Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence (IJCAI-18)

Make Evasion Harder: An Intelligent Android Malware Detection System *

Shifu Hou[1], Yanfang Ye [1] †, Yangqiu Song[2], Melih Abdulhayoglu[3]
[1] Department of CSEE, West Virginia Univerisity, WV, USA
[2] Department of CSE, Hong Kong University of Science and Technology, Hong Kong
[3]Comodo Security Solutions, Inc., Clifton, NJ, USA
shhou@mix.wvu.edu, yanfang.ye@mail.wvu.edu, yqsong@cse.ust.hk, melih@comodo.com

Abstract

To combat the evolving Android malware attacks, in this paper, instead of only using Application Programming Interface (API) calls, we further analyze the different relationships between them and create higher-level semantics which require more efforts for attackers to evade the detection. We represent the Android applications (apps), related APIs, and their rich relationships as a structured heterogeneous information network (HIN). Then we use a meta-path based approach to characterize the semantic relatedness of apps and APIs. We use each meta-path to formulate a similarity measure over Android apps, and aggregate different similarities using multi-kernel learning to make predictions. Promising experimental results based on real sample collections from Comodo Cloud Security Center demonstrate that our developed system *HinDroid* outperforms other alternative Android malware detection techniques.

1 Introduction

Due to the mobility and ever expanding capabilities, smart phones have been widely used in people's daily life. Android, as an open source and customizable operating system for smart phones, is currently dominating the smart phone market by over 81% [IDC, 2017]. Due to its large market share and open source ecosystem of development, Android attracts not only the developers for producing legitimate Android applications (apps), but also attackers to disseminate malware (***mal***icious soft***ware***) that deliberately fulfills the harmful intent to the smart phone users. Many examples of Android malware have already been released in the market (e.g., Geinimi, DriodKungfu and Lotoor) which posed serious threats to the smart phone users, such as stealing user credentials and locking user's smart phone until the ransom is paid [Felt *et al.*, 2011]. It's reported that one in every five Android apps were actually malware [Wood, 2015]. The increasing volume and sophistication of Android malware calls for new defensive techniques that are robust and capable of protecting users against novel threats [Burguera *et al.*, 2011; Wu and Hung, 2014; Dimjasevic *et al.*, 2016; Chen *et al.*, 2017a; Ye *et al.*, 2017b].

To combat the Android malware's evasion tactics, in this paper, instead of using API calls only, we further analyze the relationships among them, e.g., whether the extracted API calls belong to the same code block, are with the same package name, or use the same invoke method, etc. Relations between APIs and apps and different types relations among apps themselves can introduce higher-level semantics and require more efforts for attackers to evade the detection. To represent the rich semantics of relationships, we first introduces a structured heterogeneous information network (HIN) [Sun and Han, 2012] representation to depict apps and APIs. Then we use meta-path [Sun *et al.*, 2011] to incorporate higher-level semantics to build up the semantic relatedness of apps. Since there can be multiple meta-paths to define different similarities and we want to incorporate all useful meta-paths and discard useless ones, we propose to use a multi-kernel learning algorithm [Vishwanathan *et al.*, 2010] to automatically learn the weights of different similarities from data. We then develop an intelligent system *HinDroid* for automatic Android malware detection. Promising experimental results based on the real sample collection from Comodo Cloud Security Center demonstrate the effectiveness and efficiency of *HinDroid*, which has been incorporated into the scanning tool of Comodo Mobile Security product.

2 System Architecture

The system architecture of *HinDroid* is shown in Figure 1, which consists of the following five major components.

- ***Unzipper and Decompiler:*** It first unzips each Android Application Package (APK) to get the dex file, and then generates the smali codes by decompiling the dex file. *Dex* [Hou *et al.*, 2017] is a file format which contains compiled code written for Android and can be interpreted by the DalvikVM, but it is unreadable. We need to convert the dex file to a readable format. *Smali* is an assembler/disassembler for the dex format, which provides us readable

*This paper is an abridged version of a paper titled "HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network" that won the best paper award and the best student paper award in the applied data science track at the SIGKDD 2017 conference.

†Corresponding author.

code in smali language. *Smali code* [Hou *et al.*, 2017] is the intermediate but interpreted code between Java and DalvikVM. Listing 1 shows a segment of the converted smali code from a ransomware "Locker.apk" that will lock user's smart phone until the ransom is paid.

```
1  .method protected
2  loadLibs(Landroid/content/Context;)V
3  .locals 4
4  :try_start_0
5  new-instance v0, Ljava/io/BufferedReader;
6  new-instance v1, Ljava/io/InputStreamReader;
7  invoke-static {}, Ljava/lang/Runtime;->getRuntime()
   Ljava/lang/Runtime;
8  move-result-object v2
9  const-string v3, "getprop_ro.product.cpu.abi"
10 invoke-virtual {v2, v3}, Ljava/lang/Runtime;->exec
   (Ljava/lang/String;)Ljava/lang/Process;
11 move-result-object v2
12 invoke-virtual {v2}, Ljava/lang/Process;->getInputStream()
   Ljava/io/InputStream;
13 ......
14 .end method
```

Listing 1: An example of smali code.

- ***Feature Extractor:*** As API calls can be used to represent the behaviors of an Android app, it automatically extracts the API calls from the decompiled smali codes generated from the previous component. Based on the extracted API calls, the relationships among them will be further analyzed, i.e., if the extracted API calls belong to the same smali code block, are with the same package name, or use the same invoke method. (See Section 3.1 for details).
- ***HIN Constructor:*** This component constructs the HIN based on the features extracted by the previous components. It first builds connections between the apps and the extracted API calls, and defines the types of relationships between these API calls. Then the adjacency matrices among different entity types are constructed, and further the commuting matrices of different meta-paths are enumerated and built. (See Section 3.2 for details.)
- ***Multi-kernel Learner:*** Given the commuting matrices from HIN, we build kernels for the Support Vector Machines (SVMs). Using standard multi-kernel learning, the weights of different meta-paths can be optimized. Given the meta-path weights, the different commuting matrices can be combined to formulate a more powerful kernel for Android malware detection. (See Section 3.3 for details.)
- ***Malware Detector:*** For each newly collected unknown Android app, it will be first parsed through the unzipper and decompiler to get the smali codes, then its API calls will be extracted from the smali codes, and the relationships among these API calls will be further analyzed. Based on these extracted features and using the constructed classification model, this app will be labeled as either benign or malicious.

3 Proposed Method

3.1 Feature Extraction

As API calls are used by the Android apps to access operating system functionality and system resources [Tam *et al.*, 2015], they can be used as representations of the behaviors of Android apps [Wu *et al.*, 2012; Peiravian and Zhu, 2013; Yang *et al.*, 2014]. After we unzip and decompile the collected Android apps, the API calls will be extracted from the decompiled smali codes. For example, in the smali code segment shown in Listing 1, the API calls of "*Ljava/lang/Runtime; → getRuntime() Ljava/lang/Runtime*" , "*Ljava/lang/Runtime; → exec (Ljava/lang/String;) Ljava/lang/Process*" and "*Ljava/lang/Process; → getInputStream() Ljava/io/InputStream*"will be extracted. After the extraction of the API calls, to describe the relation (i.e., ***R0***) between app and API, we build the ***App-API*** matrix $\mathbf{A}$ whose element $\mathbf{A}_{ij} = a_{ij} \in \{0, 1\}$ denotes if app i contains API j.

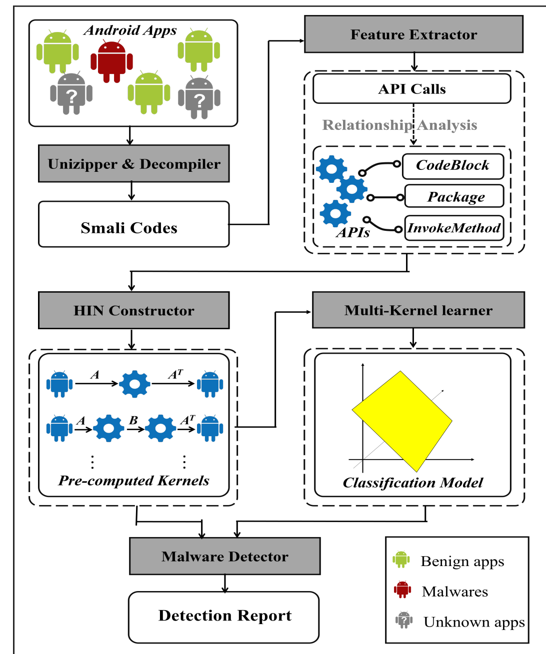

Figure 1: System architecture of *HinDroid*.

Although API calls can be used to represent the behaviors of an Android app, the relations among them can imply important information for malware detection [Ye *et al.*, 2011]. For example, as the aforementioned ransomeware "Locker.apk", the API calls of "*Ljava/io/FileOutputStream → write*", "*Ljava/io/IOException → printStackTrace*", and "*Ljava/lang /System → load*" together in the method of "*loadLibs*" in the converted smali code indicate this ransomware intends to write malicious code into system kernel. Though it may be common to use them individually in benign apps, they three together in the same method of the converted smali code rarely appear in benign files. Thus, the relationship that these three API calls co-exist in the same method in the converted smali code is an important information for such ransomware detection. To describe such relationships, we define a *code block* as the code between a pair of "*.method*" and "*.endmethod*" in the smali file, which reflects the structural information among the API calls. To represent such kind of relationship ***R1***, we generate the ***API-CodeBlock*** matrix $\mathbf{B}$ where each element $\mathbf{B}_{ij} = b_{ij} \in \{0, 1\}$

denotes whether this pair of API calls belong to the same code block. Except for that whether the API calls co-exist in the same code block, we find that API calls which belong to the same package always show similar intent and indicate strong connections among them. For example, the API calls in the package of "Lorg/apache/http/HttpRequest" are related to Internet connection. To represent such kind of relationship ***R2***, we generate the ***API-Package*** matrix $\mathbf{P}$ where each element $\mathbf{P}_{ij} = p_{ij} \in \{0, 1\}$ denotes if a pair of API calls belong to the same package. In the smali code, there are five different methods to invoke an API call, which are *invoke-static*, *invoke-virtual*, *invoke-direct*, *invoke-super*, and *invoke-interface*. The same invoke method can show the common properties of the API calls (like the words have the same part of speech), which indicate implicit associations among them. To represent this kind of relationship ***R3***, we generate the ***API-InvokeMethod*** matrix $\mathbf{I}$ where each element $\mathbf{I}_{ij} = i_{ij} \in \{0, 1\}$ indicates whether a pair of API calls use the same invoke method.

The relationships among the extracted API calls (i.e., whether they belong to the same code block, are with the same package name, or use the same invoke method) create a higher-level representation than a simple list of API calls and require more efforts for attackers to evade the detection (e.g., it may result in execution collapse if the attackers add several non-associated API calls in the same code block).

3.2 HIN Construction

Given the analysis of rich relationship types of API calls for Andriod apps, it is important to model them in a proper way so that different relations can be better and easier handled. When we apply machine learning algorithms, it is also better to distinguish different types of relations. Here, we introduce how to use heterogeneous information network to represent the Andriod apps by using the features extracted above.

Definition 1 *[Sun and Han, 2012] A **heterogeneous information network** (HIN) is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with an entity type mapping ϕ: $\mathcal{V} \to \mathcal{A}$ and a relation type mapping ψ: $\mathcal{E} \to \mathcal{R}$, where $\mathcal{V}$ denotes the entity set and $\mathcal{E}$ denotes the link set, $\mathcal{A}$ denotes the entity type set and $\mathcal{R}$ denotes the relation type set, and the number of entity types $|\mathcal{A}| > 1$ or the number of relation types $|\mathcal{R}| > 1$. The **network schema** for network G, denoted as $\mathcal{T}_G = (\mathcal{A}, \mathcal{R})$, is a graph with nodes as entity types from $\mathcal{A}$ and edges as relation types from $\mathcal{R}$.*

HIN not only provides the network structure of the data associations, but also provides a high-level abstraction of the categorical association. In our application for Android malware detection, we have two entity types (i.e., Android app and API call) and four relation types (i.e., ***R0-R3***). The different types of entities and different relations of APIs motivate us to use a machine-readable representation to enrich the semantics of similarities among APIs. Meta-path [Sun *et al.*, 2011] was used in the concept of HIN to formulate the semantics of higher-order relationships among entities. We follow this concept and extend it to our *HinDroid* framework.

Definition 2 *[Sun* et al.*, 2011] A **meta-path** $\mathcal{P}$ is a path defined on the graph of network schema $\mathcal{T}_G = (\mathcal{A}, \mathcal{R})$, and is denoted in the form of $A_1 \xrightarrow{R_1} A_2 \xrightarrow{R_2} \ldots \xrightarrow{R_L} A_{L+1}$, which defines a composite relation $R = R_1 \cdot R_2 \cdot \ldots \cdot R_L$ between types A_1 and A_{L+1}, where $\cdot$ denotes relation composition operator, and L is the length of $\mathcal{P}$.*

A typical meta-path $\mathcal{P}_1$ for apps is *App* $\xrightarrow{\text{contains}}$ *API* $\xrightarrow{\text{contains}^{-1}}$ *App*, which means that we want to connect two apps through the path containing the same API over the HIN; another example meta-path $\mathcal{P}_2$ is *App* $\xrightarrow{\text{contains}}$ *API* $\xrightarrow{\text{same code block}}$ *API* $\xrightarrow{\text{contains}^{-1}}$ *App*, which denotes that we compute the similarity between two apps not only considering API calls inside, but also considering the type of API calls inside. To measure the similarity over apps using a particular meta-path, we use commuting matrix [Sun *et al.*, 2011] to compute the counting-based similarity matrix for a meta-path. Take $\mathcal{P}_1$ as an example, the commuting matrix of apps computed using $\mathcal{P}_1$ is $\mathbf{A}\mathbf{A}^T$, while the one for $\mathcal{P}_2$ is $\mathbf{A}\mathbf{B}\mathbf{A}^T$.

3.3 Multi-Kernel Learning

Given a network schema with different types of entities and relations, we can enumerate a lot of meta-paths. However, not all of the meta-paths are useful for the particular Android malware detection problem. Here we propose to use a multi-kernel learning algorithm to automatically incorporate different similarities and determine the weight for each meta-path when classifying apps.

Suppose we have K meta-paths $\mathcal{P}_k, k = 1, \ldots, K$. We can compute the corresponding commuting matrices $\mathbf{M}_{\mathcal{P}_k}, k = 1, \ldots, K$, where $\mathbf{M}_{\mathcal{P}_k}$ is regarded as a kernel. If the commuting matrix is not a kernel (not positive semi-definite, PSD), we simply use the trick to remove the negative eigenvalues of the commuting matrix. Following [Gönen and Alpaydin, 2011; Vishwanathan *et al.*, 2010], we use the linear combination of kernels to form a new kernel:

$$\mathbf{M} = \sum_{k}^{K} \beta_k \mathbf{M}_{\mathcal{P}_k}, \quad (1)$$

where the weights $\beta_k \geq 0$ and satisfy $\sum_{k=1}^{K} \beta_k = 1$. In our application, we then use the p-norm multi-kernel learning framework [Vishwanathan *et al.*, 2010] to learn the weight of each meta-path [Hou *et al.*, 2017].

4 Experimental Results and Analysis

In this section, we evaluate the performance of our developed system *HinDroid* which integrates the above proposed method for automatic Android malware detection, based on the real sample collection from Comodo Cloud Security Center, which contains 1,834 training Android apps (920 benign and 914 malicious) and 500 testing samples (198 benign and 302 malicious). We use accuracy (ACC) and F1 measure [Hou *et al.*, 2017] for evaluations.

Based on the collected dataset, resting on the 200 extracted API calls and the three different kinds of relationships generated among them (*R1-R3* described in Section 3.1), we construct 16 meta-paths (*PID1-PID16* shown in Table 1) and compare their detection performances by using SVM. We

Method	PID1	PID2	PID3	PID4	PID5	PID6	PID7	PID8	PID9
ACC	0.944	0.950	0.942	0.904	0.940	0.942	0.846	0.850	0.812
F1	0.953	0.958	0.950	0.918	0.948	0.950	0.868	0.872	0.837
Method	PID10	PID11	PID12	PID13	PID14	PID15	PID16	PID17	PID18
ACC	0.866	0.908	0.846	0.918	0.826	0.816	0.846	0.968	**0.986**
F1	0.876	0.918	0.860	0.928	0.824	0.860	0.860	0.974	**0.988**

PID1: $\mathbf{AA}^T$; PID2: $\mathbf{ABA}^T$; PID3: $\mathbf{APA}^T$; PID4: $\mathbf{AIA}^T$; PID5: $\mathbf{ABPB}^T\mathbf{A}^T$; PID6: $\mathbf{APBP}^T\mathbf{A}^T$; PID7: $\mathbf{ABIB}^T\mathbf{A}^T$; PID8: $\mathbf{AIBI}^T\mathbf{A}^T$; PID9: $\mathbf{APIP}^T\mathbf{A}^T$; PID10: $\mathbf{AIPI}^T\mathbf{A}^T$; PID11: $\mathbf{ABPIP}^T\mathbf{B}^T\mathbf{A}^T$; PID12: $\mathbf{APBIB}^T\mathbf{P}^T\mathbf{A}^T$; PID13: $\mathbf{ABIPI}^T\mathbf{B}^T\mathbf{A}^T$; PID14: $\mathbf{AIBPB}^T\mathbf{I}^T\mathbf{A}^T$; PID15: $\mathbf{AIPBP}^T\mathbf{I}^T\mathbf{A}^T$; PID16: $\mathbf{APIBI}^T\mathbf{P}^T\mathbf{A}^T$; PID17: Combined-kernel (16); PID18: Multi-kernel (16) (i.e., ***HinDroid***).

Method	ANN-***f1***	NB-***f1***	DT-***f1***	SVM-***f1***	ANN-***f2***	NB-***f2***	DT-***f2***	SVM-***f2***	*HinDroid*
ACC	0.902	0.836	0.904	0.944	0.930	0.886	0.944	0.952	**0.986**
F1	0.917	0.851	0.920	0.953	0.941	0.903	0.954	0.959	**0.988**

f1: all the algorithms use original app features (i.e., API calls) as input;
f2: we simply put all HIN-related entities and relations as features for different algorithms to learn.

Table 1: Detection performance evaluation of *HinDriod* and comparisons with other alternative methods.

then use all the meta-paths as the kernels and apply multi-kernel learning (described in Section 3.3) for Android malware detection (i.e., $PID18$); for comparison, we also evaluate the combined similarity [Wang *et al.*, 2015] by using the Laplacian scores as the weights for the constructed meta-paths to construct a new kernel (i.e., $PID17$) fed to the SVM. From the results shown in Table 1, it can be observed that (1) different meta-paths indeed show different detection performance, as they provide different similarities with different semantic meanings; (2) by combining different meta-paths using Laplacian scores, it can improve the performance; (3) the method using multi-kernel learning successfully outperforms the single meta-paths and the unsupervised meta-path selection algorithm, i.e., Laplacian score, which demonstrates that multi-kernel learning can successfully filter out the meta-paths that do not perform well on the malware prediction problem while maintaining the "good" ones for final decision of malware detection.

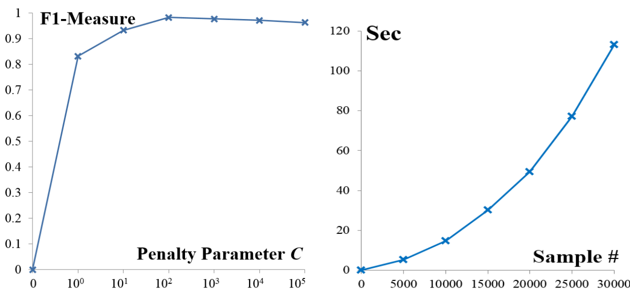

Figure 2: Parameter sensitivity. Figure 3: Scalability evaluation.

To further evaluate the detection performance of *HinDriod*, (i.e., the system integrating multi-kernel learning based on all the constructed 16 meta-paths - *PID18*), we also compare *HinDroid* with other four typical classification methods [Hou *et al.*, 2017] (i.e., Artificial Neural Network (ANN), Naive Bayes (NB), Decision Tree (DT), and SVM), based on the extracted API calls (denoted as ***f1***) and feature engineering (denoted as ***f2***, i.e., we simply put all HIN-related entities and relations as features for different algorithms to learn). From the results shown in Table 1, we can see that (1) feature engineering helps the performance of machine learning; however, (2) *HinDroid* further outperforms these alternative classification methods with feature engineering in Android malware detection. The reason behind this is that, in *HinDroid* we use more expressive representation for the data, and build the connection between the higher-level semantics of the data and the final results.

We also further evaluate the stability and scalability of *HinDroid*. Figure 2 shows the stability of *HinDroid* (i.e., using different values of the penalty parameter C ranging from 1 to 10^5 based on five-fold cross validations); while Figure 3 shows the scalability of *HinDroid* (i.e., training time with different sizes of the training data sets).

5 System Deployment and Operation

Due to its detection effectiveness and efficiency, our developed system *HinDroid* has already been incorporated into the scanning tool of Comodo's Mobile Security Product. *HinDroid* has been used to predict the daily sample collection from Comodo Cloud Security Center which contains over 15,000 unknown files per day. Note that Android malware techniques are constantly evolving and new malware samples are produced on a daily basis. To account for the temporal trends of Android malware writing, the training sets of our developed system are dynamically changing to include newly collected apps. Our system *HinDroid* has been deployed and tested based on the real daily sample collection for around half a year (about 2,700,000 Android apps in total that have either been trained or tested).

6 Conclusion

To combat the Android malware threats, our preliminary works [Hou *et al.*, 2016a; 2016b; Chen *et al.*, 2017a; 2017b;

Ye *et al.*, 2017a] have attempted for automatic Andriod malware detection. To make the attackers evade the detection harder, in this paper, we present a novel Android malware detection framework *HinDroid*, which introduces a structured HIN representation of Android apps and employ multi-kernel learning to aggregate different similarities formulated by different meta-paths over HIN for classification. Due to its detectionn effectiveness and efficiency, our developed system *HinDroid* has been incorporated into the scanning tool of Comodo Mobile Security product.

Acknowledgments

The authors would also like to thank the anti-malware experts of Comodo Security Lab for the data collection as well as helpful discussions and supports. The work of S. Hou and Y. Ye is supported by the U.S. NSF Grant (CNS-1618629), WV HEPC Grant (HEPC.dsr.18.5) and WVU Research and Scholarship Advancement Grant (844). The work of Y. Song is supported by China 973 Fundamental R&D Program (No.2014CB340304).

References

[Burguera *et al.*, 2011] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. Crowdroid: Behavior-based malware detection system for android. In *SPSM*, 2011.

[Chen *et al.*, 2017a] Lingwei Chen, Shifu Hou, and Yanfang Ye. Securedroid: Enhancing security of machine learning-based detection against adversarial android malware attacks. In *ACSAC*, 2017.

[Chen *et al.*, 2017b] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *EISIC*, 2017.

[Dimjasevic *et al.*, 2016] Marko Dimjasevic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. Evaluation of android malware detection based on system calls. In *IWSPA*, 2016.

[Felt *et al.*, 2011] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. A survey of mobile malware in the wild. In *SPSM*, 2011.

[Gönen and Alpaydin, 2011] Mehmet Gönen and Ethem Alpaydin. Multiple kernel learning algorithms. *Journal of Machine Learning Research*, 12:2211–2268, 2011.

[Hou *et al.*, 2016a] Shifu Hou, Aaron Saas, Lifei Chen, and Yanfang Ye. Deep4maldroid: A deep learning framework for android malware detection based on linux kernel system call graphs. In *WIW*, pages 104–111, 2016.

[Hou *et al.*, 2016b] Shifu Hou, Aaron Saas, Yanfang Ye, and Lifei Chen. Droiddelver: An android malware detection system using deep belief network based on api call blocks. In *WAIM*, pages 54–66, 2016.

[Hou *et al.*, 2017] Shifu Hou, Yanfang Ye, Yangqiu Song, and Melih Abdulhayoglu. Hindroid: An intelligent android malware detection system based on structured heterogeneous information network. In *SIGKDD*, 2017.

[IDC, 2017] IDC. *Smartphone OS Market Share*, 2017. https://www.idc.com/promo/smartphone-market-share/os.

[Peiravian and Zhu, 2013] Naser Peiravian and Xingquan Zhu. Machine learning for android malware detection using permission and api calls. In *IEEE ICTAI*, pages 300–305, Nov 2013.

[Sun and Han, 2012] Yizhou Sun and Jiawei Han. Mining heterogeneous information networks: principles and methodologies. *Synthesis Lectures on Data Mining and Knowledge Discovery*, 3(2):1–159, 2012.

[Sun *et al.*, 2011] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *PVLDB*, pages 992–1003, 2011.

[Tam *et al.*, 2015] Kimberly Tam, Salahuddin J. Khan, Aristide Fattori, and Lorenzo Cavallaro. Copperdroid: Automatic reconstruction of android malware behaviors. In *NDSS*, 2015.

[Vishwanathan *et al.*, 2010] S. V. N. Vishwanathan, Zhaonan sun, Nawanol Ampornpunt, and Manik Varma. Multiple kernel learning and the SMO algorithm. In *NIPS*, pages 2361–2369, 2010.

[Wang *et al.*, 2015] Chenguang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. Knowsim: A document similarity measure on structured heterogeneous information networks. In *ICDM*, pages 1015–1020, 2015.

[Wood, 2015] P. Wood. Internet security threat report 2015. In *Symantec*, 2015.

[Wu and Hung, 2014] Wen-Chieh Wu and Shih-Hao Hung. Droiddolphin: A dynamic android malware detection framework using big data and machine learning. In *RACS*, 2014.

[Wu *et al.*, 2012] Dong-Jie Wu, Ching-Hao Mao, Te-En Wei, Hahn-Ming Lee, and Kuo-Ping Wu. Droidmat: Android malware detection through manifest and api calls tracing. In *Asia JCIS*, pages 62–69, Aug 2012.

[Yang *et al.*, 2014] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. Droidminer: Automated mining and characterization of fine-grained malicious behaviors in android applications. In *19th European Symposium on Research in Computer Security*, pages 163–182, 2014.

[Ye *et al.*, 2011] Yanfang Ye, Tao Li, Shenghuo Zhu, Weiwei Zhuang, Egemen Tas, Umesh Gupta, and Melih Abdulhayoglu. Combining file content and file relations for cloud based malware detection. In *SIGKDD*, pages 222–230, 2011.

[Ye *et al.*, 2017a] Yanfang Ye, Lingwei Chen, Shifu Hou, William Hardy, and Xin Li. Deepam: A heterogeneous deep learning framework for intelligent malware detection. *Knowledge and Information Systems*, pages 1–21, 2017.

[Ye *et al.*, 2017b] Yanfang Ye, Tao Li, Donald Adjeroh, and S Sitharama Iyengar. A survey on malware detection using data mining techniques. *ACM Computing Surveys*, 50(3):41, 2017.