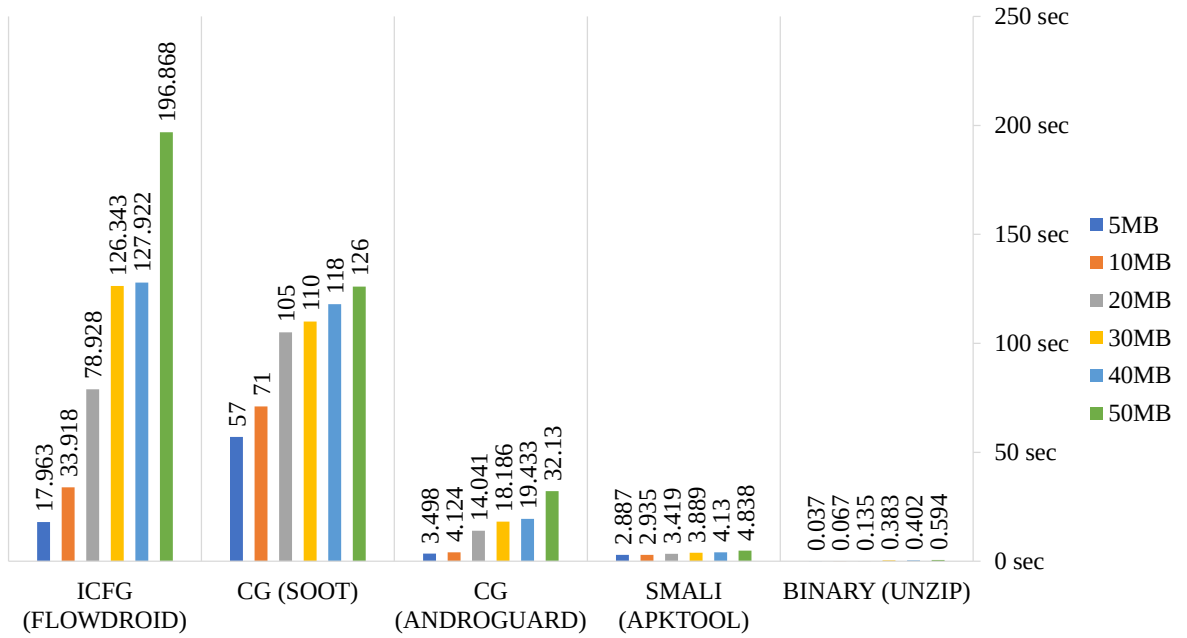


Fig. 4.2: Extraction time of different raw data types

Secondly, apart from the above raw data extracting methods, we further evaluate the extracting performance (raw data→feature) of 3 different feature types (i.e., manifest properties, API calls, and opcode sequence) that generated from two kinds of raw data types (i.e., decompiling and unzipping). We do not further evaluate the graph-related features due to the large time cost of raw data extracting.

For the decompiled manifest and smali files, we use a XML tag parser to extract manifest properties from manifest file decompiled by ApkTool. To extract API calls, we obtain the result by matching the API call dictionary and smali files directly. We extract the opcode sequences for each smali file by matching it to the opcode list [129]. For the unzipped binary manifest and Dalvik binary files, we evaluate the extraction time of 2 different feature types (i.e., manifest properties and API calls). To extract manifest properties from the binary manifest file, we apply Axmldec to extract manifest properties. We extract API calls by loading the API table directly with the offset and size defined in the metadata of the Dalvik binary file.

Results. We demonstrate the results from the 2 aspects (APK→raw data and raw data→feature) as below.

(1) *Raw data extraction.* Fig. 4.2 shows the extraction time of ICFG and CG is too large to be accepted performance-sensitive approach like MobiTive. Specifically, extracting ICG via FlowDroid takes 196.868 seconds on 50MB apps and even 17.963 seconds on 5MB apps on average. Generating CG with Soot takes 126 seconds on 50MB apps and 57 seconds on 5MB apps, and it costs 32.13 seconds and 3.498 seconds accordingly when prepared with AndroGuard. AndroGuard achieves a better performance than Soot on CG extraction. As for MobiTive, the detection should be performed in a responsive period comparing to the app installing time, users cannot buy it if the reacting time takes too long. The extraction time costs of decompiling, which applied in our previous work [114], is acceptable but also limited, by comparing with the app installing time on average. Considering the time cost of extracting raw data by unzipping, 5MB apps take only 0.037 seconds and 50MB apps take 0.594 seconds, which reaches a much better performance than the processing time of decompiling. Therefore, we decide to use unzipping as our raw data extraction method.

(2) *Feature extraction.* Fig. 4.3 shows that the time consumption of features extracted from decompiled files is much longer than the same features generated directly from unzipped binary files. Specifically, in terms of the time cost of API calls, extracting them from 5MB apps only takes only 0.042 seconds on average. However, if we extract the API calls from decompiled smali files, it takes 2.923 seconds. For the 50MB apps, it will cost 0.601 and 5.002 seconds for extracting the API calls. Considering the extraction time of manifest properties, 5MB and 50MB apps will take 2.89 and 4.841 seconds, when we extract the manifest properties from the decoded manifest file by ApkTool. When we extract them from the unzipped binary manifest file, the time is reduced to 0.041 and 0.599. Apart from manifest properties and API calls, we find that the extraction time of opcode sequence is much larger than the other two feature types. For 50MB apps, it will take over 6 seconds on average. Therefore, to improve the performance of MobiTive in feature extraction, we decide to use the two feature types with shorter extraction time as our model inputs.

4.3.2.2 Accuracy evaluation of behavior-based feature updating method

In this experiment, we evaluate the effectiveness of the behavior-based feature updating method presented in §4.2.2.2 by comparing the results between the features used in our

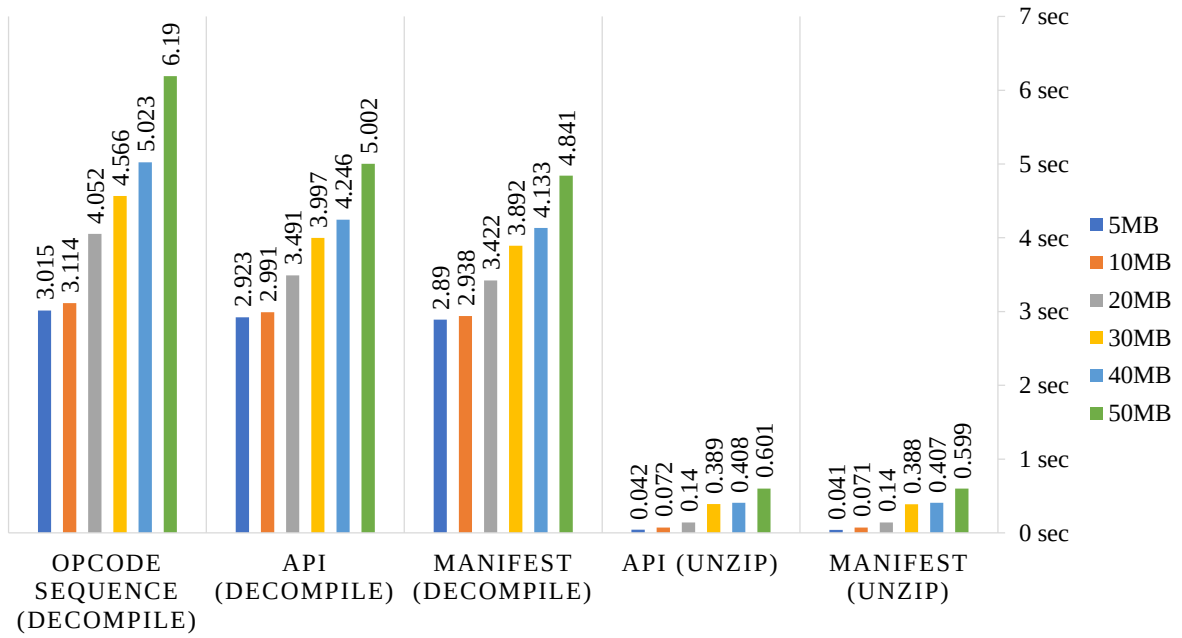


Fig. 4.3: Extraction time of different feature types

previous work [114] (MobiDroid) and MobiTive.

Dataset. As shown in Table 4.6, we collect more than 70k Android apps in total as our evaluation subject. Specifically, these apps consist of 29,010 Android malware, and others are benign apps crawled from Google Play Store. However, these might be malware on the official market. To filter the potential malware as far as possible, we upload them to VirusTotal [130], which is an online antivirus service with over 60 security scanners, to make a verification. The 29,010 malicious samples contain 5,560 apps that downloaded from Drebin [27], 1,260 apps validated in Genome project [10], 20,000 crawled from VirusShare, and the remaining are used in KuafuDet [31], including 360 from Contagio Mobile Website [131] and 1,830 from Pwnzen Infotech Inc. [132]. In summary, we collect a large-scale dataset of benign and malicious samples for the following experiments. Since our dataset comes from multiple sources, there have a lot of duplicated samples. Therefore, we perform a hash check for eliminating redundant apps among malicious and benign apps. During the data prepossessing, which has raw data decompiling and feature vector generation steps, we receive some failed cases due to the capabilities of API parser. The rest of the failures are just caused by the broken APK packages, we also remove them directly. As a result, we choose 18,000 benign and malicious samples

Table 4.6: Used Dataset

Datasets	Labels	Original Size	Deduplicated Size
Drebin	Malware	5,560	3,561
Genome	Malware	1,260	1,009
Contagio	Malware	360	198
Pwnzen	Malware	1,830	572
VirusShare	Malware	20,000	12,660
Total	Malware	29,010	18,000
Total	Benign	45,284	18,000

Table 4.7: Detection results of feature updating

Feature Dictionary Size	Networks	Accuracy	Precision	Recall
1509 API + 613 Manifest	CNN	90.03%	91.00%	89.26%
2290 API + 625 Mniafest	CNN	95.11%	95.06%	95.16%
1509 API + 613 Manifest	LSTM	95.44%	95.56%	95.34%
2290 API + 625 Mniafest	LSTM	96.56%	96.72%	96.40%
1509 API + 613 Manifest	GRU	95.56%	95.22%	95.86%
2290 API + 625 Mniafest	GRU	96.75%	96.78%	96.72%

respectively from our dataset to conduct the following experiments. In training stage, we divide these 18,000 malware and 18,000 benign apps into three parts, 80% of them are configured as training data, other 20% are equally split into validating and testing data.

Setup. Because our previous work MobiDroid [114] applied three types of features (i.e., API calls, manifest properties, and opcode sequence), in this experiment, we determine to take API calls and manifest properties (i.e., 1,509 and 613), which our behavior-based feature updating method may benefit on, as the feature of MobiDroid to reveal the improvement on detection. Meanwhile, the updated version used in MobiTive has 2,290 API calls and 625 manifest properties. For each feature version, we apply three kinds of deep neural networks, which presented in §4.2.3.1, to investigate whether the feature updating method can improve the accuracy of our system.

Results. In Table 4.7, the accuracy of updated feature version on CNN, LSTM and GRU is 95.11%, 96.56% and 96.75%. Comparing to the previous results, there is around 1%~5% improvement after feature updating. Therefore, based on the result, we accept updating features summarised from potential malicious behaviors a part of our input feature set.

Table 4.8: Detection results of feature categories and networks on server side

Categories	Neural Networks	Accuracy	Precision	Recall
Manifest	CNN	79.89%	79.50%	80.12%
API Calls	CNN	93.17%	92.33%	93.90%
Two Types	CNN	95.11%	95.06%	95.16%
Two Types	LSTM	96.56%	96.72%	96.40%
Two Types	GRU	96.75%	96.78%	96.72%
Two Types	Stacked LSTM	96.64%	96.83%	96.46%
Two Types	Stacked GRU	96.67%	97.00%	96.36%
Two Types	Bidirectional LSTM	96.61%	96.67%	96.56%
Two Types	Bidirectional GRU	96.78%	97.00%	96.57%

4.3.2.3 Accuracy evaluation of feature category selection and deep neural network selection

In this experiment, to find out the correlation between selected features and the effectiveness of different deep neural networks on detection accuracy, we first evaluate the effect of two newly-updated feature categories (Table 4.1) on detection accuracy separately. Second, we investigate the effect of computational architecture in different deep neural networks on detection accuracy.

Dataset. The dataset configuration used in this experiment is same as §4.3.2.2 (Table 4.6).

Setup. To find out the correlation between the two selected features (i.e., manifest properties and API calls), we investigate their corresponding accuracy by accepting both single and combined feature categories as the input of a same neural network with a same training data configuration. To determine the best deep neural network, we evaluate seven widely-used neural networks by using the combined two feature categories.

Results. We demonstrate the results from the 2 aspects (feature category selection and network selection) as below.

(1) Feature selection. As shown in Table 4.8, the accuracy of the three CNN models is 79.89%, 93.17% and 95.11%. By comparing the accuracy of feature categories, we decide to use manifest properties and API calls together as an input bundle in our approach since the input with two feature types has the best result.

(2) Network selection. In general, RNN models perform a better accuracy than CNN models. A possible reason is that RNN has an internal state (memory), which can also

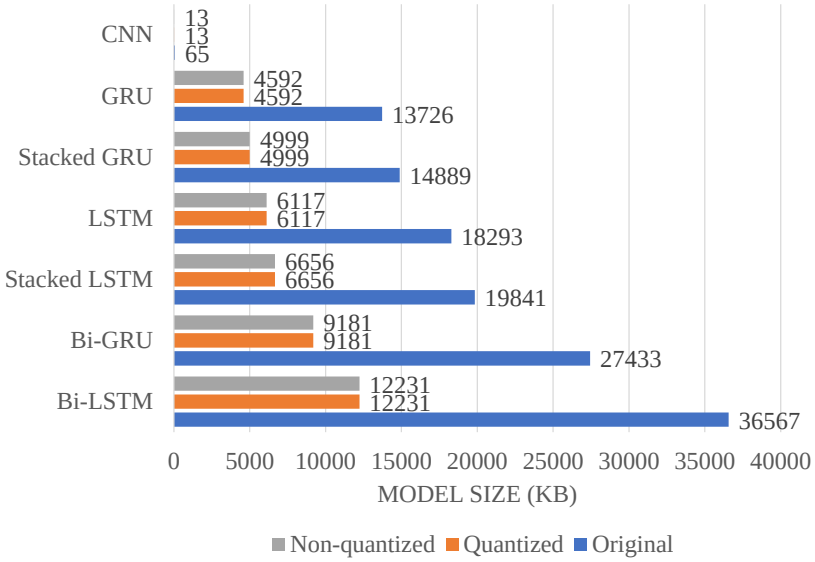


Fig. 4.4: Comparison of model size changes on migration and quantization

take the correlation between the different feature positions into consideration. In the training stage, this internal state will make RNN be able to keep the highly potential related in a long-term and finally keeps the most corresponding feature positions. However, CNN considers every different feature position individually in training. In terms of RNN models, GRU and bidirectional GRU achieve a similar accuracy (96.75% vs. 96.78%), which is better than other RNN models' accuracy. They also have a better recall than precision (96.78% vs. 96.72% for GRU and 97.00% vs. 96.57% for Bidirectional GRU). Besides, we also compare the size of original pre-trained model with the quantized and non-quantized models. In Fig. 4.4, we can find that the size of the original pre-trained model reduces 3 times on RNNs and 5 times on CNN by migrating it to the TensorFlow Lite model. In the experiment, whether the quantization configuration is enabled or not, the migrated model size for both RNN and CNN models will keep unchanged.

4.3.2.4 Comparison between the existing learning-based Android malware detection systems and MobiTive

In this experiment, we evaluate our MobiTive together with several existing learning-based Android malware detection systems on both two directions, effectiveness and efficiency.

Table 4.9: Comparison of MobiTive against existing approaches

Feature Type	Extraction Time (s)	Classification Method	Accuracy	System
Opcode Seq	5.065	Deep Learning(CNN)	94.79%	McLaughlin et al. [83]
API Call Seq	3.515	Deep Learning(CNN)	96.25%	MalDozer [133]
iCFG	198.920	Representation Learning	90.98%	Apk2Vec [70]
Manifest API Call Opcode Seq	5.892	Deep Learning(CNN)	96.87%	MobiDroid [114]
Manifest API Call	0.051	Deep Learning(GRU)	96.75%	MobiTive

Dataset. The dataset configuration used in this experiment is same as § 4.3.2.2 (Table 4.6).

Setup. We briefly compare our MobiTive with 4 open-source learning-based Android malware detection approaches, which applies different types of features (i.e., vector, sequence, and graph) as their inputs. There are three reasons to help us illustrating why we select these four approaches. By conducting a study on the corresponding literature which published in recent years, we first survey them on the feature types, and then select one representative work from each organized column. Further, by searching on the Github and sending emails to the authors, we obtain the source code of these four approaches and further evaluate them with our dataset to provide a more concrete comparison. Since one of our basic concept in this work is balancing the performance and accuracy to satisfy user's real usage, we not only evaluate the accuracy, but also compare the average feature extraction time for each approach on our dataset.

Results. As shown in Table 4.9, comparing to McLaughlin et al. [83], MalDozer [133], Apk2Vec [70], there are obvious improvements on both the accuracy and feature extraction time cost on PC. Considering the approaches based on sequential features, the accuracy of MobiTive is higher than McLaughlin et al. [83] and MalDozer (96.75% vs. 94.79% and 96.25%), and the time cost of extracting feature is almost improved for 100 and 70 times than them (0.051 vs. 5.065 and 3.515 seconds). Considering our previous work, MobiDroid [114], the accuracy of MobiTive is a little lower than MobiDroid (96.75% vs. 96.87%), however, with a tiny decrease at 0.12% on the accuracy, the feature

Table 4.10: Multi-Class Dataset

Name	Type	#
admogo	Adware	1,918
adwo	Adware	3,882
airpush	Trojan	6,368
anydown	Adware	1,343
baiduprotect	Adware	3,026
cnzz	File-Infector	970
commplat	File-Infector	1,442
domob	File-Infector	5,696
dowgin	Adware	3,223
feiwo	Adware	1,694
fictus	Adware	1,435
gappusin	Adware	9,378
igexin	Spyware	3,911
jiagu	Riskware	2,662
kuguo	Adware	4,031
kyview	Adware	1,433
leadbolt	Adware	5,929
mecor	Riskware	833
plankton	Trojan	2,571
revmob	Trojan	7,517
scamapp	Trojan	868
Total		70,130

extraction time cost of MobiTive is almost 100 times shorter than MobiDroid (0.051 seconds vs. 5.892 seconds).

4.3.2.5 Accuracy evaluation of multi-class classification

In this experiment, we evaluate the effectiveness of our MobiTive on predicting malware in different virus types such as Spyware and Trojan.

Dataset. We collect 70,130 Android applications from VirusShare as shown in Table 4.10 and classify them with VirusTotal [130], which is an online detection platform, to retrieve their types as our ground truth in multi-class classification. Finally, we have 21 virus-labels of these Android malware, which locate in 5 types (i.e., Adware, Spyware, Riskware, Trojan, and File-Infector). In training stage, we also accept the same data split in binary classification (i.e., 80%, 10% and 10%) as the train/validate/test data split portion.

Setup. To evaluate the effectiveness on classifying malware into different virus types,

Table 4.11: Detection result with multi-class dataset

Name	Type	#	Accuracy
admogo	Adware	192	84.38%
adwo	Adware	388	86.08%
airpush	Trojan	637	95.92%
anydown	Adware	134	100.00%
baiduprotect	Adware	303	98.35%
cnzz	File-Infector	97	100.00%
commplat	File-Infector	144	100.00%
domob	File-Infector	570	93.33%
dowgin	Adware	322	78.88%
feiwo	Adware	169	95.86%
fictus	Adware	144	96.53%
gappusin	Adware	938	96.16%
igexin	Spyware	391	99.74%
jiagu	Riskware	266	100.00%
kuguo	Adware	403	88.83%
kyview	Adware	143	79.72%
leadbolt	Adware	593	96.80%
mecor	Riskware	83	100.00%
plankton	Trojan	257	97.28%
revmob	Trojan	752	97.47%
scamapp	Trojan	87	100.00%
Overall		7,013	94.45%

we train a multi-class malware classifier on our collected dataset in Table 4.10 with the determined deep neural network (i.e., GRU).

Results. As shown in Table 4.11, we reach an overall accuracy at 94.45% in multi-class malware classification task. Considering those families respectively, on each of the two Riskware families, our MobiTive performs a perfect prediction, which has an accuracy at 100.00%. The detection accuracy of Malware families, cnzz and commplat in File-Infector type, also reach 100.00%, and the other family, domob, has an accuracy at 93.33%. Each of the malware types located in Trojan achieves an accuracy above 95% (i.e., 95.92%, 97.28%, 97.47%, and 100.00%). For the only spyware, the accuracy reaches 99.74% among 391 test malware applications. For Adware families, with our selected features, MobiTive achieves an accuracy above 95.00% in detecting anydown, baiduprotect, feiwo, fictus, gappusin, leadbolt (i.e., 100.00%, 98.35%, 95.86%, 96.53%, 96.16%, and 96.80%), however, it fails to provide a dependable prediction result on admogo, adwo, dowgin, kuguo, kyview (i.e., 84.38%, 86.08%, 78.88%, 88.83%, 79.72%).

Remark. To face the high latency during feature preparation, we find extracting API calls and manifest properties from unzipped Dalvik binary and binary manifest file will cost less than 1 second. To validate the effect of our newly supplemented features, we find the RNNs have an improvement at over 1% on the accuracy, and the accuracy of CNN increased by 5%. Meanwhile, by comparing the result on different feature categories and deep neural networks, we find that (1) two feature types combined input has a much better result than single feature type; (2) on average, the RNN models have a better result than CNN. GRU models have a better accuracy than the LSTM models on our dataset. Moreover, by comparing 4 existing approaches with MobiTive, it achieves a better performance with a considerable detection accuracy. To validate the effect on multi-class classification, we find that MobiTive can efficiently handle most malware families (i.e., 17/21 obtain an accuracy larger than 95%).

4.3.3 Effectiveness Evaluation of MobiTive on Mobile Devices

4.3.3.1 Performance evaluation of feature preparation on real devices

We evaluate the performance of feature preparation on real mobile devices in this experiment. The time cost of feature preparation step includes unzipping time and feature extraction time.

Dataset. The dataset configuration used in this experiment is same as §4.3.2.1.

Setup. We first evaluate the performance of raw data extraction by investigating the time cost of unzipping the applications with 6 different sizes on 6 different real Android devices. Second, with the extracted raw data (i.e., binary manifest file and Dalvik executable file), we further evaluate the performance of feature extraction by investigating the time cost of extracting the features from raw data on the devices.

Results. We introduce the results from 2 aspects (unzipping time and feature extraction time) as below.

(1) *Unzipping time evaluation on real devices.* Fig. 4.5 shows the average unzipping time of 50MB apps on common specification devices (Huawei Mate 10, HTC U11, Nexus 6P, and LG G6) locates between 1.023 and 2.586 seconds. For 5MB apps, it locates between 0.119 and 0.264 seconds. For the performance of low-profile device (Samsung Galaxy J7 Pro), the unzipping time of 5MB and 50MB apps are 0.261 and 3.918 seconds.

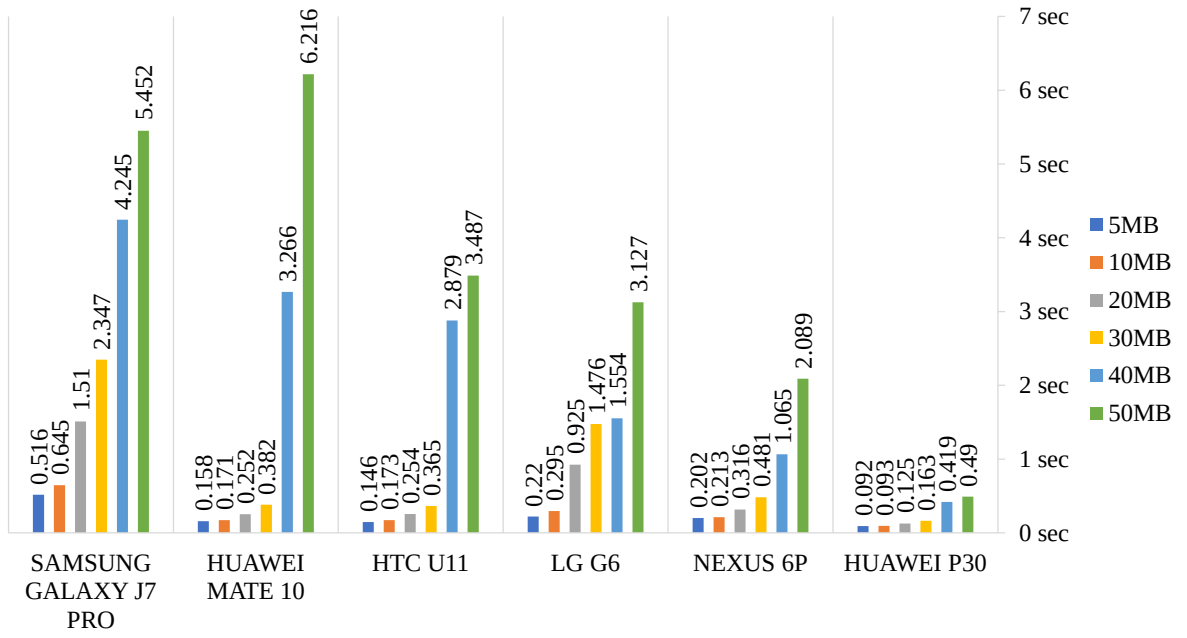


Fig. 4.5: Unzipping time on different mobile devices

Considering flagship device (Huawei P30), they are limited to less than 0.6 second. For 5MB apps, it only takes 0.046.

(2) *Feature extraction time evaluation on real devices.* Apart from the performance evaluation of unzipping time, we further evaluate the feature extraction time. To extract the API calls, we package our API call parser used on the server side into a jar with the help of JitPack. The API call parser is used to extract API calls from binary code. Since the XML decoder (axmldec) used on sever is implemented in C++, we apply AXML as a lib to extract the manifest properties, which is more suitable on the mobile side. Fig. 4.6 shows the average feature extraction time of 50MB apps on common specification devices locates between 2.089 and 6.216 seconds. For 5MB apps, it locates between 0.146 and 0.22 seconds. On low-profile device (Samsung Galaxy J7 Pro), the extraction time of 5MB and 50MB apps are 0.516 and 5.452 seconds. Considering flagship device (Huawei P30), they are limited to less than 0.49 second. For 5MB apps, it only takes 0.092 second, which is very fast in practice.

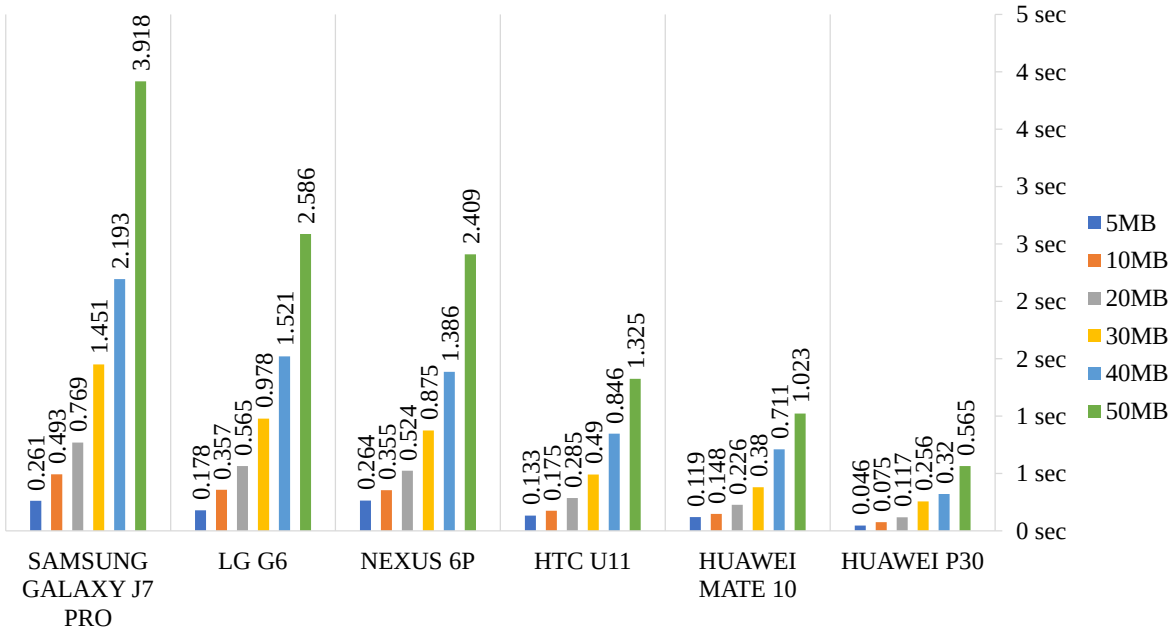


Fig. 4.6: Extraction time on different mobile devices

4.3.3.2 Performance evaluation of RNN models on real device

Besides analyzing the performance of unzipping and extracting features, in this experiment, we further evaluate the efficiency of prediction with different RNN models on real devices.

Dataset. To make sure that the test accuracy is comparable to the results obtained on server, the testing data used in this mobile-end experiment is same to the testing data generated by the data split function mentioned in §4.3.2.2 (Table 4.6), including 1,800 malware and 1,800 benign samples respectively. To get rid of the influence of feature preparation phase in this performance evaluation against RNN models, we directly use a set of feature vectors extracted from testing data as the input.

Setup. We first convert and migrate the RNN models obtained in §4.3.2.3 (i.e., simple RNN LSTM/GRU, stacked LSTM/GRU, and bidirectiontal LSTM/GRU) to TensorFlow Lite models and further deploy them on real device (e.g., Huawei P30). Secondly, to provide an insight for both the prediction accuracy and performance, we investigate the prediction time for each model with our dataset and organize the result together with the accuracy obtained in §4.3.2.3 (Table 4.8).

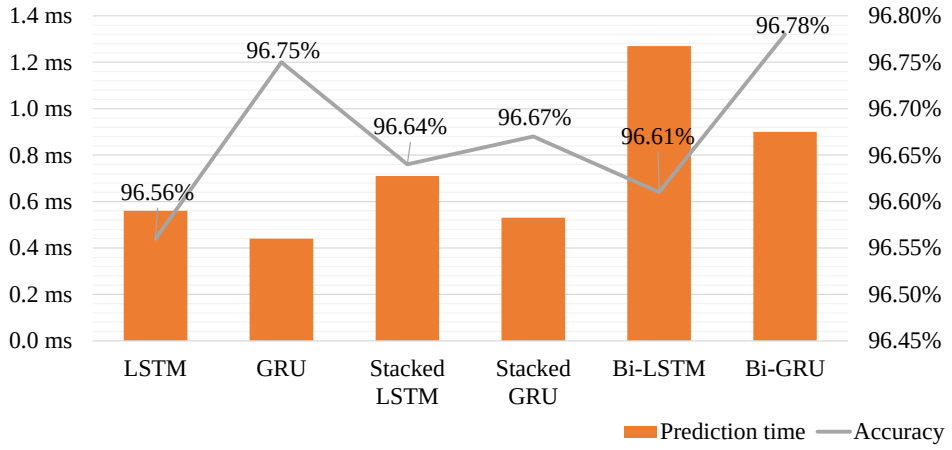


Fig. 4.7: The accuracy and prediction time of RNN models on Huawei P30

Results. As shown in Fig. 4.7, by comparing the prediction time of different RNN models, which presented in the histogram, we can see that the pre-trained model with GRU has the best performance than any others. Meanwhile, from the grey accuracy line in this figure, we can see it has the second highest accuracy among them, which only has a small difference comparing to the accuracy of bidirectional GRU (96.75% vs. 96.78%). Considering all situations, we select GRU model to evaluate the performance and accuracy of our approach on the real mobile devices.

4.3.3.3 Accuracy and prediction time on different real devices

In this experiment, we evaluate the effectiveness of MobiTive on real Android mobile devices by conducting a comparison experiment on test accuracy and the total prediction time on real devices.

Dataset. To make sure that the test accuracy is comparable to the results obtained on server, the testing data used in this mobile-end experiment is same to the testing data generated by the data split function mentioned in §4.3.2.2 (Table 4.6), including 1,800 malware and 1,800 benign samples respectively.

Setup. We first convert and migrate the GRU model obtained in §4.3.2.3 to quantized/non-qunatized TensorFlow Lite models and deploy them on real devices. Second, we record the average prediction time and detection accuracy by testing the quantized/non-quantized GRU models. Note that, to provide an insight for the performance of each processing

Table 4.12: Accuracy and performance of MobiTive on real mobile devices

Devices	Released Year	Unzipping Time (s)	Extraction Time (s)	Quantized	Accuracy	Prediction Time (ms)	Total Time (s)
Nexus 6P	Sep 2015	0.97	0.73	No	96.75%	1.14	1.70
				Yes	96.75%	1.14	1.70
LG G6	Apr 2017	1.03	1.27	No	96.75%	0.74	2.30
				Yes	96.75%	0.73	2.30
Samsung J7 Pro	Jun 2017	1.51	2.45	No	96.75%	1.67	3.96
				Yes	96.75%	1.66	3.96
HTC U11	Jun 2017	0.54	1.22	No	96.75%	1.10	1.76
				Yes	96.75%	1.10	1.76
Huawei Mate 10	Feb 2018	0.44	1.74	No	96.75%	1.10	2.18
				Yes	96.75%	1.09	2.18
Huawei P30	Mar 2019	0.23	0.23	No	96.75%	0.56	0.46
				Yes	96.75%	0.55	0.46

phase, we record the time cost by 3 parts (i.e., raw data unzipping, feature extraction, and prediction).

Results. With the obtained GRU model in §4.3.2.3 (accuracy: 96.75%), in Table 4.12, by comparing the accuracy of non-quantized and quantized models, we find that the accuracy of quantized model will almost equal to the non-quantized model for RNN (GRU). However, by comparing the prediction time of them, it shows that the performance of predicting with quantized model is a little better than non-quantized model. In this experiment, the result shows that the difference of prediction time, which brought by the quantization technique, is less than 0.01 microseconds. As a result of the current inadequate support for the operators in Tensorflow Lite, the structure of current applied deep neural networks are relatively simple. However, with a more complicated neural network, the quantization technique will definitely provide a performance boost during the prediction phase [134].

By calculating the unzipping, analyzing, and prediction time together, the time is always acceptable for mobile users (i.e., less than 3 seconds on average, less than 1 second in best practice). By comparing the specifications of these devices used in our experiment with the summarized devices' specifications (details on our website [117]), we find that the performance benchmark result of most newly released devices are better than the common devices selected in our experiments. Thus, we can claim that the current mobile phones can support our off-line prediction system smoothly.

Composition of overall time analysis. Comparing the feature preparation and prediction time in common cases in Table 4.12. we can see that the detection time only cost

Table 4.13: Run-time performance comparison of MobiTive against dynamic Android analysis tool

CPU Usage	Memory Usage (MB)	Energy Usage	Execution Time (s)	System
$P_{40\%}$	60	$Q_M \times 0.46$	0.46	MobiTive
$P_{10\%} \times n$	$130 \times n$	$Q_L \times t \times n$	∞	Inspeckage [135]

1. t is the execution time to finish one detection on an app.
2. n is the total number of monitored apps running in foreground or background.
3. P is the run-time CPU power usage in one detection.
4. Q is the run-time energy cost in one detection, Q_M and Q_L represent the energy cost at level medium and light (defined by Android Profiler [136]).

less than 1% among the total time on common spec devices. Thus, reducing the time cost in feature preparation will bring a considerable performance improvement for our detection system. It is also a strong motivation for us. Additionally, as a result of the installation mechanism, the downloaded Android APK will be always unzipped by the Android operation system. Thus, there will be a same step between our approach and the installing procedures, which is extracting the same raw data from the target APK. If we can deploy our approach on the Android operation system framework directly, the time cost of unzipping step in our approach will be saved. It is a new research point of this field in the future.

4.3.3.4 Performance comparison between MobiTive and dynamic run-time detection

In this experiment, we evaluate and discuss the efficiency of our MobiTive against other tool, which uses dynamic behavior analysis as the baseline technique.

Setup. To evaluate the run-time performance of our MobiTive against other tool, which applied dynamic behavior analysis as the baseline technique, we select Inspeckage [135], which is a state-of-art tool developed to offer dynamic analysis for Android applications, as the target system in this experiment. We investigate the run-time performance cost of both Inspeckage and MobiTive on three aspects (i.e., CPU, memory, energy) with the help of Android Profiler [136].

Results. In Table 4.13, the result shows that if detecting a **single application** successfully in the **same limited time period**, the CPU usage and energy consumption of Inspeckage look better than MobiTive (i.e., CPU: $P_{40\%}$ vs. $P_{10\%}$ and Q_M vs. Q_L),

but the average memory usage is 70MB larger than ours (i.e., 60 vs. 130MB). More importantly, the differences of Inspeckage and MobiTive on their basic mechanisms cause that more factors need to be involved in this evaluation. First, the protection of MobiTive is only provided before the installation of target application. In other words, every application will only trigger MobiTive's detection once in each installation and cost 0.46s on average. Considering the protection of Inspeckage is provided by monitoring and analyzing the behaviors of applications in real time, namely, it will be assigned to every application, no matter running in foreground or background. Meanwhile, unlike MobiTive can be slept after every detection, the Inspeckage has to be kept running all the time if any applications are running. Thus, the CPU usage, memory usage and energy usage of Inspeckage will turn to be $P_{10\%} \times n$, $130 \times n$ and $Q_L \times t \times n$, where t is the execution time to finish one detection on an application and n is the total number of protected applications running in foreground/background. In conclusion, in common scenario of Android devices' usage, malware detection tool using dynamic analysis, like the Inspeckage, will definitely lead to a higher performance cost than MobiTive.

Remark. To determine the feature preparation and prediction performance on real devices, we find (1) the feature preparation time is less than 4 seconds on average; (2) the prediction time of RNN models is less than 2ms on average. GRU costs 0.44ms with the best performance. Meanwhile, by comparing the result on six mobile devices, we find MobiTive costs (1) less than 3 seconds on common devices, (2) less than 0.5 second on flagship device. Meanwhile, by comparing the run-time performance of MobiTive and Inspeckage, we find that MobiTive can serve user with a much more efficient experience as an on-device protection system.

4.3.4 Analysis of Hardware Performance Evolution Trend of Android Mobile Devices

In this section, we conduct a study from three different aspects to investigate the hardware performance evolution trend of Android mobile devices.

To provide insights into the current and future usability of MobiTive, we study 45 widely-used chipsets, which released between 2016 and 2019. They are collected from three well-known brands, Exynos, Kirin, and Snapdragon We select 4–5 chipsets from

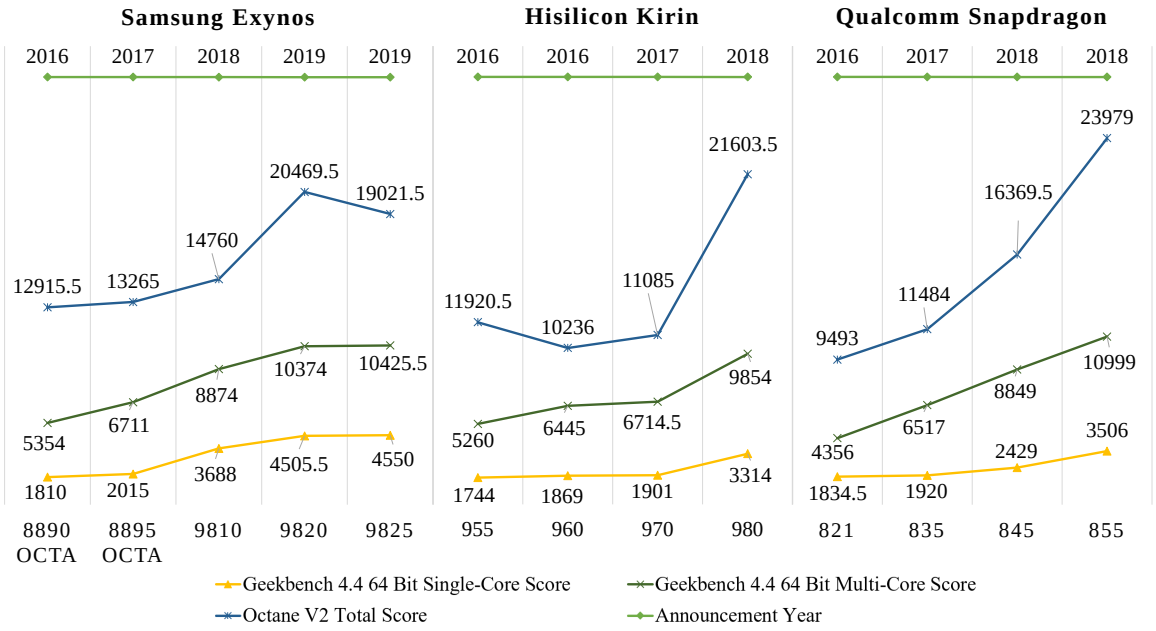


Fig. 4.8: Benchmark score comparison among different chipsets from Exynos, Kirin, and Snapdragon

each brand and compare them with 3 different kinds of benchmark test scores (i.e., Greekbench 4.4 64 Bit Single-Core score, Greekbench 4.4 64 Bit Multi-Core score, and Octane V2 total score). As shown in Fig 4.8, we present the results along time line to reveal the fast evolution trend of the chipsets. From the polylines, which refer to the different score results, we can see that the performance of the chipsets has doubled during the past 5 years. We detail the full specifications of chipsets on our website [117]. Besides the analysis of chipsets, we also investigate the clock freqency and RAM size on 167 Android mobile devices, which is released in 2019, and present them on our website [117]. As shown in Fig. 4.9, we can see that the current frequency of new released Android devices are mostly located in 2000~2500MHz and 2500~3000MHz, which refers to common devices and flagship devices respectively. As shown in Fig. 4.10, we can see that the current RAM sizes of new released Android devices are mostly larger than 3GB. The mainstream RAM sizes are 4GB, 6GB, and 8GB, which have a proportion around 72% among the whole specification data. By investigating the hardware performance evaluation results of real devices on chipsets and RAM with the six devices used in our experiments, we can tell that the most of the current Android mobile phones can support

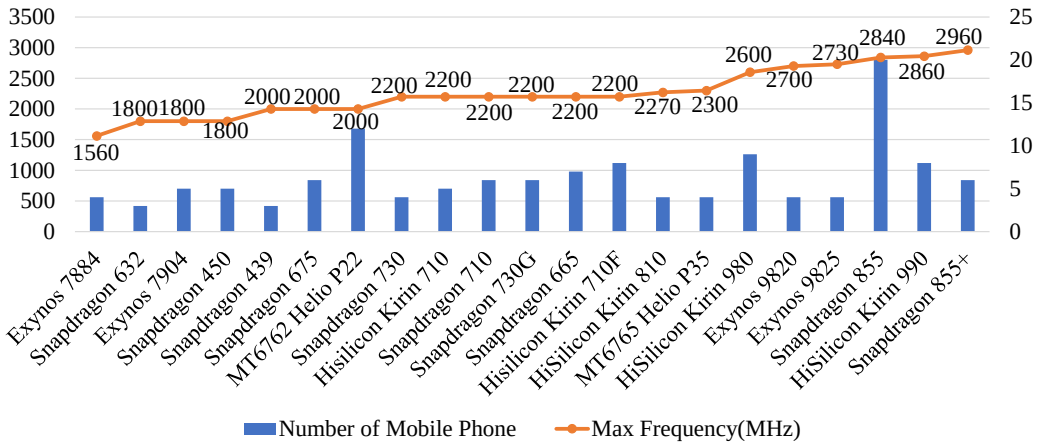


Fig. 4.9: Top 21 chipsets assembled in Android mobile phones released in 2019

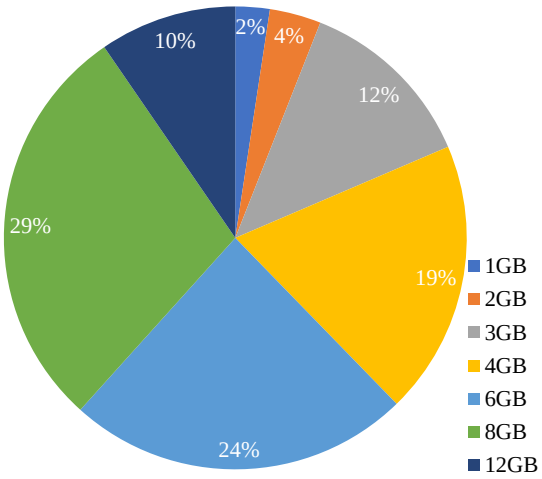


Fig. 4.10: RAM size of Android mobile devices released in 2019

MobiTive smoothly and achieve a responsive detection.

Remark. By collecting and analyzing the specifications of chipsets and devices, we find the evolution trend of chipsets will provide a better performance for MobiTive in the future. Meanwhile, the study on device specifications released in 2019 shows that most new devices will have a performance not worse than our 4 selected common devices.

4.4 Limitations and Discussions

In this section, we discuss the limitations of MobiTive.

Feature selection. As a result of the performance requirement of MobiTive, the limited selected feature categories (i.e., manifest properties and API calls) surely will not cause large overhead when it is working on an Android device. However, the limited two feature types will also provide limited information from the Android malware. If there will be a new malware family, whose malicious behaviors cannot be represented by our selected feature types, the MobiTive may not be able to detect them. In the future, we aim to add more effective feature types with low-performance costs as well.

Meanwhile, we totally agree with the reviewer's view that it is very important to detect new malware families in practice. Actually, neither dynamic nor static methods can fully guarantee the validity of protection against the new malware samples. But we can make some efforts to improve the ability to detect new samples, for example, for the static method, we can extract more robust features to ensure the robustness of trained models; for the dynamic method, we can define more self-adaptive rules to capture more malicious behaviors in real time. A possible solution for detecting more new malware is that combining two types of methods. In the future, we will try to improve the ability to detect new malware by designing a new adaptive method, which is also an open question for this community.

New malware family detection. For any malware detection tool, it is no doubt that detecting new malware families in practice is a very important task. However, neither dynamic nor static methods can fully guarantee the validity of protection against the new malware samples. For example, due to the limited training dataset, MobiTive would have a similar limitation as other static analysis based malware detection systems, which

is different from the dynamic analysis approaches. Specifically, considering a new malware family, the situation may be that the malicious features are totally different from existing data. Consequently, as a result of lack of knowledge, the trained classifier may not be able to make the right decision, although learning-based approaches sometimes have the ability to detect new malware variants. Therefore, in the future, we can make some efforts to improve the ability to detect new samples (e.g., combining static and dynamic methods). For example, for the static method, we can extract more robust features to ensure the robustness of trained models; for the dynamic method, we can define more self-adaptive rules to capture some high risk malicious behaviors in real time. We surely will also try to improve the ability to detect new malware by designing new adaptive methods, which will also benefit the community by discovering the possible techniques on solving this open question.

Against adversarial attack. Indeed, deep learning based systems (e.g., voice/image recognition) will suffer from adversarial attacks [137, 138], so that maintaining the robustness of deep learning based system becomes a challenging topic. However, there are several differences in the deep learning based systems between malware detection task and voice/image recognition. (1) First and most important, unlike voice/image recognition, the adversarial attacks in malware detection cannot break the entire functionality in the applications easily in practice, so that the existing adversarial attacks against malware detection are always generated by manipulating the target malware application with un-triggered code snippets (e.g., dead code) instead of changing real functionalities [139]. Although it is able to generate adversarial samples to evade the classifier and achieve a high miss-classification rate, it is impractical so far, because such attack can be easily detected by leveraging other techniques such as static data flow analysis to delete such features that are introduced by adding dead code from attackers. Meanwhile, it is also evidenced by lacking real adversarial malware samples in the existing researches. (2) Secondly, different from malware detection approaches on other system (e.g., Windows/Linux), our approach abstracts the entire Android app with a limited feature list instead of embedding the whole package program, so that the attackers have to manipulate their malware applications with our defined features to bypass MobiTive. In practice, attackers cannot obtain the accurate feature list easily. Meanwhile, considering

that most of our selected features (i.e., manifest properties and API calls) are defined by the official/trustworthy third-party developers, it is almost impossible to bypass MobiTive as easily as the deep learning based voice/image recognition systems under the restriction of maintaining the functionalities in the malware applications. All in all, adversarial attacks on deep learning based malware have domain-specific challenges compared with image/voice classification, which is also belonging to a new research direction as an open question.

Dynamic behavior analysis. According to our knowledge and an in-depth literature review, static analysis acts an important in past and current cyber security research, and the number of research publications on Android malware detection is also larger than dynamic analysis (static analysis [27, 32, 40, 61–68, 72, 83–85, 140] vs. dynamic analysis [20, 76–80]). Indeed, on a specific given detection task, dynamic behavior analysis may achieve a more accurate result (e.g., lower false positive) than static analysis, however, there are several limitations, which undertake its applicability on specific scenarios, need to be discussed. (1) First and most important, the available scenarios of dynamic behavior analysis based malware detection systems are more limited, because the high cost on computational resources makes dynamic behavior analysis based systems unable to satisfy users' requirements on performance and energy. For example, using performance counter [80] while doing program analysis in malware/bug detection task is widely used. However, unlike traditional windows/linux programs, Android application have a more complicated HCI mechanism. In other word, generating good quality test benchmarks with a good coverage to the corner cases is much more difficult than programs on windows/linux. Assuming we have the ability to obtain the benchmarks, the time cost in generating and executing them will also bring a conflict to the target, which is satisfying user's demand on efficiency. (2) Second, the detection efficiency is highly depend on the coverage of the predefined behaviors. Namely, once the malicious behavior in the target malware is not specifically defined by the detection system, the security of system will be no longer promised. (3) Third, different from MobiTive, dynamic behavior analysis based system may suffer from its working mechanism (i.e., before installation vs. run-time). For example, a social engineering based spyware can easily store the privacy information on the device and trick the user to upload them, as a result of that most

users are not as professional as security researchers. In the end, according to the diverse usage scenarios and targets, we think Android malware detection approaches based on dynamic behavior and static analysis have their own advantages and weaknesses respectively, which both call for research on them.

4.5 Chapter Summary

This work presents MobiTive, a performance-sensitive Android malware detection system on mobile devices as a pre-installed solution. According to the effectiveness of selected features and the efficiency of feature extraction, MobiTive can provide a reliable detection accuracy and fast responsive (i.e., less than 3 seconds on average) detection service on mobile devices directly. To validate the efficiency and reliability, we evaluate MobiTive on six real mobile devices. To provide more insights of this work, we also make an in-depth analysis of the performance trend on over one hundred mobile phones.

Chapter 5

An Efficient Sequence-Based Malware Detection System Using RNN on Mobile Devices

5.1 Introduction

Smartphones have revolutionized our lives for the better in some ways. Besides calling and sending text messages, people are using these devices to watch movies, perform banking transactions, read the news, etc. It is undeniable that these smartphones have yielded many benefits for society, allowing millions of people to stay connected through the Internet. Consequently, it has also drawn the attention of malware authors to disseminate their malware on the application markets (e.g., Google Play Store). However, unlike the application market (i.e., App Store) for Apple iOS, the protocol for uploading an application on the Android application market is not that stringent. Therefore, there is a demand for an effective malware detection system running on the device to address the above security problem.[1]

Currently, majority of the machine learning-based malware detection systems performed their analysis on the server side [29, 31], until Drebin [27], which is a lightweight method for detection of Android malware that enables identifying malicious applications using machine learning directly on the smartphone. Recently, researchers have been looking at ways to implement effective solutions with deep learning techniques on the

[1]The work in this chapter has been published in [141].

device-end. Due to the performance limitations on mobile devices, researchers tend to extract limited syntax features to meet certain time constraints [114, 117]. Although using syntax features without semantics (e.g., order, position) has achieved a relatively high accuracy, it will consequently fail to maintain a more robust detection system by providing necessary information to represent certain malicious behaviors. Therefore, there is a demand to research into ways to effectively represent meaningful and robust features such that it contains more semantics on the malicious behaviors with limited performance overhead on mobile devices. Unlike syntax feature based learning approaches, the feature input for sequence-based learning approaches provides not only the existence of each determined syntax feature, but also represents the semantics corresponding to certain behavior patterns [83, 142–144]. However, due to the complexity of sequence-based feature, traditional server-end sequence-based learning approaches failed to satisfy the demand of run-time performance when it comes to detection on mobile devices. To address these problems, we intend to provide an efficient sequence-based malware detection system using deep learning on mobile devices.

In this work, we propose SeqMobile, which adopts behavior-based sequence features and customized deep neural networks to provide an effective and efficient malware detection service on Android devices. To enhance the performance of SeqMobile, we propose a series of performance optimization methods that can effectively reduce the training and prediction time for sequence-based approaches. In the experiments, we first summarize and propose 8 feature categories (e.g., combination of permissions, intent filters, API sequence and intent sequence) and investigate their corresponding performance with different deep neural networks. We then perform an evaluation using accuracy and prediction time as metrics to decide on a suitable network configuration. After that, we accept the pre-trained model that yields the best results and deploy it onto Android devices. To ensure that our pre-trained models are compatible with Android device, we convert our pre-trained models into lightweight TensorFlow Lite models. In our proposed system, we prioritize time cost over accuracy such that lower-end devices can choose to trade off less than 1% of the classification accuracy for lower prediction time. Overall, through our performance optimization methods, SeqMobile can achieve a relatively higher classification accuracy (i.e., 97.85%) as well as lower feature extraction and prediction time cost (i.e., <5 seconds).

In this work, we make the following contributions:

- We propose an efficient sequence-based malware detection system, which adopts behavior-based sequence feature and customized deep neural network to provide an effective and efficient malware detection service on Android devices.
- We present a systematic approach to directly extract the semantic feature sequence, which can provide information of certain malicious behaviors, from binary files under a certain time constraint. Thereby, achieving a relatively higher classification accuracy (i.e., 97.85%).
- We propose a method to remove repetitive elements in sequences and further evaluate how it can affect the overall performance of our malware detection system. Results has shown that our removal method significantly enhances the training and prediction performance with insignificant effects on the accuracy. To our best knowledge, this is the first comprehensive study on how removing repetitive elements in sequences can affect training and prediction performance in sequence-based learning approach on mobile devices.
- We conduct an evaluation on the state-of-the-art mobile-end model optimization toolkit provided by TensorFlow for our proposed sequence-based learning approach. The evaluation results can serve as a guidance for other mobile-end sequence-based learning approaches.

5.2 System Design of SeqMobile

5.2.1 Overview

Fig. 5.1 demonstrates the overview of SeqMobile, which contains two major phases (i.e., learning phase and deployment phase).

Learning phase, which is done on a server, consists of *feature preparation, network training, and model conversion*. In the feature preparation step, we focus more on extracting features as sequences such that it can provide more semantics to help distinguish between malicious and benign behaviors inside Android applications. First, a set of feature dictionaries will be constructed (step ①). Next, we extract the feature sequences

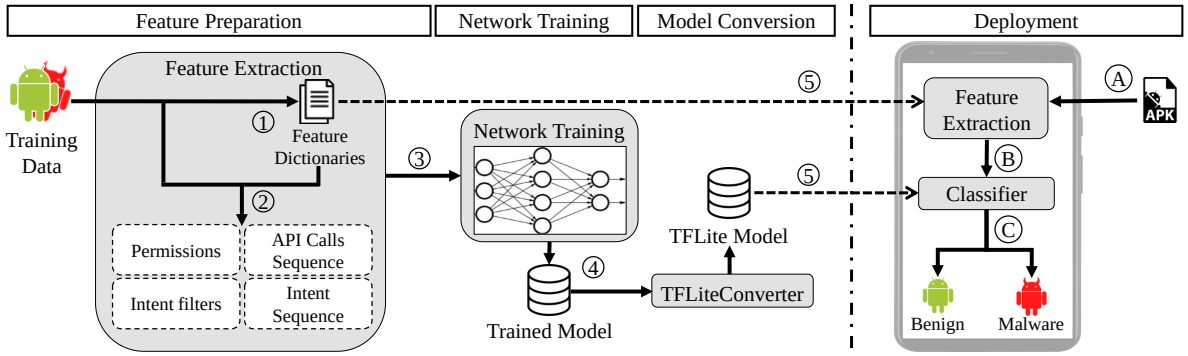


Fig. 5.1: Overview of SeqMobile

and use the constructed dictionaries to filter out the redundant elements (step ②). After that, we represent each element in the filtered sequence with a unique integer identifier and pass it into our proposed network for training (step ③). Once a trained model is obtained, it will be converted into a *TensorFlow Lite* [56] model (step ④), which can then be loaded onto mobile device and used with the Tensorflow Lite interpreter to do inferences (step ⑤). In the midst of conversion, Tensorflow provides user an option to enable quantization, which is a technique that can further reduce the size of the pre-trained model with minimal effect on the accuracy. However, we do not perform quantization in SeqMobile since our experiments in § 5.3.3.2 shows that it is more advantageous for our proposed network.

Deployment phase will first perform feature extraction from the target APK and use the constructed dictionaries from the learning phase to filter out the redundant elements (step Ⓐ), when an Android package (APK) is downloaded into the device. To ensure efficiency, a small but crucial part of the feature extraction module is implemented using native code, where a performance gain can be observed even on lower-end devices. After that, the extracted sequence will be fed into the classifier module to determine whether the target application is benign or malicious (step Ⓑ). Finally, SeqMobile will output the classification results to the user (step Ⓒ).

5.2.2 Feature Preparation

In the feature preparation step, SeqMobile mainly focuses on selecting features from the AndroidManifest.xml and DEX file (classes.dex). In order to determine the features

Table 5.1: An example of API mapping

API	Description
API_0	Ljava/net/HttpURLConnection;->connect
API_1	Ljava/io/BufferedReader;->readLine
API_2	Landroid/net/Uri;->parse
API_3	Landroid/content/ContentResolver;->query

used in SeqMobile, we perform a static analysis and select 4 kinds of informative and semantic feature sets that can potentially help to distinguish between malware and benign samples. For each feature set, we rationalize the reason of our selection with concrete examples. Besides, to remove the effect of certain elements that may not be helpful in providing information, we have built dictionaries (step ①) for the purpose of filtering out those elements. Each element in the filtered sequence will then be represented as a unique integer value so that it can be fed into our neural network for training. In addition, we also experiment with different combinations of feature sets to determine which combination yields the best accuracy with acceptable extracting performance. Based on the results, we selected 4 feature sets as our input to the neural network. The details of the experiment can be found in § 5.3.2.1.

5.2.2.1 Feature Selection

As a result of the strong dependency between the detection accuracy of learning-based approaches and the coverage of malicious information, the features that are able to represent more semantics in the malicious behaviors will have a higher probability to achieve a better result. Thus, based on this concept, we determine to accept two *non-sequential features*: Permission, Intent filters, and two *sequential features*: API call sequence, Intent sequence depended on our analysis against Android malware.

- $N^{\{Perm\}}$: Permissions are defined in the AndroidManifest.xml file. Previous studies [145–147] have shown that majority of the malicious application tend to request dangerous permissions as compared to benign applications. This indicates that including permissions as part of our feature set can potentially help us distinguish malware and benign samples.

- $N^{\{Intent\}}$: Intent filters are defined inside the AndroidManifest.xml file that specifies what type of intents a component (e.g., Activity) would like to receive. Intent filter makes it possible for other applications to directly start the activity by sending out the defined intent message. An example of how malicious application abuse the intent filter is that they usually listen for the BOOT_COMPLETED intent, which is sent after successfully booting up the mobile device, to start their malicious activities [147].

- $S^{\{API\}}$: Representing API calls features in an unordered manner is usually sufficient to provide enough information on the behaviors of an Android application. However, if the API calls are represented in a sequential manner, it can provide us with additional semantics amongst the API calls. Two behaviors are defined using sequence of API calls, where $S_0^{\{API\}}$=$\{API_0, API_1, API_2, API_3\}$ and $S_1^{\{API\}}$=$\{API_2, API_3, API_0, API_1\}$. The details of each API can be found in Table 5.1. The behavior $S_0^{\{API\}}$ represents the action of communicating to the internet followed by reading of SMS inbox messages. This is a typical behavior for instant messaging applications where they try to verify your mobile number. In contrast, $S_1^{\{API\}}$ represents the action of reading of SMS inbox messages followed by a communication to the internet, which is possibly a malicious behavior where an adversary attempts to retrieve your inbox messages.

- $S^{\{Intent\}}$: Besides picking API call sequence as a part of our feature set, we also accept the sequential intent as a supplemental feature for the API call sequence with the following two reasons. Firstly, when API call sequence is included as our feature set, the function parameters are not taken into account, thus, generic API calls such as Landroid/content/Intent;->init or Landroid/content/Intent;->setAction is unable to provide information on the purpose of invocation unless the corresponding *string* parameter is included. Secondly, relying on intent filters is not sufficient as it represents what type of intent the component is looking out for. Oftentimes, intents can be omitted from the manifest file. One typical example is that malware can make use of the ACTION_CALL intent to call premium rate numbers while the user is not looking [148]. In addition, the ACTION_CALL intent does not need to be defined in the manifest file. Thus, we include the *string* parameter of the generic API calls mentioned above in a sequential order and intertwine it with the API call sequence (e.g., $S^{\{API,Intent\}}$=$\{API_1, IN_0, API_2, API_0, IN_2, API_3\}$).

Table 5.2: Vocabulary size of each dictionary

Dictionary	Vocabulary size
Permissions	324
Intents	262
API calls	2,288

5.2.2.2 Feature Dictionary Construction

Since there are certain elements that may not be related to potential malicious behaviors in the raw data (i.e., .xml and .dex files) within each feature set, a set of feature dictionaries are necessary in the feature extraction step. Thus, based on the static analysis result on potential malicious behaviors, a set of feature dictionaries has been constructed (step ①).

Permission and intent dictionary. To build the permission and intent dictionary, we refer to the Android source code which are predefined by Google developers [149, 150] to retrieve all the permissions and intent filter values.

API call dictionary. To build the API call dictionary, we conduct a data-driven analysis to collect API calls from more than 60,000 real-world applications and pruned the result set by removing self-defined API calls as well as uncommon third-party API calls [117].

In summary, we constructed 3 feature dictionaries and the total number of vocabulary in each dictionary can be found in Table 5.2.

5.2.2.3 Feature Extraction

To generate and formalize the selected features (step ②), we first propose a repetitive elements removal method, which can boost the performance of our system with little effect on the accuracy (discussed in 5.3.2.2), by reducing the length of feature sequence for the selected sequence-based features. Next, with the shorten feature sequence, we transform it into a numeral sequence such that it will be suitable to be fed into the neural network.

Raw string based feature representation. To extract $S^{\{API\}}$, we disassemble the DEX file and look for instructions starting with the *invoke-** opcode. At the same time, we also include the $S^{\{Intent\}}$ feature into the sequence by looking out for instructions

that starts with the *const-string* opcode. For each of the instructions found, we concatenate them to form a *string* based sequence, S_d. An example of such sequence will be $S^{\{API,Intent\}}$=$\{API_0, API_1, IN_1, IN_2, API_3\}$ where IN_1 and IN_2 are intent values. After that, permissions and intent filters will be extracted and matched against the corresponding dictionaries to construct the non-sequential feature, $N^{\{Perm\}}$ and $N^{\{Intent\}}$. Finally, we concatenate the extracted sequence, S_d, together with $N^{\{Perm\}}$ and $N^{\{Intent\}}$ to form the final sequence, S_f. Based on the experiment results presented in § 5.3.2.1, we select 4 features as the final feature set namely, $N^{\{Perm\}}$, $N^{\{Intent\}}$, $S^{\{API\}}$, and $S^{\{Intent\}}$.

Repetitive elements removal. Oftentimes, there can be repeated elements in a sequence. An example of how this can occur is that a developer may define two different methods, M_1 and M_2, that execute very similar tasks. We define API_i and IN_i as API calls and intent values which are defined in the dictionaries and S_API_i as self-defined API calls not found in the API dictionary. Consider M_1= $\{IN_0, API_0, S_API_0\}$ and M_2=$\{IN_0, API_0, S_API_1, API_1\}$. After the extracted sequence is filtered against the respective dictionaries, the resulting sequence is $\{IN_0, API_0, IN_0, API_0, API_1\}$, which is basically a repetition of $\{IN_0, API_0\}$ followed by an API_1. If we view it from the perspective of a text sentiment analysis problem, having the sentence "this movie is great, this movie is great" does not change the polarity of the sentence, likewise, having repetitive elements in the sequence will have insignificant effects (§ 5.3.2.2) on the polarity (i.e., malicious or benign). Thus, the resulting sequence after removing the repetitive elements will be $\{IN_0, API_0, API_1\}$.

Dictionary identifier assignment and integer based sequence representation. In order to transform the string based sequence into an integer sequence, we first assign each dictionary element with a unique integer identifier and store the pair in a look up table, T_l. By referring to the look up table, we represent each element in S_f with their respective integer identifier. After that, we can feed it into the embedding layer of the proposed network Table 5.3 (step ③).

5.2.3 Deep Learning Model Construction

To discover the usability of different neural networks for our selected feature sets, we present 6 basic neural networks to train our classifier (i.e., single layer LSTM/GRU,

Table 5.3: Network architecture - Bi-LSTM

Layer	Output Shape
Embedding	(None, None, 128)
Bidirectional (LSTM)	(None, None, 512)
Batch Normalization	(None, None, 512)
GlobalMaxPooling1D	(None, 512)
Dense (ReLU)	(None, 64)
Dense (ReLU)	(None, 32)
Dense (softmax)	(None, 2)

stacked LSTM/GRU, and Bi-LSTM/GRU) (details on our website [151]). By comparing the accuracy, we accept the Bi-LSTM, which yields the highest accuracy, as basic network and perform a customization on the architecture to further enhance the accuracy. The architecture of the customized Bi-LSTM network is shown in Table 5.3.

5.2.3.1 Sequence Padding

The length of S_f varies from one application to another, hence, we pad the input sequence to ensure that the length is consistent with the required input length of the network, L. If the length of S_f is shorter than L, we perform post-zero padding to S_f till it reaches L. If the length of S_f is longer than L, we truncate S_f till L.

5.2.3.2 Customized Deep Neural Network Architecture

To train our malware classifier, we present a customized Bi-LSTM network. As shown in Table 5.3, the first layer is an embedding layer. Due to our large vocabulary size, representing our input sequence using one-hot encoding results in a sparse vector, which is not memory efficient during training and prediction. By incorporating an embedding layer, it can help to reduce the dimensionality of our feature vectors where each element of our integer input sequence is represented as a lower dimension fixed sized vector. The second layer is a Bi-LSTM layer, which has the ability to preserve information from both the forward and backward along the sequence, thus allowing the network to understand the contextual information better and results in a more comprehensive learning of the problem. The third layer is a batch normalization layer, it has been shown that incorporating a batch normalization layer to the network can reduce the internal

covariate shift [152, 153]. After that, a global max-pooling function is used to capture the most important factor. Then, the output from the global max-pooling function will pass through the 2 fully connected layers with Rectified Linear Unit (ReLU) activation function followed by a fully connected layer with a softmax activation function. Finally, the output from the last fully connected layer determines which class the input sequence belongs to.

5.2.4 Model Conversion and Quantization

In order to make our pre-trained model compatible with mobile devices, we convert the pre-model into a TensorFlow Lite model using TFLiteConverter [154]. We have also conducted experiments and decided not to perform quantization even though it has advantages such as prediction time and model size reduction. By not applying quantization to our model, we are able to achieve dynamic prediction time where the prediction time cost is dependent on the extracted sequence length (details in § 5.3.3.2).

5.2.5 Real Time Detection System

Before conducting a real-time detection on device, the feature dictionaries and converted model will be loaded into the feature extraction and classifier modules respectively (step ⑤). Once a new APK file is received, SeqMobile first performs feature extraction on the target APK with the constructed dictionaries from the learning phase (step Ⓐ). To improve the overall performance of SeqMobile, we directly extract the selected features from the binary files and perform repetitive sequence removal such that the sequence length can be reduced. In addition, we also optimized our feature extraction module by incorporating some native code in the implementation. Next, we perform truncation to the input sequence but not padding. Results from our experiments shows that without padding, we can not only achieve the best accuracy, but also a shorter prediction time. Finally, the classifier module will determine from the extracted features whether the Android application is malicious or benign.

Table 5.4: Dataset

Class	Source	Quantity
Malicious	Contagio	327
Malicious	Drebin	5,339
Malicious	Genome	1,253
Malicious	Pwnzen	774
Malicious	Virusshare	14,797
Benign	Google Play Store	22,490
Total Malicious	-	22,490
Total Benign	-	22,490
Total	-	44,980

5.3 Evaluation

In this section, we present four sets of experimental studies. We aim to determine: (1) the detection performance of different networks across different combination of feature categories; (2) the training and prediction performance gain through our repetitive elements removal method; (3) the feature extraction performance across different mobile devices; and (4) the performance comparison between the quantized and non-quantized dynamic RNN model. Finally, we briefly compare the performance of our approach with two other previous work [114, 117].

5.3.1 Experiment Environment and Dataset

Environment. All experiments are conducted on an Ubuntu server with Intel Xeon E5-2699 V3 CPUs, NVIDIA GeForce RTX 2080 Ti GPU and 6 different Android devices, which consists of 3 flagship devices (i.e., Samsung Note10+, S10+, S9+), 1 common device (i.e., Samsung S7), and 2 low-end devices (i.e., Samsung J2 Pro and HTC ONE A9).

In our server-end tasks, Java is our choice of language for implementing the feature extraction module and TensorFlow 2 [128] is chosen as our deep learning framework. For feature extraction, we use additional tools such as *dexdump* [155] and AXML-Printer2 [156]. As for the TensorFlow Lite converter, a specific TensorFlow nightly build (2.2.0.dev20200430) [157] which supports our proposed network architecture is used.

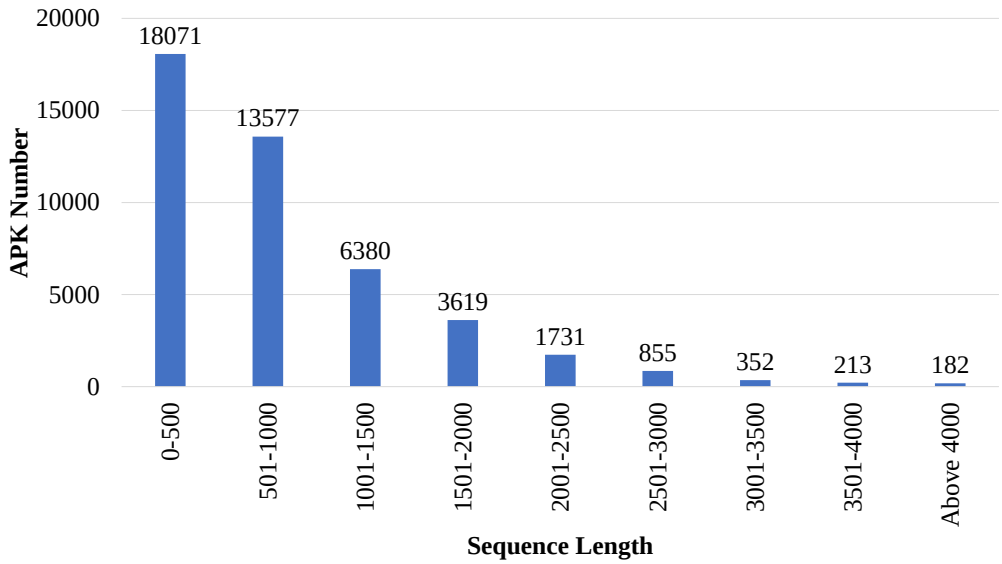


Fig. 5.2: Distribution of extracted sequence length

Dataset. To evaluate SeqMobile, we collected 44,980 Android applications samples which can be divided into two classes; benign and malicious. We crawled the benign samples from Google Play Store, while the malicious set is composed of samples from different sources such as Drebin [27], Genome project [10], Contagio Mobile [131], VirusShare [158], and Pwnzen Infotech Inc. [29, 31]. The breakdown of the dataset is shown in Table 5.4. To split our dataset, we randomly select 70% of the samples from each class for training, 15% for validation and 15% for testing.

5.3.2 Effectiveness Evaluation of Feature Selection, Deep Neural Networks, and Repetitive Pattern Removal in Feature Preparation and Network Training Phases

We evaluate the effectiveness of SeqMobile in the learning phase from the following aspects: (1) the accuracy across different feature categories, length, and deep neural networks and (2) the effect on training accuracy and performance brought by our repetitive pattern removal method.

5.3.2.1 Accuracy Comparison Across Feature Categories and Deep Neural Networks

To determine the best training configuration, we set up an experiment across three aspects

Table 5.5: Detection results of best feature category combinations across difference networks

Network	Feature Combination	Seq. Len.	Accuracy	Precision	Recall
Single LSTM	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API\}}$	600	96.47%	97.66%	95.23%
Single GRU	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API\}}$	1,500	96.78%	96.99%	96.56%
Stacked LSTM	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API\}}$	600	96.49%	97.86%	95.05%
Stacked GRU	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API\}}$	1,500	96.90%	97.42%	96.35%
Basic Bi-LSTM	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API\}}$	1,500	96.98%	97.51%	96.41%
Basic Bi-GRU	$N^{\{Perm\}}$ and $S^{\{API,Intent\}}$	1,900	96.75%	97.47%	96.00%
Customized Bi-LSTM	$N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API,Intent\}}$	1,700	97.85%	97.89%	97.81%
Customized Bi-GRU	$S^{\{API\}}$	1,500	97.67%	98.11%	97.21%

(i.e., feature categories, sequence length, and network types). To find out the most suitable network, we first evaluate the detection accuracy of 6 different basic network configurations across 8 different feature categories (details on our website [151]), which apply $S^{\{API\}}$ as the basic feature and combined with $N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{Intent\}}$ respectively. In addition, as shown in Fig. 5.2, majority of the APKs in our dataset have sequence length ranging up to 2,000. Thus, we also train each network across different sequence length to determine the appropriate sequence length that yields the best accuracy. To choose the range of sequence length to experiment on, we progressively increase the sequence length until no significant improvement on the accuracy can be observed. In total, according to the distribution of extracted sequence length, we select 6 different values ranging from 300 to 1,900 to train the networks. As shown in Table 5.5, the basic Bi-LSTM network achieves the highest accuracy (i.e., 96.98%) out of the other basic networks. To further improve the detection accuracy, we customize the basic Bi-LSTM to our best effort such that there is a significant improvement on the accuracy. We first progressively increase the number of parameters (i.e., increase LSTM units and add fully connected layer) in the network. Consequently, a longer training duration can be observed due to the increased network parameters. Thus, we added a batch normalization followed by *GlobalMaxPooling1D* layer to reduce the training duration. Our experiment results shows that by configuring the network as shown in Table 5.3, the feature set combination that achieves the best accuracy is $N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API,Intent\}}$, which achieves an accuracy of 97.85% when the sequence length is 1,700 (shown in Fig. 5.3).

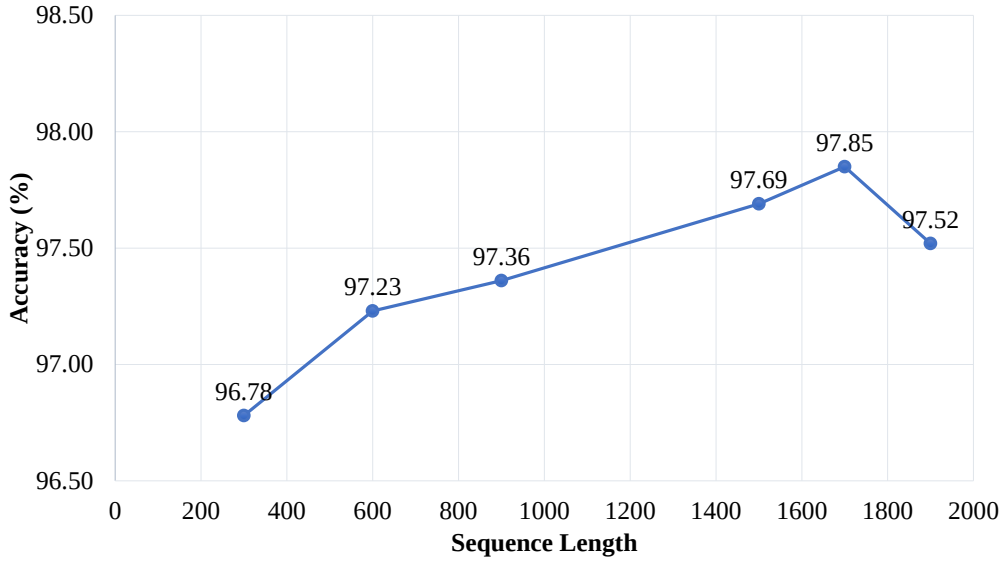


Fig. 5.3: Accuracy change across different sequence length on $N^{\{Perm\}}$, $N^{\{Intent\}}$, and $S^{\{API,Intent\}}$

Table 5.6: Average sequence length

Dataset	Original Length	After Removal
Benign	2,789	1,060
Malware	1,249	598

5.3.2.2 Effect of Repetitive Pattern Removal

We conduct experiments to investigate how removing of repetitive sequence will benefit the performance of the system. In the experiment, we use the following metrics: (1) accuracy and (2) training duration to evaluate our method. The results show that (1) removing the repetitive pattern in a sequence has insignificant effects on the learning ability of the neural network; (2) it can help improve the training and prediction performance in terms of time cost.

Average number of removed repetitive elements. To show the necessity of our repetitive elements removal method, we first calculate the number of repetitive elements removed in each dataset (i.e., benign and malware) to get a rough estimation of the proportion being removed by comparing with the original average sequence length. As shown in Table 5.6, the average sequence length is reduced by approximately 62% (1,060

vs. 2,789) in the benign dataset. Similarly, the average sequence length is reduced by approximately 52% for the malware dataset. When taking into account for both datasets, the sequence length is reduced by 57% on average. Thus, based on the results, we take 0.6 as an approximate proportion of the repetitive elements in the original sequences and define equation (5.1) to calculate the estimated original sequence length for each non-repetitive sequence length.

$$Seq_{original} = \frac{Seq_{non-repetitive}}{1 - 0.6} \tag{5.1}$$

Accuracy comparison between non-repetitive and original sequence. To discover the effect of our repetitive elements removal method on model accuracy, we define the following 2 models, M_1, which adopts the sequences without repetitive elements as its input, $Seq_{non-repetitive}$, and M_2, which is trained using the original sequences that contain repetitive elements, $Seq_{original}$, and train them with sequence length ranging from 100 to 750. From the previous experiment, the input sequence length is reduced by 57% on average after removing repetitive elements, as a result, with the same sequence length, $Seq_{non-repetitive}$ will provide more information than $Seq_{original}$. To compare the accuracy of M_1 and M_2, we use the equation in (5.1) to estimate the corresponding original sequence length for each $Seq_{non-repetitive}$, and assign them as a control group. For example, if the accuracy of M_1 at sequence length 200 is 96.15%, we will apply equation (5.1) to calculate the length of $Seq_{original}$ (i.e., 500). From the line chart in Fig. 5.4, the accuracy for M_2 is 96.13% when the sequence length is 500, which is very close to the accuracy for M_1 (i.e., 96.15%) at sequence length 200. Similarly, the accuracy at M_1 when the sequence length is 300, is very close to the accuracy of M_2 when the sequence length is at 750 (96.78% vs. 96.74%). This shows that by removing repetitive elements in a sequence, it will have very little effect on the learning ability of the network.

Performance improvement in network training. To find out the performance improvement of our repetitive elements removal method in the network training phase, we also provide a comparison on training time between each grouped models with non-repetitive and original sequences respectively. From the histogram in Fig. 5.4, the training time for M_2 is 28.42 minutes when the sequence length is 500. Comparing with the time for M_1 (i.e., 13.33 minutes) at sequence length 200, our repetitive elements removal

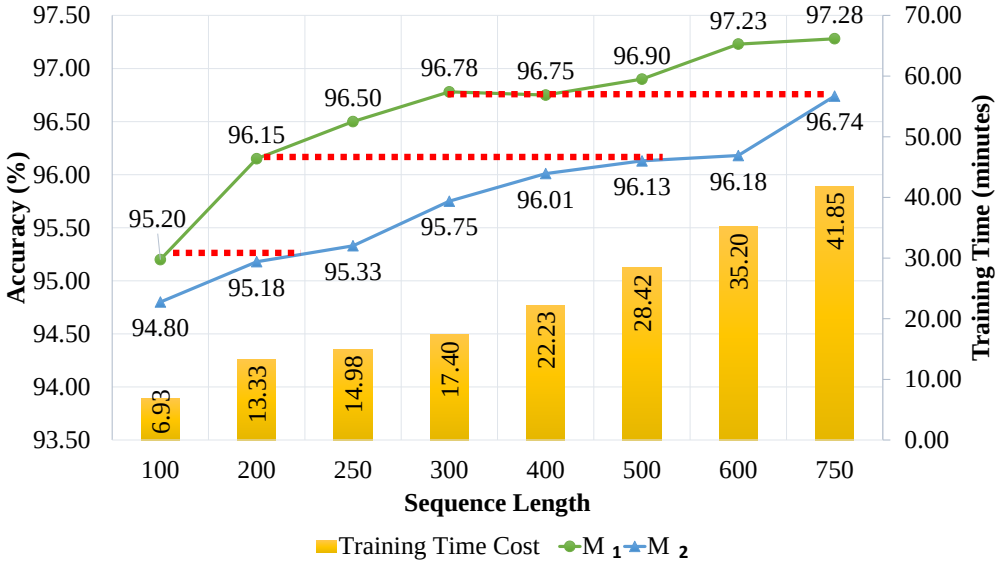


Fig. 5.4: Training time across different sequence length and accuracy comparison between M_1 and M_2

method improves the training performance by around 53.1%. Similarly, comparing the training time for M_1 when the sequence length is 300, we observe a much shorter training time for M_2 when the sequence length is at 750 (17.40 minutes vs. 41.85 minutes). This provides a strong evidence for proving that our repetitive elements removal method benefits a lot in the training performance.

5.3.3 Performance Evaluation and Optimization of Feature Extraction and Detection on Android Mobile Devices

We evaluate the performance of our device-end modules (i.e., feature extraction and prediction modules) separately. For the feature extraction module, we conduct experiments to determine (1) the performance of extraction time on mobile devices, by comparing the extraction time of our final feature combination (i.e., $N^{\{Perm\}}$, $N^{\{Intent\}}$ and $S^{\{API,Intent\}}$) across 6 different APK sizes; and further evaluate (2) the performance gain of the feature extraction module where certain parts are rewritten in native code. For the prediction module, we assess the performance gain across 2 aspects, which are input sequence and deployed model configuration.

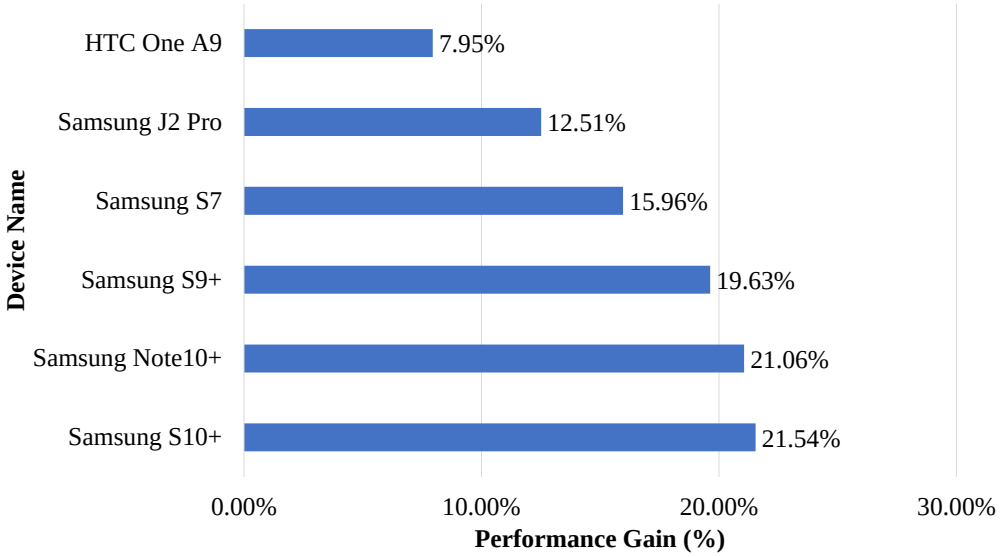


Fig. 5.5: Performance gain with JNI implementation

5.3.3.1 Performance Comparison of Feature Extraction between Implementing with JNI and Java Language on Android Devices

To discover the potential performance optimizations in feature extraction on real devices, we implement part of the feature extraction module that involves *string* manipulation in C++ and use JNI to interact with the native implementation. Based on the previous experiment, we select the feature set combination that yields the highest accuracy (i.e., $N^{\{Perm\}}$, $N^{\{Intent\}}$ and $S^{\{API,Intent\}}$) and measure the time cost between the Java and JNI implementation. The APKs used in this experiment are handpicked, with file sizes ranging from 5MB to 50MB. We then calculate the performance gain between the implementations using equation (5.2).

$$Performance\ gain = (1 - \frac{Timecost_{JNI}}{Timecost_{Java}}) \times 100\% \tag{5.2}$$

As shown in Fig. 5.5, the JNI implementation can improve the performance of the feature extraction module by approximately 21% on flagship phones. Even on low-end devices such as HTC ONE A9, a 7.95% increase in performance can be observed.

A detailed chart of the feature extraction (JNI implementation) time cost across different devices is shown in Fig. 5.6. We observe that flagship phones such as Samsung S10+, Samsung Note10+ and Samsung S9+, are able to extract the features faster than

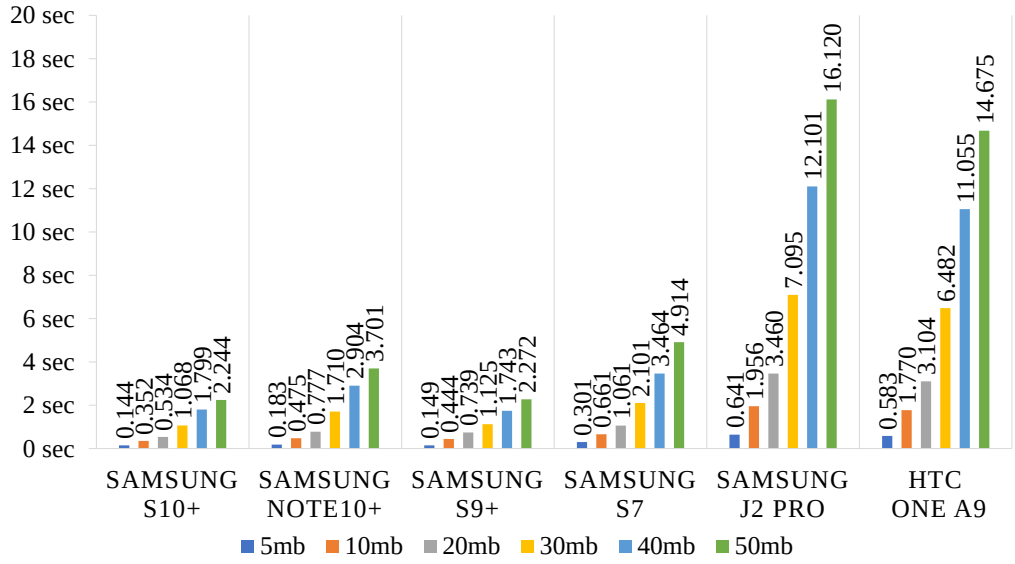


Fig. 5.6: Feature extraction time cost for different APK sizes across different devices

common and lower end devices (Samsung S7, Samsung J2 Pro, and HTC ONE A9). The time cost to extract 5MB applications on the flagship phones is between 0.144 seconds and 0.183 seconds. While on common and low-end devices, it takes approximately 0.301 seconds to 0.641 seconds to extract the features. Similarly, for 50MB applications, the flagship phones outperform the low-end devices (3.701 seconds vs. 16.120 seconds).

5.3.3.2 Performance Optimizations in Prediction on Android Devices

To provide a guidance for improving the performance of sequence-based learning approaches, which may also accept RNN as their computational layer, in the real-time prediction on mobile devices, we conduct two experiments from different aspects. (1) We investigate the performance optimization, which brought by our proposed repetitive pattern removal method, by comparing the prediction time between the original and non-repetitive sequence inputs. (2) We evaluate the state-of-the-art model optimization toolkit provided by TensorFlow for our sequence-based approach. In the second experiment, we first conduct a preliminary investigation to determine the characteristic differences (i.e., model size, input requirements, etc.) between the quantized and non-quantized models. Based on the findings, we further investigate the influence of quantization by comparing the time taken to predict sequences of different lengths between the

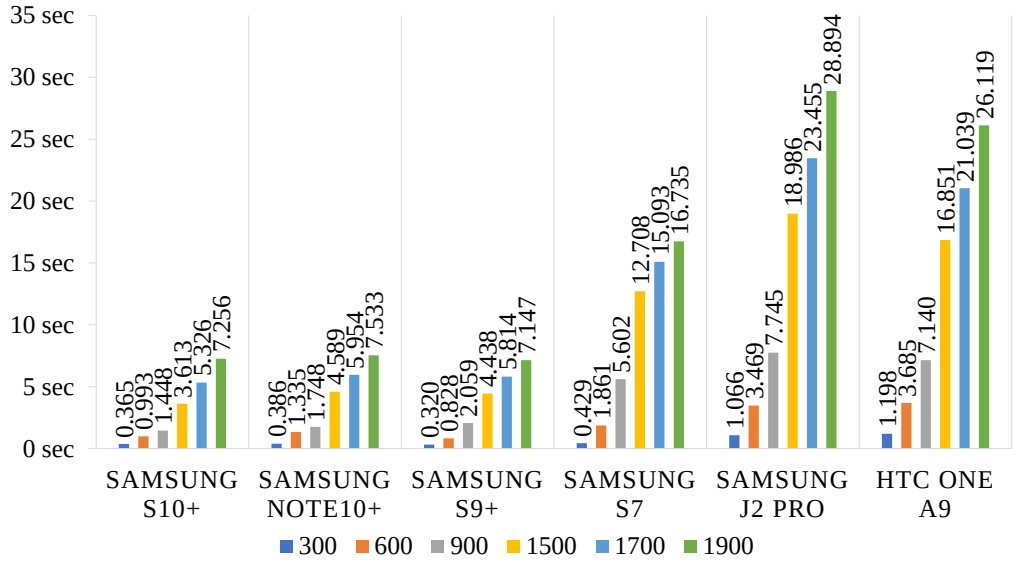


Fig. 5.7: Prediction time cost for different sequence lengths across different devices

quantized and non-quantized dynamic RNN models.

Repetitive pattern removal. To discover the effect of our proposed repetitive element removal method in prediction on real devices, we conduct an experiment to measure the prediction time for 6 different sequence lengths across 6 different devices. In this experiment, we randomly pick 5 APKs from each sequence length category (i.e., 300, 600, 900, 1,500, 1,700, and 1,900).

As shown in Fig. 5.7, flagship devices such as Samsung S10+ takes approximately 2.72 times longer (0.365 seconds vs. 0.993 seconds) to predict a sequence that is double the original length (e.g., 300 vs. 600). Similarly for low-end devices such as Samsung J2 Pro and HTC ONE A9, the prediction time is approximately 3.25 times longer (1.198 seconds vs. 3.685 seconds) to predict a sequence that is double the original length. We observe that it takes at least a twofold increase in prediction time cost to predict a sequence that is twice the original length. Based on our experiment in § 5.3.2.2, we observe that our method reduces the sequence length by approximately 57%, which directly translates to an improvement in prediction time of at least twofold. Apart from the removal method, different from the traditional sequence-based learning approaches that work on server end, approaches on real devices, which may have a strong performance limitation, should take

Table 5.7: Non-quantized and quantized model size comparison

Seq Len.	Non-quantized(MB)	Quantized(MB)
300	7.62	2.97
600	7.62	3.12
900	7.62	3.27
1,500	7.62	3.56
1,700	7.62	3.66
1,900	7.62	3.75

the input sequence length of their defined networks as an important factor to optimize their real-time performance on device.

Model quantization. To investigate the potential performance oriented influence and their corresponding factors of quantization in our proposed system, we first conduct a preliminary investigation to figure out the main differences between a quantized and non-quantized model. In this experiment, we first compare the model size of the quantized and non-quantized models across different sequence length. As shown in Table 5.7, when the sequence length is 300, the model size is reduced by 61% (7.62MB to 2.97MB). However, as the sequence length increases to 1,900, the model size is reduced by approximately 50.8% (7.62MB to 3.75MB). From the results, it is evident that the size of the quantized model is dependent on the sequence length (i.e., longer sequence length will constitute to a larger file size), while the size for the non-quantized model remains constant across different sequence length.

Next, we also conduct experiments to compare the accuracy between the quantized and non-quantized model against the pre-trained model. The accuracy of quantized and non-quantized model remains 97.85%, which is same as the pre-trained model. Otherwise, we also notice that if the input requirements of a network is designed to allow variable length input (e.g., dynamic RNN), quantizing the model will remove the flexibility for allowing variable length input (i.e., inputs will be static fixed length). On the contrary, the input requirements remain unchanged if quantization is not applied. In the event where a quantized model is deployed, padding or truncation is required to ensure that the input sequence is of a certain length.

Dynamic RNN. Since our dataset consists of sequences of varying lengths, designing our network to use dynamic RNN will be beneficial in the deployment phase. Based on

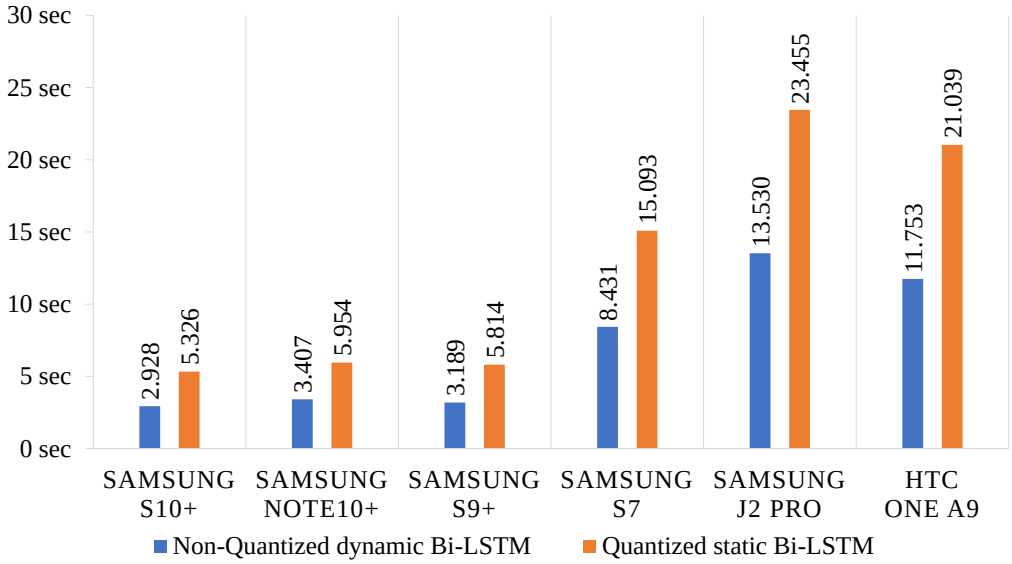


Fig. 5.8: Average prediction time between non-quantized dynamic Bi-LSTM and quantized static Bi-LSTM across different devices

the findings from the preliminary investigation, we observe that quantizing our model is equivalent to using a static RNN model, hence, we refer to the quantized model as quantized static Bi-LSTM model and non-quantized dynamic Bi-LSTM for the non-quantized model.

We conduct an experiment to determine if quantizing our dynamic RNN will improve the performance of our system. In this experiment, we apply the same test set from our previous experiment, which contains 30 APKs with 6 different sequence length. Next, we accept the pre-trained model with the highest accuracy to compare the average prediction time between the quantized static Bi-LSTM and non-quantized dynamic Bi-LSTM models. As shown in Fig. 5.8, we observe that the average prediction time for the non-quantized dynamic Bi-LSTM model is much faster than the quantized static Bi-LSTM model. On flagship phones such as Samsung S10+, the average time taken to predict an extracted sequence is 2.928 seconds when using the non-quantized dynamic Bi-LSTM model. On the contrary, it takes 5.326 seconds on average to predict with the quantized static Bi-LSTM model. The average time cost is lowered by approximately 45% across all devices. The root cause of this observation is expected as padding or truncation is required for the quantized model to make the input length consistent, i.e., 1,700, which

Table 5.8: Comparison of SeqMobile against previous work

Systems	Features used	Accuracy(%)	Time cost(s)
MobiDroid	Opcode sequence API calls 3 Manifest properties	97.35%	17.76
MobiTive	API calls 3 Manifest properties	96.75%	0.46
SeqMobile	API and intent sequence 2 Manifest properties	97.85%	4.16

constitutes to a higher average prediction time. Based on the results, we decide not to quantize our pre-trained model as it brings about more advantages for our proposed network. The allow for variable length input in the non-quantized dynamic Bi-LSTM model has enable us to achieve a dynamic prediction time as no padding is required (i.e., the time cost per prediction is dependent on the extracted sequence length). On the contrary, a fixed length input is required for the quantized static Bi-LSTM model, where each sequence is padded or truncated to a certain length. By doing so, it constitutes to a consistently higher time cost per prediction. Although the model size is reduced by half (7.62MB to 3.66MB) after quantization, compromising time cost for model size is not a feasible option for our performance-sensitive malware detection system. Hence, for any sequence based performance-sensitive learning approaches, which uses RNN as the basic computational layer, dynamic RNN is currently an important option to optimize their system performance instead of quantization, which is widely adopted.

Currently, TensorFlow does not support quantization for models that accept variable length input (e.g., dynamic RNN) [116]. During quantization, a "reshape" operation is added internally to ensure that the input requirement is of fixed length. Thus, padding or truncation of the sequence is required, which results in the dynamic RNN model being indirectly converted into a static RNN model.

5.3.4 Comparison between Previous Work and SeqMobile

We briefly compare SeqMobile against two other previous works (i.e., MobiDroid [114] and MobiTive [117]). We use our results for Samsung S10+ as a baseline to benchmark against Nexus 6P from MobiDroid and Huawei P30 from MobiTive. As shown in

Table 5.8, even though the features used in the other two systems are similar (e.g., API calls and manifest properties), our sequence-based approach, which contains additional semantics, is able to achieve a much higher accuracy (i.e., 97.85%). Although there is an improvement in time cost when comparing to MobiDroid (4.16 seconds vs. 17.76 seconds), our time cost is still approximately 10 times higher (4.16 seconds vs. 0.46 seconds) when compare to MobiTive. Despite our high time cost, with the additional semantics from the sequence-based features, our accuracy is higher than MobiTive (96.75% vs. 97.85%). Also, the study from MobiTive [117] shows a trend of mobile hardware performance improving over the years, we strongly believe that our approach can achieve a significantly lower time cost in the upcoming years, making the accuracy of the detection system a much more important factor. In other word, for current devices' hardware performance, MobiTive have achieved an advantage with its simpler feature selection, however, in the near future, when the hardware specification of Android devices improved along with the Moore's law, SeqMobile will surely become more suitable to be accepted with its higher accuracy from more complicated feature selection.

5.4 Chapter Summary

In this work, we have investigated the effectiveness of using sequence-based learning approach together with performance optimization methods to detect malicious applications on device end. The evaluation results show that our approach achieves a high accuracy (i.e., 97.85%) and a reasonable detection time (i.e., < 5 seconds) on flagship phones. In addition, we have also provided a guidance on the state-of-the-art TensorFlow model optimization toolkit for device-end sequence-based approaches.

Chapter 6

Conclusions and Future Directions

In this chapter, we summarize the research work that we have conducted in the thesis and further discuss our future research directions and work.

6.1 Conclusions

Along with the increasing market sharing ratio of Android OS and diversity of deployed devices, the security threats of Android malware in different usage scenarios become more sophisticated and hard to be handled by strait forward solutions. In this thesis, we present a series of works to detection Android malware with domain knowledge in different usage scenarios to provide potential solutions for the community.

In the first work, we introduce a novel attack model to depict the essential characteristics and features. In addition, we build a transformation from an Android application to a directed graph, called the inter-component communication graph. ICCG captures all structure information of application, including call relationships and communications between different methods, and it contains all control flow information for each method. Then we propose an effective algorithm to search attacks in ICCG. The approach is proved to be feasible and effective in the experiments.

The second work presents MobiTive, a performance-sensitive Android malware detection system on mobile devices as a pre-installed solution. According to the effectiveness of selected features and the efficiency of feature extraction, MobiTive can provide a reliable detection accuracy and fast responsive (i.e., less than 3 seconds on average) detection service on mobile devices directly. To validate the efficiency and reliability, we evaluate

MobiTive on six real mobile devices. To provide more insights of this work, we also make an in-depth analysis of the performance trend on over one hundred mobile phones.

In the third work, we investigate the effectiveness of using sequence-based learning approach together with performance optimization methods to detect malicious applications on device end. The evaluation results show that our approach achieves a high accuracy (i.e., 97.85%) and a reasonable detection time (i.e., < 5 seconds) on flagship phones. In addition, we have also provided a guidance on the state-of-the-art TensorFlow model optimization toolkit for device-end sequence-based approaches.

6.2 Future Directions

In this thesis, we have an overall understanding of Android malware, and get familiar the techniques and their limitation to detect malware. In future, we are going to conduct the following works:

6.2.1 Adversarial Attack Against Learning Based Android Malware Detection

Although learning based systems are proved to be effective and efficient, the current research work shows that DL based system on specific areas (e.g., voice/image recognition) were easy to be confused by varies kinds of adversarial attack techniques. However, there are several differences in the deep learning based systems between malware detection task and voice/image recognition [159]. Unlike voice/image recognition, the adversarial attacks in malware detection cannot break the entire functionality in the applications easily in practice, so that the existing adversarial attacks against malware detection are always generated by manipulating the target malware application with un-triggered code snippets (e.g., dead code) instead of changing real functionalities [139]. Although it is able to generate adversarial samples to evade the classifier and achieve a high miss-classification rate, it is impractical so far, because such attack can be easily detected by leveraging other techniques such as static data flow analysis to delete such features that are introduced by adding dead code from attackers. Thus, in the future, we aim to investigate the potential vulnerabilities on the above problems in existing DL based Android malware detection by leveraging our domain knowledge and techniques.

6.2.2 Automatic Code Re-construction and Manipulation Technique

There is an important difference between voice/image recognition and malware detection tasks. Besides cheating the classification system, in malware detection tasks, the functionalities of the adversarial samples need to be maintained. Thus, with this restriction, re-constructing the code from adversarial features, and further manipulating them back into the original malware will be a necessary and challengeable road to build valuable adversarial samples against any learning based detection system. In the future, we aim to investigate the challenges and solutions in generating the adversarial samples for Android malware detection systems with various kinds of characterization methods.

6.2.3 Empirical Study of Interpretation Methods in Learning Based Android Malware Analysis

Unlike tradition program analysis based malware detection solutions, there are still gaps between the domain knowledge and learning based Android malware detection systems. Considering some state-of-the-art work used different kinds of interpretation methods to explain the decision made by DL based systems, we are interested in whether they are applicable for Android malware detection systems. Thus, we trend to present an empirical study on the applicability and effectiveness of existing interpretation methods on various learning based Android malware detection approaches to provide an insight for relevant area.

6.2.4 Robustness Evaluation for Android Malware Detection System

Unlike the DL based system on specific areas (e.g., voice/image recognition), DL based Android malware detection systems are still questioned on its robustness since lacking of enough research work which can provide evidence on that. In the future, we are interested to involve the interpretation methods together with the adversarial generation techniques to investigate the robustness of DL based Android malware detection system and further provide quantitative indicators for them.

6.2.5 Effectiveness Study of Used Feature Representation Towards Usage Scenarios in Android Security Analysis

Besides, we are also curious about investigating whether the existing feature representations (e.g., sequence, graph) of Android malware are really suitable for those many types of tasks. If the quality of representations can be enhanced specifically for different tasks, a prototype guide will become necessary for any industrial usage of Android Malware detection. In the future, we would like to present a study which will involve relevant topics to discover the implicit connection between different usage scenarios and their used representations to further provide an insight for relevant area.

References

[1] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, G. Pu, and Z. Su, "Large-scale analysis of framework-specific exceptions in android apps," in *2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE)*. IEEE, 2018, pp. 408–419.

[2] T. Su, L. Fan, S. Chen, Y. Liu, L. Xu, G. Pu, and Z. Su, "Why my app crashes understanding and benchmarking framework-specific exceptions of android apps," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[3] L. Fan, T. Su, S. Chen, G. Meng, Y. Liu, L. Xu, and G. Pu, "Efficiently manifesting asynchronous programming errors in android apps," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. ACM, 2018, pp. 486–497.

[4] X. Zhan, L. Fan, S. Chen, F. Wu, T. Liu, X. Luo, and Y. Liu, "Atvhunter: Reliable version detection of third-party libraries for vulnerability identification in android applications," in *Proceedings of the 43rd International Conference on Software Engineering*. IEEE Press, 2021.

[5] S. Chen, L. Fan, G. Meng, T. Su, M. Xue, Y. Xue, Y. Liu, and L. Xu, "An empirical assessment of security risks of global android banking apps," in *Proceedings of the 42nd International Conference on Software Engineering*. IEEE Press, 2020, pp. 596–607.

[6] S. Chen, T. Su, L. Fan, G. Meng, M. Xue, Y. Liu, and L. Xu, "Are mobile banking apps secure? what can be improved?" in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 797–802.

[7] Y. Xiao, B. Chen, C. Yu, Z. Xu, Z. Yuan, F. Li, B. Liu, Y. Liu, W. Huo, W. Zou, and W. Shi, "MVP: Detecting vulnerabilities using patch-enhanced vulnerability signatures," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 1165–1182. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/xiao

[8] Z. Xu, Y. Zhang, L. Zheng, L. Xia, C. Bao, Z. Wang, and Y. Liu, "Automatic hot patch generation for android kernels," in *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Aug. 2020, pp. 2397–2414. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/xu

[9] M. Arapinis, L. Mancini, E. Ritter, M. Ryan, N. Golde, K. Redon, and R. Borgaonkar, "New privacy issues in mobile telephony: fix and verification," in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*. Raleigh, North Carolina, USA: ACM Press, 2012, p. 205. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2382196.2382221

[10] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," in *2012 IEEE Symposium on Security and Privacy*. San Francisco, CA, USA: IEEE, May 2012, pp. 95–109. [Online]. Available: http://ieeexplore.ieee.org/document/6234407/

[11] X. Jiang and Y. Zhou, *Android malware*, ser. SpringerBriefs in computer science. New York: Springer, 2013, oCLC: ocn830367429.

[12] Y. Zhou and X. Jiang, "An Analysis of the AnserverBot Trojan," Technical report. [Online]. Available: https://www.csc2.ncsu.edu/faculty/xjiang4/pubs/AnserverBot_Analysis.pdf

[13] "The State of Mobile Phishing," Lookout, Tech. Rep. [Online]. Available: https://www.lookout.com/documents/threat-reports/us/lookout-phishing-spotlight-research-report-us.pdf

[14] "Simplocker: First Confirmed File-Encrypting Ransomware for Android," Tech. Rep. [Online]. Available: https://community.broadcom.com/symantecenterprise/communities/community-home/librarydocuments/viewdocument?DocumentKey=cb511b9d-5ae8-4a14-b1c7-aac55cfecea0&CommunityKey=1ecf5f55-9545-44d6-b0f4-4e4a7f5f5e68&tab=librarydocuments

[15] C. Tang, S. Chen, L. Fan, L. Xu, Y. Liu, Z. Tang, and L. Dou, "A Large-Scale Empirical Study on Industrial Fake Apps," in *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*. Montreal, QC, Canada: IEEE, May 2019, pp. 183–192. [Online]. Available: https://ieeexplore.ieee.org/document/8804455/

[16] S. Chen, L. Fan, C. Chen, M. Xue, Y. Liu, and L. Xu, "Gui-squatting attack: Automated generation of android phishing apps," *IEEE Transactions on Dependable and Secure Computing*, 2019.

[17] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: A Stealthy and Context-Aware Sound Trojan for Smartphones," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2011, San Diego, California, USA, 6th February - 9th February 2011*. The Internet Society, 2011. [Online]. Available: https://www.ndss-symposium.org/ndss2011/soundcomber-stealthy-and-context-aware-sound-trojan-smartphones

[18] 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 *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/hey-you-get-my-market-detecting-malicious-apps-official-and-alternative-android-markets

[19] W. Zhou, Y. Zhou, M. Grace, X. Jiang, and S. Zou, "Fast, scalable detection of "Piggybacked" mobile applications," in *Proceedings of the third ACM conference on Data and application security and privacy - CODASPY '13*. San Antonio, Texas, USA: ACM Press, 2013, p. 185. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2435349.2435377

[20] L. K. Yan and H. Yin, "DroidScope: Seamlessly Reconstructing the OS and Dalvik Semantic Views for Dynamic Android Malware Analysis," in *Presented as part of the 21st USENIX Security Symposium (USENIX Security 12)*. Bellevue, WA: USENIX, 2012, pp. 569–584. [Online]. Available: https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan

[21] C. Wu, Y. Zhou, K. Patel, Z. Liang, and X. Jiang, "AirBag: Boosting Smartphone Resistance to Malware Infection," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/airbag-boosting-smartphone-resistance-malware-infection

[22] K. Tam, S. J. Khan, A. Fattori, and L. Cavallaro, "CopperDroid: Automatic Reconstruction of Android Malware Behaviors," in *22nd Annual Network and Distributed System Security Symposium, NDSS 2015, San Diego, California, USA, February 8-11, 2015*. The Internet Society, 2015. [Online]. Available: https://www.ndss-symposium.org/ndss2015/copperdroid-automatic-reconstruction-android-malware-behaviors

[23] S. Rasthofer, S. Arzt, M. Miltenberger, and E. Bodden, "Harvesting Runtime Values in Android Applications That Feature Anti-Analysis Techniques," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/harvesting-runtime-values-android-applications-feature-anti-analysis-techniques.pdf

[24] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "FlowDroid: precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for Android apps," *ACM SIGPLAN Notices*, vol. 49, no. 6, pp. 259–269, Jun. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2666356.2594299

[25] L. Li, A. Bartel, T. F. Bissyande, J. Klein, Y. Le Traon, S. Arzt, S. Rasthofer, E. Bodden, D. Octeau, and P. McDaniel, "IccTA: Detecting Inter-Component Privacy Leaks in Android Apps," in *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*. Florence, Italy: IEEE, May 2015, pp. 280–291. [Online]. Available: http://ieeexplore.ieee.org/document/7194581/

[26] M. Y. Wong and D. Lie, "IntelliDroid: A Targeted Input Generator for the Dynamic Analysis of Android Malware," in *23rd Annual Network and Distributed System Security Symposium, NDSS 2016, San Diego, California, USA, February 21-24, 2016*. The Internet Society, 2016. [Online]. Available: http://wp.internetsociety.org/ndss/wp-content/uploads/sites/25/2017/09/intellidroid-targeted-input-generator-dynamic-analysis-android-malware.pdf

[27] D. Arp, M. Spreitzenbarth, M. Hubner, H. Gascon, and K. Rieck, "DREBIN: Effective and Explainable Detection of Android Malware in Your Pocket," in *21st Annual Network and Distributed System Security Symposium, NDSS 2014, San Diego, California, USA, February 23-26, 2014*. The Internet Society, 2014. [Online]. Available: https://www.ndss-symposium.org/ndss2014/drebin-effective-and-explainable-detection-android-malware-your-pocket

[28] 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*, M. Kutyłowski and J. Vaidya, Eds. Cham: Springer International Publishing, 2014, vol. 8712, pp. 163–182, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/978-3-319-11203-9_10

[29] S. Chen, M. Xue, Z. Tang, L. Xu, and H. Zhu, "StormDroid: A Streaminglized Machine Learning-Based System for Detecting Android Malware," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security - ASIA CCS '16*. Xi'an, China: ACM Press, 2016, pp. 377–388. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2897845.2897860

[30] E. Mariconti, L. Onwuzurike, P. Andriotis, E. De Cristofaro, G. Ross, and G. Stringhini, "MaMaDroid: Detecting Android Malware by Building Markov Chains of Behavioral Models," *arXiv:1612.04433 [cs]*, Nov. 2017, arXiv: 1612.04433. [Online]. Available: http://arxiv.org/abs/1612.04433

[31] S. Chen, M. Xue, L. Fan, S. Hao, L. Xu, H. Zhu, and B. Li, "Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach," *Computers & Security*, vol. 73, pp. 326–344, Mar. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167404817302444

[32] W. Yu, L. Ge, G. Xu, and X. Fu, "Towards Neural Network Based Malware Detection on Android Mobile Devices," in *Cybersecurity Systems for Human Cognition Augmentation*, R. E. Pino, A. Kott, and M. Shevenell, Eds. Cham: Springer International Publishing, 2014, vol. 61, pp. 99–117, series Title: Advances in Information Security. [Online]. Available: http://link.springer.com/10.1007/978-3-319-10374-7_7

[33] D. Octeau, P. McDaniel, S. Jha, A. Bartel, E. Bodden, J. Klein, and Y. L. Traon, "Effective Inter-Component Communication Mapping in Android: An Essential Step Towards Holistic Security Analysis," in *22nd USENIX Security Symposium (USENIX Security 13)*. Washington, D.C.: USENIX Association, 2013, pp. 543–558. [Online]. Available: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/octeau

[34] A. Bosu, F. Liu, D. D. Yao, and G. Wang, "Collusive Data Leak and More: Large-scale Threat Analysis of Inter-app Communications," in *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security*. Abu Dhabi United Arab Emirates: ACM, Apr. 2017, pp. 71–85. [Online]. Available: https://dl.acm.org/doi/10.1145/3052973.3053004

[35] X. Zhan, L. Fan, T. Liu, S. Chen, L. Li, H. Wang, Y. Xu, X. Luo, and Y. Liu, "Automated third-party library detection for android applications: Are we there yet?" in *Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering*. ACM, 2020.

[36] S. Chen, L. Fan, C. Chen, T. Su, W. Li, Y. Liu, and L. Xu, "Storydroid: Automated generation of storyboard for android apps," in *Proceedings of the 41st International Conference on Software Engineering*. IEEE Press, 2019, pp. 596–607.

[37] "ProGuard," Tech. Rep. [Online]. Available: https://developer.android.com/studio/build/shrink-code

[38] "Symantec Security." [Online]. Available: https://www.symantec.com/security-center/threats/

[39] "XDA." [Online]. Available: https://forum.xda-developers.com/

[40] Z. Yuan, Y. Lu, and Y. Xue, "Droiddetector: android malware characterization and detection using deep learning," *Tsinghua Science and Technology*, vol. 21, no. 1, pp. 114–123, Feb. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7399288/

[41] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An empirical study towards characterizing deep learning development and deployment across different frameworks and platforms," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Press, 2019.

[42] J. M. Zhang, M. Harman, L. Ma, and Y. Liu, "Machine learning testing: Survey, landscapes and horizons," *IEEE Transactions on Software Engineering*, pp. 1–1, 2020.

[43] K. Pei, Y. Cao, J. Yang, and S. Jana, "Deepxplore: Automated whitebox testing of deep learning systems," in *SOSP*, 2017.

[44] L. Ma, F. Juefei-Xu, F. Zhang, J. Sun, M. Xue, B. Li, C. Chen, T. Su, L. Li, Y. Liu *et al.*, "Deepgauge: Multi-granularity testing criteria for deep learning systems," in *ASE*, 2018.

[45] X. Xie, L. Ma, F. Juefei-Xu, M. Xue, H. Chen, Y. Liu, J. Zhao, B. Li, J. Yin, and S. See, "Deephunter: a coverage-guided fuzz testing framework for deep neural networks," in *ISSTA*, 2019.

[46] X. Du, X. Xie, Y. Li, L. Ma, Y. Liu, and J. Zhao, "Deepstellar: Model-based quantitative analysis of stateful deep learning systems," in *ESEC/FSE*, 2019.

[47] L. Ma, F. Zhang, J. Sun, M. Xue, B. Li, F. Juefei-Xu, C. Xie, L. Li, Y. Liu, J. Zhao *et al.*, "Deepmutation: Mutation testing of deep learning systems," in *ISSRE*, 2018.

[48] X. Zhang, X. Xie, L. Ma, X. Du, Q. Hu, Y. Liu, J. Zhao, and M. Sun, "Towards characterizing adversarial defects of deep learning software from the lens of uncertainty," in *ICSE*, 2020.

[49] B. David, X. Xie, L. Ma, L. Zhou, Y. Liu, C. Xu, and J. Zhao, "Cats are not fish: Deep learning testing calls for out-of-distribution awareness," in *ASE*, 2020.

[50] X. Xie, L. Ma, H. Wang, Y. Li, Y. Liu, and X. Li, "Diffchaser: Detecting disagreements for deep neural networks," in *IJCAI*, 2019.

[51] X. Du, Y. Li, X. Xie, L. Ma, Y. Liu, and J. Zhao, "Marble: Model-based robustness analysis of stateful deep learning systems," in *ASE*, 2020.

[52] S. Arzt and E. Bodden, "StubDroid: automatic inference of precise data-flow summaries for the android framework," in *Proceedings of the 38th International Conference on Software Engineering - ICSE '16*. Austin, Texas: ACM Press, 2016, pp. 725–735. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2884781.2884816

[53] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-Based Malware Detection on Android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, Jun. 2016. [Online]. Available: http://ieeexplore.ieee.org/document/7398053/

[54] F. Wei, S. Roy, X. Ou, and Robby, "Amandroid: A Precise and General Inter-component Data Flow Analysis Framework for Security Vetting of Android Apps," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. Scottsdale, Arizona, USA: ACM Press, 2014, pp. 1329–1341. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660267.2660357

[55] "Apkmirror market." [Online]. Available: https://www.apkmirror.com/

[56] "Tensorflow Lite." [Online]. Available: https://www.tensorflow.org/lite/

[57] "Caffe2 mobile." [Online]. Available: https://caffe2.ai/docs/mobile-integration.html

[58] "Core ML." [Online]. Available: https://developer.apple.com/documentation/coreml/

[59] "Pytorch mobile." [Online]. Available: https://pytorch.org/mobile/android/

[60] "Java native interface (JNI) framework." [Online]. Available: https://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html

[61] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in Android," in *Proceedings of the 9th international conference on Mobile systems, applications, and services - MobiSys '11*. Bethesda, Maryland, USA: ACM Press, 2011, p. 239. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1999995.2000018

[62] C.-Y. Huang, Y.-T. Tsai, and C.-H. Hsu, "Performance Evaluation on Permission-Based Detection for Android Malware," in *Advances in Intelligent Systems and Applications - Volume 2*, J.-S. Pan, C.-N. Yang, and C.-C. Lin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, vol. 21, pp. 111–120, series Title: Smart Innovation, Systems and Technologies. [Online]. Available: http://link.springer.com/10.1007/978-3-642-35473-1_12

[63] Z. Aung and W. Zaw, "Permission-Based Android Malware Detection," *International Journal of Scientific and Technology Research*, vol. 2, pp. 228–234, 2013.

[64] Y. Zhongyang, Z. Xin, B. Mao, and L. Xie, "DroidAlarm: an all-sided static analysis tool for Android privilege-escalation malware," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security - ASIA CCS '13*. Hangzhou, China: ACM Press, 2013, p. 353. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2484313.2484359

[65] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," in *2012 Seventh Asia Joint Conference on Information Security*. Tokyo, Japan: IEEE, Aug. 2012, pp. 62–69. [Online]. Available: http://ieeexplore.ieee.org/document/6298136/

[66] L. Deshotels, V. Notani, and A. Lakhotia, "DroidLegacy: Automated Familial Classification of Android Malware," in *Proceedings of ACM SIGPLAN on Program Protection and Reverse Engineering Workshop 2014 - PPREW'14*. San Diego, CA, USA: ACM Press, 2014, pp. 1–12. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2556464.2556467

[67] 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 SIGSAC Conference on Computer and Communications Security - CCS '14*. Scottsdale, Arizona, USA: ACM Press, 2014, pp. 1105–1116. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660267.2660359

[68] A. Narayanan, L. Yang, L. Chen, and L. Jinliang, "Adaptive and scalable Android malware detection through online learning," in *2016 International Joint Conference on Neural Networks (IJCNN)*. Vancouver, BC, Canada: IEEE, Jul. 2016, pp. 2484–2491. [Online]. Available: http://ieeexplore.ieee.org/document/7727508/

[69] A. Narayanan, G. Meng, L. Yang, J. Liu, and L. Chen, "Contextual Weisfeiler-Lehman graph kernel for malware detection," in *2016 International Joint Conference on Neural Networks (IJCNN)*. Vancouver, BC, Canada: IEEE, Jul. 2016, pp. 4701–4708. [Online]. Available: http://ieeexplore.ieee.org/document/7727817/

[70] A. Narayanan, C. Soh, L. Chen, Y. Liu, and L. Wang, "Apk2vec: Semi-Supervised Multi-view Representation Learning for Profiling Android Applications," in *2018 IEEE International Conference on Data Mining (ICDM)*. Singapore: IEEE, Nov. 2018, pp. 357–366. [Online]. Available: https://ieeexplore.ieee.org/document/8594860/

[71] A. Narayanan, M. Chandramohan, L. Chen, and Y. Liu, "A multi-view context-aware approach to Android malware detection and malicious code localization," *Empirical Software Engineering*, vol. 23, no. 3, pp. 1222–1274, Jun. 2018. [Online]. Available: http://link.springer.com/10.1007/s10664-017-9539-8

[72] G. Meng, Y. Xue, Z. Xu, Y. Liu, J. Zhang, and A. Narayanan, "Semantic modelling of Android malware for effective malware comprehension, detection, and classification," in *Proceedings of the 25th International Symposium on Software Testing and Analysis - ISSTA 2016*. Saarbrücken, Germany: ACM Press, 2016, pp. 306–317. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2931037.2931043

[73] G. Meng, Y. Xue, M. Chandramohan, A. Narayanan, Y. Liu, J. Zhang, and T. Chen, "Mystique: Evolving android malware for auditing anti-malware tools," in *Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security (AsiaCCS)*, 2016, pp. 365–376.

[74] Y. Xue, G. Meng, Y. Liu, T. H. Tan, H. Chen, J. Sun, and J. Zhang, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Transactions on Information Forensics and Security (TIFS)*, vol. 12, no. 7, pp. 1529–1544, July 2017.

[75] G. Meng, M. Patrick, Y. Xue, Y. Liu, and J. Zhang, "Securing android app markets via modelling and predicting malware spread between markets," *IEEE Transactions on Information Forensics and Security*, vol. xx, no. xx, p. xx, Dec 2018.

[76] 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*, vol. 38, no. 1, pp. 161–190, Feb. 2012. [Online]. Available: http://link.springer.com/10.1007/s10844-010-0148-x

[77] A.-D. Schmidt, F. Peters, F. Lamour, C. Scheel, S. A. Camtepe, and S. Albayrak, "Monitoring Smartphones for Anomaly Detection," *Mobile Networks and Applications*, vol. 14, no. 1, pp. 92–106, Feb. 2009. [Online]. Available: http://link.springer.com/10.1007/s11036-008-0113-x

[78] 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 '12*. Low Wood Bay, Lake District, UK: ACM Press, 2012, p. 281. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2307636.2307663

[79] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones," *ACM Transactions on Computer Systems*, vol. 32, no. 2, pp. 1–29, Jun. 2014. [Online]. Available: https://dl.acm.org/doi/10.1145/2619091

[80] K. Basu, P. Krishnamurthy, F. Khorrami, and R. Karri, "A Theoretical Study of Hardware Performance Counters-Based Malware Detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 512–525, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8744309/

[81] L. Fan, M. Xue, S. Chen, L. Xu, and H. Zhu, "Poster: Accuracy vs. time cost: Detecting android malware through pareto ensemble pruning," in *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. ACM, 2016, pp. 1748–1750.

[82] S. Chen, M. Xue, and L. Xu, "Towards adversarial detection of mobile malware: poster," in *Proceedings of the 22nd Annual International Conference on Mobile Computing and Networking, MobiCom 2016, New York City, NY, USA, October 3-7, 2016*, Y. Chen, M. Gruteser, Y. C. Hu, and K. Sundaresan, Eds. ACM, 2016, pp. 415–416. [Online]. Available: https://doi.org/10.1145/2973750.2985246

[83] N. McLaughlin, J. Martinez del Rincon, B. Kang, S. Yerima, P. Miller, S. Sezer, Y. Safaei, E. Trickel, Z. Zhao, A. Doupé, and G. Joon Ahn, "Deep Android Malware Detection," in *Proceedings of the Seventh ACM on Conference on Data and Application Security and Privacy*. Scottsdale Arizona USA: ACM, Mar. 2017, pp. 301–308. [Online]. Available: https://dl.acm.org/doi/10.1145/3029806.3029823

[84] K. Xu, Y. Li, R. H. Deng, and K. Chen, "DeepRefiner: Multi-layer Android Malware Detection System Applying Deep Neural Networks," in *2018 IEEE European Symposium on Security and Privacy (EuroS&P)*. London: IEEE, Apr. 2018, pp. 473–487. [Online]. Available: https://ieeexplore.ieee.org/document/8406618/

[85] T. Kim, B. Kang, M. Rho, S. Sezer, and E. G. Im, "A Multimodal Deep Learning Method for Android Malware Detection Using Various Features," *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 3, pp. 773–788, Mar. 2019. [Online]. Available: https://ieeexplore.ieee.org/document/8443370/

[86] S. Chen, M. Xue, L. Fan, L. Ma, Y. Liu, and L. Xu, "How can we craft large-scale android malware? an automated poisoning attack," in *2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile)*. IEEE, 2019, pp. 21–24.

[87] B. Wu, S. Chen, C. Gao, L. Fan, Y. Liu, W. Wen, and M. R. Lyu, "Why an android app is classified as malware? towardsmalware classification interpretation," in *ACM Transactions on Software Engineering and Methodology*. ACM, 2020.

[88] M. C. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic Detection of Capability Leaks in Stock Android Smartphones," in *19th Annual Network and Distributed System Security Symposium, NDSS 2012, San Diego, California, USA, February 5-8, 2012*. The Internet Society, 2012. [Online]. Available: https://www.ndss-symposium.org/ndss2012/systematic-detection-capability-leaks-stock-android-smartphones

[89] L. Xing, X. Pan, R. Wang, K. Yuan, and X. Wang, "Upgrading Your Android, Elevating My Malware: Privilege Escalation through Mobile OS Updating," in *2014 IEEE Symposium on Security and Privacy*. San Jose, CA: IEEE, May 2014, pp. 393–408. [Online]. Available: http://ieeexplore.ieee.org/document/6956577/

[90] H. Gunadi and A. Tiu, "Efficient Runtime Monitoring with Metric Temporal Logic: A Case Study in the Android Operating System," in *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, ser. Lecture Notes in Computer Science, C. B. Jones, P. Pihlajasaari, and J. Sun, Eds., vol. 8442. Springer, 2014, pp. 296–311. [Online]. Available: https://doi.org/10.1007/978-3-319-06410-9_21

[91] P. Vekris, R. Jhala, S. Lerner, and Y. Agarwal, "Towards Verifying Android Apps for the Absence of No-Sleep Energy Bugs," in *2012 Workshop on Power-Aware Computing and Systems (HotPower 12)*. Hollywood, CA: USENIX Association, 2012. [Online]. Available: https://www.usenix.org/conference/hotpower12/workshop-program/presentation/Vekris

[92] A. Pathak, Y. C. Hu, and M. Zhang, "Where is the energy spent inside my app?: fine grained energy accounting on smartphones with Eprof," in *Proceedings of the 7th ACM european conference on Computer Systems - EuroSys '12*. Bern, Switzerland: ACM Press, 2012, p. 29. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2168836.2168841

[93] G. Meng, R. Feng, G. Bai, K. Chen, and Y. Liu, "DroidEcho: an in-depth dissection of malicious behaviors in Android applications," *Cybersecurity*, vol. 1, no. 1, p. 4, Dec. 2018. [Online]. Available: https://cybersecurity.springeropen.com/articles/10.1186/s42400-018-0006-7

[94] Z. Yang, M. Yang, Y. Zhang, G. Gu, P. Ning, and X. S. Wang, "AppIntent: analyzing sensitive data transmission in android for privacy leakage detection," in *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security - CCS '13*. Berlin, Germany: ACM Press, 2013, pp. 1043–1054. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2508859.2516676

[95] K. Z. Chen, N. Johnson, V. D'Silva, S. Dai, K. MacNamara, T. Magrino, E. Wu, M. Rinard, and D. Song, "Contextual Policy Enforcement in Android Applications with Permission Event Graphs," in *Proc. of the Conference on Networked and Distributed System Security*, p. 19.

[96] W. Enck, M. Ongtang, and P. McDaniel, "Understanding Android Security," *IEEE Security & Privacy Magazine*, vol. 7, no. 1, pp. 50–57, Jan. 2009. [Online]. Available: http://ieeexplore.ieee.org/document/4768655/

[97] A. Shabtai, Y. Fledel, U. Kanonov, Y. Elovici, S. Dolev, and C. Glezer, "Google Android: A Comprehensive Security Assessment," *IEEE Security & Privacy Magazine*, vol. 8, no. 2, pp. 35–44, Mar. 2010. [Online]. Available: http://ieeexplore.ieee.org/document/5396322/

[98] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri, "A Study of Android Application Security," in *Proceedings of the 20th USENIX Conference on Security*, ser. SEC'11. USA: USENIX Association, 2011, p. 21, event-place: San Francisco, CA.

[99] M. Zhang and H. Yin, "Efficient, context-aware privacy leakage confinement for android applications without firmware modding," in *Proceedings of the 9th ACM symposium on Information, computer and communications security - ASIA CCS '14*. Kyoto, Japan: ACM Press, 2014, pp. 259–270. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2590296.2590312

[100] W. Luo, S. Xu, and X. Jiang, "Real-time detection and prevention of android SMS permission abuses," in *Proceedings of the first international workshop on Security in embedded systems and smartphones - SESP '13*. Hangzhou, China: ACM Press, 2013, p. 11. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2484417.2484422

[101] A. Pathak, Y. C. Hu, and M. Zhang, "Bootstrapping energy debugging on smartphones: a first look at energy bugs in mobile devices," in *Proceedings of the 10th ACM Workshop on Hot Topics in Networks - HotNets '11*. Cambridge, Massachusetts: ACM Press, 2011, pp. 1–6. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2070562.2070567

[102] S. Hao, D. Li, W. G. J. Halfond, and R. Govindan, "Estimating mobile application energy consumption using program analysis," in *2013 35th International Conference on Software Engineering (ICSE)*. San Francisco, CA, USA: IEEE, May 2013, pp. 92–101. [Online]. Available: http://ieeexplore.ieee.org/document/6606555/

[103] B. Prince, "New Android Malware Targets Banking Apps, Phone Information: FireEye," Tech. Rep., Jul. 2014. [Online]. Available: https://www.securityweek.com/new-android-malware-targets-banking-apps-phone-information-fireeye

[104] "Mobile Threat Report 2013," F-Secure Lab, Technical report, Mar. 2013. [Online]. Available: https://www.infopoint-security.de/medien/mobile_threat_report_q3_2013.pdf

[105] C. Hilgers, H. Macht, T. Muller, and M. Spreitzenbarth, "Post-Mortem Memory Analysis of Cold-Booted Android Devices," in *2014 Eighth International Conference on IT Security Incident Management & IT Forensics*. Münster, Germany: IEEE, May 2014, pp. 62–75. [Online]. Available: http://ieeexplore.ieee.org/document/6824082/

[106] Q. A. Chen, Z. Qian, and Z. M. Mao, "Peeking into Your App without Actually Seeing It: UI State Inference and Novel Android Attacks," in *Proceedings of the 23rd USENIX Conference on Security Symposium*, p. 17.

[107] C. Orthacker, P. Teufl, S. Kraxberger, G. Lackner, M. Gissing, A. Marsalek, J. Leibetseder, and O. Prevenhueber, "Android Security Permissions – Can We Trust Them?" in *Security and Privacy in Mobile Information and Communication Systems*, R. Prasad, K. Farkas, A. U. Schmidt, A. Lioy, G. Russello, and F. L. Luccio, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2012, vol. 94, pp. 40–51, series Title: Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering. [Online]. Available: http://link.springer.com/10.1007/978-3-642-30244-2_4

[108] Y. Feng, S. Anand, I. Dillig, and A. Aiken, "Apposcopy: semantics-based detection of Android malware through static analysis," in *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*. Hong Kong, China: ACM Press, 2014, pp. 576–587. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2635868.2635869

[109] Z. Qu, V. Rastogi, X. Zhang, Y. Chen, T. Zhu, and Z. Chen, "AutoCog: Measuring the Description-to-permission Fidelity in Android Applications," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security - CCS '14*. Scottsdale, Arizona, USA: ACM Press, 2014, pp. 1354–1365. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2660267.2660287

[110] K. W. Y. Au, Y. F. Zhou, Z. Huang, and D. Lie, "PScout: analyzing the Android permission specification," in *Proceedings of the 2012 ACM conference on Computer and communications security - CCS '12*. Raleigh, North Carolina, USA: ACM Press, 2012, p. 217. [Online]. Available: http://dl.acm.org/citation.cfm?doid=2382196.2382222

[111] R. Vallée-Rai, P. Co, E. Gagnon, L. Hendren, P. Lam, and V. Sundaresan, "Soot: a Java bytecode optimization framework," in *CASCON First Decade High Impact Papers on - CASCON '10*. Toronto, Ontario, Canada: ACM Press, 2010, pp. 214–224. [Online]. Available: http://portal.acm.org/citation.cfm?doid=1925805.1925818

[112] O. Lhoták and L. Hendren, "Scaling Java Points-to Analysis Using Spark," in *Compiler Construction*, G. Goos, J. Hartmanis, J. van Leeuwen, and G. Hedin, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, vol. 2622, pp. 153–169, series Title: Lecture Notes in Computer Science. [Online]. Available: http://link.springer.com/10.1007/3-540-36579-6_12

[113] R. Feng, Y. Liu, and S. Lin, "A Performance-Sensitive Malware Detection System on Mobile Platform," in *Formal Methods and Software Engineering*, Y. Ait-Ameur and S. Qin, Eds. Cham: Springer International Publishing, 2019, pp. 493–497.

[114] R. Feng, S. Chen, X. Xie, L. Ma, G. Meng, Y. Liu, and S.-W. Lin, "MobiDroid: A Performance-Sensitive Malware Detection System on Mobile Platform," in *2019 24th International Conference on Engineering of Complex Computer Systems (ICECCS)*. Guangzhou, China: IEEE, Nov. 2019, pp. 61–70. [Online]. Available: https://ieeexplore.ieee.org/document/8882775/

[115] R. Feng, S. Chen, X. Xie, G. Meng, S. W. Lin, and Y. Liu, "A performance-sensitive malware detection system using deep learning on mobile devices," *IEEE Transactions on Information Forensics and Security*, vol. 16, pp. 1563–1578, 2021.

[116] "Quantized model not supporting dynamic input length." [Online]. Available: https://github.com/tensorflow/tensorflow/issues/24607#issuecomment-582675214

[117] "MobiTive." [Online]. Available: https://sites.google.com/view/mobitive2020

[118] "Axmldec: Binary XML decoder." [Online]. Available: https://github.com/ytsutano/axmldec

[119] "Axml: A parser for binary xml files." [Online]. Available: https://github.com/xgouchet/AXML

[120] "Dex2jar." [Online]. Available: https://github.com/pxb1988/dex2jar

[121] "Apktool. A tool for reverse engineering Android apk files." [Online]. Available: https://ibotpeaches.github.io/Apktool/

[122] "Androguard." [Online]. Available: https://github.com/androguard/androguard/

[123] "Soot." [Online]. Available: https://github.com/Sable/soot/

[124] "FlowDroid." [Online]. Available: https://github.com/secure-software-engineering/FlowDroid

[125] "Keras: Neural networks API." [Online]. Available: https://keras.io/

[126] "Numpy: Package for scientific computing." [Online]. Available: https://numpy.org/

[127] "Scikit-learn." [Online]. Available: https://scikit-learn.org/stable/

[128] "Tensorflow." [Online]. Available: https://www.tensorflow.org/

[129] "Dalvik opcode." [Online]. Available: http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

[130] "Virustotal." [Online]. Available: https://www.virustotal.com/

[131] "Contagio Website." [Online]. Available: http://contagiominidump.blogspot.com/

[132] "Pwnzen Infotech Inc." [Online]. Available: http://www.pwnzen.com/

[133] E. B. Karbab, M. Debbabi, A. Derhab, and D. Mouheb, "MalDozer: Automatic framework for android malware detection using deep learning," *Digital Investigation*, vol. 24, pp. S48–S59, Mar. 2018. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S1742287618300392

[134] Q. Guo, S. Chen, X. Xie, L. Ma, Q. Hu, H. Liu, Y. Liu, J. Zhao, and X. Li, "An Empirical Study Towards Characterizing Deep Learning Development and Deployment Across Different Frameworks and Platforms," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. San Diego, CA, USA: IEEE, Nov. 2019, pp. 810–822. [Online]. Available: https://ieeexplore.ieee.org/document/8952401/

[135] "Inspeckage: a tool developed to offer dynamic analysis of android applications." [Online]. Available: https://github.com/ac-pm/Inspeckage

[136] "Android Profiler." [Online]. Available: https://developer.android.com/studio/profile/android-profiler

[137] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a Defense to Adversarial Perturbations Against Deep Neural Networks," in *2016 IEEE Symposium on Security and Privacy (SP)*. San Jose, CA: IEEE, May 2016, pp. 582–597. [Online]. Available: http://ieeexplore.ieee.org/document/7546524/

[138] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The Limitations of Deep Learning in Adversarial Settings," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. Saarbrucken: IEEE, Mar. 2016, pp. 372–387. [Online]. Available: http://ieeexplore.ieee.org/document/7467366/

[139] X. Chen, C. Li, D. Wang, S. Wen, J. Zhang, S. Nepal, Y. Xiang, and K. Ren, "Android HIV: A Study of Repackaging Malware for Evading Machine-Learning Detection," *IEEE Transactions on Information Forensics and Security*, vol. 15, pp. 987–1001, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/8782574/

[140] L. Li, J. Gao, T. F. Bissyandé, L. Ma, X. Xia, and J. Klein, "Characterising deprecated Android APIs," in *Proceedings of the 15th International Conference on Mining Software Repositories - MSR '18*. Gothenburg, Sweden: ACM Press, 2018, pp. 254–264. [Online]. Available: http://dl.acm.org/citation.cfm?doid=3196398.3196419

[141] R. Feng, J. Q. Lim, S. Chen, S.-W. Lin, and Y. Liu, "Seqmobile: A sequence based efficient android malware detection system using rnn on mobile devices," in *2020 25th International Conference on Engineering of Complex Computer Systems (ICECCS)*. IEEE, 2020.

[142] R. Nix and J. Zhang, "Classification of Android apps and malware using deep neural networks," in *2017 International Joint Conference on Neural Networks (IJCNN)*. Anchorage, AK, USA: IEEE, May 2017, pp. 1871–1878. [Online]. Available: http://ieeexplore.ieee.org/document/7966078/

[143] X. Xiao, S. Zhang, F. Mercaldo, G. Hu, and A. K. Sangaiah, "Android malware detection based on system call sequences and LSTM," *Multimedia Tools and Applications*, vol. 78, no. 4, pp. 3979–3999, Feb. 2019. [Online]. Available: http://link.springer.com/10.1007/s11042-017-5104-0

[144] Q. Jerome, K. Allix, R. State, and T. Engel, "Using opcode-sequences to detect malicious Android applications," in *2014 IEEE International Conference on Communications (ICC)*. Sydney, NSW: IEEE, Jun. 2014, pp. 914–919. [Online]. Available: http://ieeexplore.ieee.org/document/6883436/

[145] S. Verma and S. Muttoo, "An Android Malware Detection Framework-based on Permissions and Intents," *Defence Science Journal*, vol. 66, p. 618, 2016.

[146] H. Kang, J.-w. Jang, A. Mohaisen, and H. K. Kim, "Detecting and Classifying Android Malware Using Static Analysis along with Creator Information," *International Journal of Distributed Sensor Networks*, vol. 11, no. 6, p. 479174, Jun. 2015. [Online]. Available: http://journals.sagepub.com/doi/10.1155/2015/479174

[147] A. Feizollah, N. B. Anuar, R. Salleh, G. Suarez-Tangil, and S. Furnell, "AndroDialysis: Analysis of Android Intent Effectiveness in Malware Detection," *Computers & Security*, vol. 65, pp. 121–134, Mar. 2017. [Online]. Available: https://linkinghub.elsevier.com/retrieve/pii/S0167404816301602

[148] "MouaBad.P : Pocket Dialing For Profit," Tech. Rep. [Online]. Available: https://blog.lookout.com/mouabadp-pocket-dialing-for-profit

[149] "Android manifest permissions." [Online]. Available: https://cs.android.com/android/platform/superproject/+/master:tools/adt/idea/android/testData/sdk1.5/docs/reference/android/Manifest.permission.html

[150] "Google Android Intents." [Online]. Available: https://cs.android.com/android/platform/superproject/+/master:frameworks/base/core/java/android/content/Intent.java

[151] "SeqMobile." [Online]. Available: https://sites.google.com/view/seqmobile/home

[152] S. Ioffe and C. Szegedy, "Batch Normalization: Accelerating Deep Network Training by Reducing Internal Covariate Shift," in *Proceedings of the 32nd International Conference on International Conference on Machine Learning - Volume 37*, ser. ICML'15. JMLR.org, 2015, pp. 448–456, event-place: Lille, France.

[153] T. Cooijmans, N. Ballas, C. Laurent, C. Gulcehre, and A. C. Courville, "Recurrent Batch Normalization," in *5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings*. OpenReview.net, 2017. [Online]. Available: https://openreview.net/forum?id=r1VdcHcxx

[154] "Tensorflow Lite Converter." [Online]. Available: https://www.tensorflow.org/api_docs/python/tf/lite/TFLiteConverter

[155] "Android sdk build tools 28." [Online]. Available: https://aur.archlinux.org/packages/android-sdk-build-tools-28/

[156] "A parser for binary xml files." [Online]. Available: https://code.google.com/archive/p/android4me/downloads

[157] "tensorflow-nightly build 2.2.0.dev20200430." [Online]. Available: https://libraries.io/pypi/tf-nightly/2.2.0.dev20200430

[158] "Virusshare." [Online]. Available: https://virusshare.com/

[159] G. Chen, S. Chen, L. Fan, X. Du, Z. Zhao, F. Song, and Y. Liu, "Who is real bob? adversarial attacks on speaker recognition systems," in *IEEE Symposium on Security and Privacy (Oakland)*, 2021.