HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network

Shifu Hou, Yanfang Ye *
Department of CSEE
West Virginia Univerisity, WV, USA
{shhou,yanfang.ye}@mail.wvu.edu

Yangqiu Song
Department of CSE
HKUST, Hong Kong
yqsong@cse.ust.hk

Melih Abdulhayoglu
Comodo Security Solutions, Inc.
Clifton, NJ, USA
melih@comodo.com

ABSTRACT

With explosive growth of Android malware and due to the severity of its damages to smart phone users, the detection of Android malware has become increasingly important in cybersecurity. The increasing sophistication of Android malware calls for new defensive techniques that are capable against novel threats and harder to evade. In this paper, to detect Android malware, instead of using Application Programming Interface (API) calls only, 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. Then each meta-path is automatically weighted by the learning algorithm to make predictions. To the best of our knowledge, this is the first work to use structured HIN for Android malware detection. Comprehensive experiments on real sample collections from Comodo Cloud Security Center are conducted to compare various malware detection approaches. Promising experimental results demonstrate that our developed system *HinDroid* outperforms other alternative Android malware detection techniques.

CCS CONCEPTS

•**Artifficial Intelligence** →**General;** •**Database applications** →*Data mining;* •**Security and Protection** →Invasive Software;

KEYWORDS

Android Malware Detection; Application Programming Interface Calls; Relation Analysis; Heterogeneous Information Network.

ACM Reference format:
Shifu Hou, Yanfang Ye *, Yangqiu Song, and Melih Abdulhayoglu. 2017. HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network. In *Proceedings of KDD'17, August 13-17, 2017, Halifax, NS, Canada, ,* 9 pages.
DOI: 10.1145/3097983.3098026

1 INTRODUCTION

Smart phones have been widely used in people's daily life, such as online banking, automated home control, and entertainment. Due to the mobility and ever expanding capabilities, the use of smart phones has experienced an exponential growth rate in recent years. It is estimated that 77.7% of all devices connected to the Internet will be smart phones in 2019 [13], leaving PCs falling behind at 4.8%. Android, as an open source and customizable operating system for smart phones, is currently dominating the smart phone market by 82.8% [4]. However, 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. Because of the lack of trustworthiness review methods, developers can upload their Android apps including repackaged apps, ransomware [5], or trojans to the market easily in even Google's official Android market. The presence of other third-party Android markets (e.g., Opera Mobile Store, Wandoujia) makes this problem worse. Many examples of Android malware have already been released in the market (e.g., Geinimi, DriodKungfu and Lotoor) [31] which posed serious threats to the smart phone users, such as stealing user credentials, auto-dialing premium numbers, and sending SMS messages without user's concern [9]. According to Symantec's Internet Security Threat Report [28], one in every five Android apps (nearly one million total) were actually malware. To protect legitimate users from the attacks of Android malware, currently, the major defense is mobile security products, such as Norton, Lookout and Comodo Mobile Security, which mainly use the signature-based method to recognize threats. However, attackers can easily use techniques, such as code obfuscation and repackaging, to evade the detection. The increasing sophistication of Android malware calls for new defensive techniques that are robust and capable of protecting users against novel threats.

To be more resilient against the Android malware's evasion tactics, in this paper, instead of using Application Programming Interface (API) calls only, we further analyze the relationships among them, e.g., whether the extracted API calls belong to the same code block [13], 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) [11, 18] representation to depict apps and APIs. Then we use meta-path [19] to incorporate higher-level semantics to build up the semantic relatedness of apps. In this way, a similarity between two

KDD'17, August 13-17, 2017, Halifax, NS, Canada
© 2017 ACM. 978-1-4503-4887-4/17/08…$15.00
DOI: 10.1145/3097983.3098026

apps can not only capture whether they are using the same sets of APIs but also capture whether the APIs have similar usage patterns, such as in the same package. 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 [23] to automatically learn the weights of different similarities from data. In short, our developed system called *HinDroid* has the following major traits:

- ***Novel structural feature representation:*** Instead of using API calls only, we further analyze the relationships among them. Based on the extracted features, the Android apps will be represented by a structural heterogeneous information network (HIN), and a meta-path based approach will be used to link the apps. In this way, the detection of a malicious Android app is an aggregation of different similarities defined by different meta-paths. This is much more complicated than traditional approaches and is more difficult and costly to be evaded.
- ***Multi-kernel learning for HIN:*** HIN is a conceptual representation of many other kinds of data, e.g., social networks, scholar networks, knowledge graphs, etc. The similarities defined by different meta-paths can be used to make decisions in an aggregated way. In this paper, we propose a multi-kernel learning to learn from data to determine the importance of different meta-paths. This is a very natural way to handle HIN based similarities but to our best knowledge is a first attempt.
- ***A practical developed system for real industry application:*** We develop a practical system *HinDroid* for automatic Android malware detection and provide a comprehensive experimental study based on the real sample collection from Comodo Cloud Security Center, which demonstrates the effectiveness and efficiency of our developed system. *HinDroid* has already been incorporated into the scanning tool of Comodo Mobile Security product. The system has been deployed and tested based on the real daily sample collection (over 15,000 Android apps per day) for around half a year (about 2,700,000 Android apps in total).

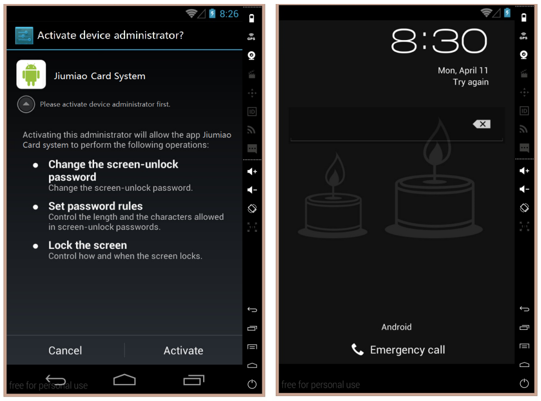
The remainder of this paper is organized as follows. Section 2 presents the overview of our system architecture. Section 3 introduces our proposed method in detail. In Section 4, based on the real sample collection from Comodo Cloud Security Center, we systematically evaluate the performance of our developed system *HinDroid* which integrates our proposed method, in comparison with other alternative detection methods and some of the popular Mobile Security products (e.g., Lookout, Norton Mobile Security). Section 5 presents the details of system development and operation. Section 6 discusses the related work. Finally, Section 7 concludes.

2 SYSTEM OVERVIEW

In this section, we present our system overview with preliminaries.

2.1 Preliminaries

Unlike traditional desktop based Portable Executable (PE) files, ***Android app*** is compiled and packaged in a single archive file (with an .apk suffix) that includes the app code (.dex file), resources, assets, and manifest file. ***Dex*** (i.e., Dalivik executable) is a file format which contains compiled code written for Android and can be interpreted by the DalvikVM [2], but it is unreadable. In order to analyze the Android apps, we need to convert the dex file to a readable format. ***Smali*** is an assembler/disassembler for the dex format [2], which provides us readable code in smali language. ***Smali code*** is the intermediate but interpreted code between Java and DalvikVM [3]. Listing 1 shows a segment of the converted smali code from a ransomware "Locker.apk" (MD5: *f836f5c6267f13bf9f6109a6b8d79175*) that will lock smart phone user's screen (shown in Figure 1(b)) after the installation (shown in Figure 1(a)). If the smart phone is infected by this malware, the victim is demanded to pay a ransom to attackers to unlock the smart phone.



Please activate device administrator first.

Activating this administrator will allow the app Jiumiao Card system to perform the following operations:

- **Change the screen-unlock password**
  Change the screen-unlock password.
- **Set password rules**
  Control the length and the characters allowed in screen-unlock passwords.
- **Lock the screen**
  Control how and when the screen locks.

(a) (b)

Figure 1: Screen shots of the ransomware "Locker.apk"

Listing 1: An example of smali code

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

2.2 System Architecture

In this paper, to analyzed the collected Android apps, we first unzip each Android Application Package (APK) to get the dex file, and then generate the smali codes by decompiling the dex file. By analyzing the smali codes, a complete Android API call list will be extracted, and then the relationships among the extracted API calls will be further analyzed. Figure 2 shows the system overview of our developed Android malware detection system *HinDroid*, which consists of the following five major components.

- ***Unzipper and Decompiler:*** The APKTool [1] is used to unzip the APKs and decompile the dex files to smali codes.
- ***Feature Extractor:*** It automatically extracts the API calls from the decompiled smali codes. The API calls extracted from the smali codes will be converted to a group of global integer IDs which represents the static execution sequence of the corresponding API calls. Based on the extracted API calls, the relationships among them will be further analyzed, i.e., whether

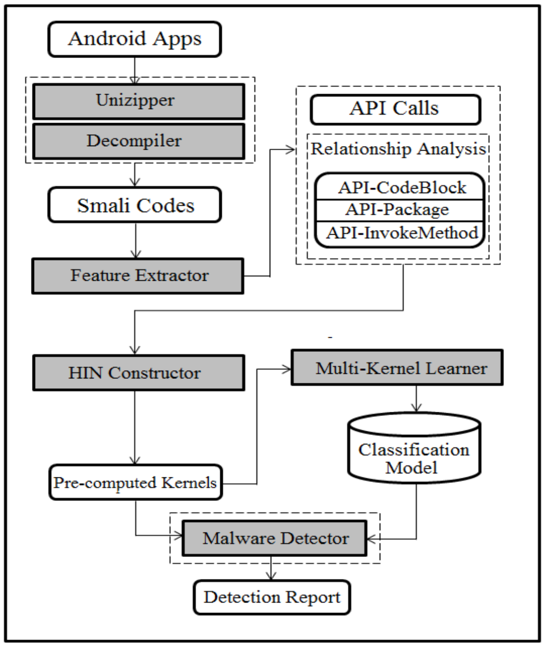


Figure 2: System architecture of *HinDroid*

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 either benign or malicious.

3 PROPOSED METHOD

In this section, we introduce the detailed approaches of how we represent the Android apps, and how to solve the classification problem based on the representation.

3.1 Feature Extraction

3.1.1 API Call Extraction. API calls are used by the Android apps in order to access operating system functionality and system resources. Therefore, they can be used as representations of the behaviors of an Android app. In order to extract the API calls, the Android app is first unzipped to provide the dex file, and then the dex file is further decompiled into smali codes using a well-known reverse engineering tool APKTool [1]. The converted smali codes can then be parsed for API call extraction. For example, in the smali code segment as 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.

3.1.2 Relationship Analysis among the Extracted API Calls. 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. 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. After the extraction of the API calls from the converted smali codes, 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. For example, the API calls in the package of "Lorg/apache/http/HttpRequest" are related to Internet connection. The API calls co-appear in the same package indicate strong relations among them. 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. As the example shown in Listing 1, both API "*Ljava/lang/Runtime; → getRuntime() Ljava/lang/Runtime*" and API "*Ljava/lang/Runtime; → exec (Ljava/lang/String;) Ljava/lang/Process*" are from the same package "*Ljava/lang/Runtime*", so the element representing the relation of these two APIs in the matrix will be set to 1.

In the smali code, there are five different methods to invoke an API call [2]: (1) *invoke-static*: invokes a static method with parameters; (2) *invoke-virtual*: invokes a virtual method with parameters; (3) *invoke-direct*: invokes a method with parameters without the virtual method resolution; (4) *invoke-super*: invokes the virtual method of the immediate parent class; and (5) *invoke-interface*: invokes an interface method. Since the same invoke method can show the common properties of the API calls (like the words have the same part of speech), two API calls using the same invoke method may indicate specifically implicit relations 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. To further illustrate, as shown in Listing 1, API calls "*Ljava/lang/Runtime; → exec (Ljava/lang/String;) Ljava/lang/Process*" and "*Ljava/lang/Process; → getInputStream() Ljava/io/InputStream*" both use *invoke-virtual* method, so the element denoting the relation of these two APIs in the matrix will be set to 1.

A summary of the description of different relations and their elements in the relation matrices is shown in Table 1. 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).

Table 1: Description of each matrix

G	Element	Description
A	a_{ij}	If app_i contains API_j, then $a_{ij} = 1$; otherwise, $a_{ij} = 0$.
B	b_{ij}	If API_i and API_j co-exist in the same code block, then $b_{ij} = 1$; otherwise, $b_{ij} = 0$.
P	p_{ij}	If API_i and API_j are with the same package name, then $p_{ij} = 1$; otherwise, $p_{ij} = 0$.
I	i_{ij}	If API_i and API_j use the same invoke method, then $i_{ij} = 1$; otherwise, $i_{ij} = 0$.

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. Thus, in this section, we introduce how to use heterogeneous information network to represent the Andriod apps by using the features (including API calls and the relationships among them) extracted above. We first introduce some concepts related to heterogeneous information networks as follow.

Definition 3.1. [18] 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. There are four types of relations, e.g., app containing API call, API calls in the same code block, API calls with the same package name, and API calls with the same invoke type. 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 [19] was used in the concept of HIN to formulate the semantics of higher-order relationships among entities. Here we follow this concept and extend it to our *HinDroid* framework.

Definition 3.2. [19] 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 for apps is $App \xrightarrow{\text{contains}} API \xrightarrow{\text{contains}^{-1}} App$. This means that we want to connect two apps through the path containing the same API over the HIN. There can be multiple API calls satisfying this meta-path constraint. Thus, we use the following commuting matrix [19] to give a general form to compute entity similarities using a particular meta-path.

Definition 3.3. [19] Given a network $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and its network schema $\mathcal{T}_\mathcal{G}$, a **commuting matrix** $\mathbf{M}_\mathcal{P}$ for a meta-path $\mathcal{P} = (A_1 - A_2 - \ldots - A_{L+1})$ is defined as $\mathbf{M}_\mathcal{P} = \mathbf{G}_{A_1A_2}\mathbf{G}_{A_2A_3}\ldots\mathbf{G}_{A_LA_{L+1}}$, where $\mathbf{G}_{A_iA_j}$ is the adjacency matrix between types A_i and A_j. $\mathbf{M}_\mathcal{P}(i, j)$ represents the number of path instances between entities $x_i \in A_1$ and $y_j \in A_{L+1}$ under the meta-path $\mathcal{P}$.

For example, the adjacency matrix between apps and API calls is $\mathbf{G}_{App,API}$. Then the commuting matrix of apps computed using the meta-path $App \xrightarrow{\text{contains}} API \xrightarrow{\text{contains}^{-1}} App$ is $\mathbf{G}_{App,API}\mathbf{G}^T_{App,API}$, which is $\mathbf{A}\mathbf{A}^T$. If we denote $\mathbf{a}_i^T$ as the ith row of the matrix $\mathbf{A}$, then the similarity between app i and j is given by $\mathbf{a}_i^T\mathbf{a}_j$, which is simply the dot product of two feature vectors. Each feature vector can be regarded as using a bag-of-APIs to represent an app.

More complicated similarities can be defined by commuting matrix based on the meta-path. Here we only consider the symmetric meta-path since we only focus on apps similarity. For example, we can define a meta-path $App \xrightarrow{\text{contains}} API \xrightarrow{\text{same code block}} API \xrightarrow{\text{contains}^{-1}} App$. This means that we compute the similarity between two apps not only considering API calls inside, but also considering the type of API calls inside. In this example, we use the bag-of-APIs in the same code block as features, and then use the dot product to compute the similarities.

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. Thus, an intuitive way is to combine different meta-paths. Since we are using a supervised learning approach to learn from examples of Android malware, and HIN can naturally

provide
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 [10, 16, 23], we use the linear combination of kernels to form a new kernel:

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

where the weights $\beta_k \geq 0$ and satisfy $\sum_{k=1}^{K} \beta_k = 1$.

To learn the weight of each meta-path, we assume we have a set of labeled data $\{x_i, y_i\}_{i=1}^{N}$, where x_i is the app (here we can regard x_i as an ID), and $y_i \in \{+1, -1\}$ is the label. Then we use the p-norm multi-kernel learning framework [23] with following objective function to learn the parameters:

$$\begin{aligned} \min_{\mathbf{w}>0, \xi_i \geq 0, \beta_k \geq 0} \quad & \frac{1}{2} \sum_{k} ||\mathbf{w}_k||^2 / \beta_k + C \sum_{i} \xi_i + \frac{\lambda}{2} \left(\sum_{k} \beta_k^p \right)^{2/p}, \\ s.t. \quad & y_i \left(\sum_{k} \mathbf{w}_k^T \phi_k(x_i) + b \right) \geq 1 - \xi_i, \end{aligned} \tag{2}$$

where for each kernel we learn a parameter vector $\mathbf{w}_k$. For each data $\{x_i, y_i\}$, the slack parameter ξ_i is introduced to allow mis-classification. $\phi_k(x_i)$ is the nonlinear mapping of features in the Hilbert space that defines the kernel, where $\phi_k(x_i)^T \phi_k(x_i) = \mathbf{M}_{\mathcal{P}_k}(i, j)$. Then by applying the representation theorem, we have $\mathbf{w}_k = \sum_i \alpha_i \phi_k(x_i)$. α_i can be solved using the dual formulation, and non-zero α_i's lead to the support vectors.

In multi-kernel learning framework, another set of parameters besides $\mathbf{w}_k$ is β_k. Here the p-norm $\left(\sum_k \beta_k^p\right)^{2/p}$ is used to regularize the optimization of β_k's. Empirically we found 2-norm is the best, and apply it to our problem throughout the paper. After the optimization, the weights β_k's are optimized to reveal the importance of the meta-paths serving as kernels. For a new app x coming, $\sum_k \mathbf{w}_k \phi_k(x) + b$ is used to evaluate whether it is malicious or not.

4 EXPERIMENTAL RESULTS AND ANALYSIS

In this section, we show four sets of experimental studies using real sample collections obtained from Comodo Cloud Security Center to fully evaluate the performance of our developed Android malware detection system *HinDroid*: (1) In the first set of experiments, we evaluate the detection performance of our proposed method; (2) In the second set of experiments, we evaluate our developed system *HinDroid* which integrates our proposed method by comparisons with other alternative classification methods in Android malware detection; (3) In the third set of experiments, we compare the detection performance of *HinDroid* with other commercial mobile security products; (5) In the last set of experiments, we systematically evaluate our developed system *HinDroid* in real industry for Android malware detection.

4.1 Experimental Setup

We obtain two datasets from Comodo Cloud Security Center: (1) The first sample set includes recent collected Android apps (through January 30, 2017 to February 5, 2017), which contains 1,834 training Android apps (920 of them are benign apps, while the other 914 apps are malware including the families of Lotoor, RevMob, Fakegupdt, and GhostPush, etc), and 500 testing samples (with the analysis by the anti-malware experts of Comodo Security Lab, 198 of them are labeled as benign and 302 of them are labeled as malicious). (2) The second dataset has larger sample collection containing 30,000 Android apps obtained within one month (Januray 2017), half of which are benign apps and the half are malicious apps. We evaluate the Android malware detection performance of different methods using the measures shown in Table 2.

Table 2: Performance indices of Android malware detection

Indices	Description
TP	# of apps correctly classified as malicious
TN	# of apps correctly classified as benign
FP	# of apps mistakenly classified as malicious
FN	# of apps mistakenly classified as benign
$Precision$	$TP/(TP + FP)$
$Recall$	$TP/(TP + FN)$
ACC	$(TP + TN)/(TP + TN + FP + FN)$
$F1$	$2 * Precision * Recall/(Precision + Recall)$

4.2 Detection Performance Evaluation of the Proposed Method

In this set of experiments, based on the first sample set described in Section 4.1, resting on the 200 extracted API calls and the three different kinds of relationships generated among them (*R1, R2, R3*) (as described in Section 3.1), we construct 16 meta-paths (shown in Table 3) and compare their detection performances by using Support Vector Machine (SVM). Table 1 gives the description of each matrix which forms different meta-paths. We also evaluate the combined similarity [25, 26] by selecting the meta-paths using Laplacian score [12]. We rank each meta-path using its Laplacian score. The order of the ranking is: $PID12 \rightarrow PID16 \rightarrow PID6 \rightarrow PID3 \rightarrow PID5 \rightarrow PID11 \rightarrow PID9 \rightarrow PID2 \rightarrow PID8 \rightarrow PID7 \rightarrow PID13 \rightarrow PID14 \rightarrow PID15 \rightarrow PID10 \rightarrow PID4 \rightarrow PID1$. We select the top five meta-paths (i.e., $PID12$, $PID16$, $PID6$, $PID3$, and $PID5$) and use their Laplacian scores as the weights to construct a new kernel (i.e., $PID17$) fed to the SVM. Similar to multi-kernel learning, if the similarity matrix is not PSD, we remove the negative eigenvalues following [26]. We also use these top five meta-paths as the kernels and apply multi-kernel learning (described in Section 3.3) for comparison (i.e., $PID19$). Another set of comparison is using all the meta-paths (i.e., $PID18$ with combined similarity and $PID20$ by applying multi-kernel learning). The experimental results are shown in Table 3.

From Table 3 we can see that different meta-paths indeed show different detection performance. For example, some meta-paths,

Table 3: Detection performance evaluation

PID	Method	F1	β	ACC	TP	FP	TN	FN
1	$\mathbf{AA}^T$	0.9529	0.1069	94.40%	283	19	189	19
2	$\mathbf{ABA}^T$	0.9581	0.0900	95.00%	286	9	189	16
3	$\mathbf{APA}^T$	0.9495	0.0858	94.20%	273	0	198	29
4	$\mathbf{AIA}^T$	0.9183	0.0623	90.40%	270	16	182	32
5	$\mathbf{ABPB}^T\mathbf{A}^T$	0.9479	0.0670	94.00%	273	1	197	29
6	$\mathbf{APBP}^T\mathbf{A}^T$	0.9502	0.0565	94.20%	277	4	194	25
7	$\mathbf{ABIB}^T\mathbf{A}^T$	0.8683	0.0639	84.60%	254	29	169	48
8	$\mathbf{AIBI}^T\mathbf{A}^T$	0.8722	0.0639	85.00%	256	29	169	46
9	$\mathbf{APIP}^T\mathbf{A}^T$	0.8373	0.0445	81.20%	242	34	164	60
10	$\mathbf{AIPI}^T\mathbf{A}^T$	0.8761	0.0572	86.60%	237	2	196	65
11	$\mathbf{ABPIP}^T\mathbf{B}^T\mathbf{A}^T$	0.9184	0.0616	90.80%	259	3	195	43
12	$\mathbf{APBIB}^T\mathbf{P}^T\mathbf{A}^T$	0.8597	0.0617	84.60%	236	11	187	66
13	$\mathbf{ABIPI}^T\mathbf{B}^T\mathbf{A}^T$	0.9284	0.0426	91.80%	266	5	193	36
14	$\mathbf{AIBPB}^T\mathbf{I}^T\mathbf{A}^T$	0.8237	0.0426	82.60%	218	3	195	84
15	$\mathbf{AIPBP}^T\mathbf{I}^T\mathbf{A}^T$	0.8597	0.0469	81.60%	215	5	193	87
16	$\mathbf{APIBI}^T\mathbf{P}^T\mathbf{A}^T$	0.8597	0.0458	84.60%	236	11	187	66
17	Combined-kernel (5)	0.9214	——	91.20%	258	0	198	44
18	Combined-kernel (16)	0.9740	——	96.80%	300	14	184	2
19	Multi-kernel (5)	0.9834	——	98.00%	297	5	193	5
20	Multi-kernel (16)	**0.9884**	——	**98.60%**	299	4	194	3

e.g., $\mathbf{AA}^T$, $\mathbf{ABA}^T$, and $\mathbf{APBP}^T\mathbf{A}^T$, perform well on the test set. Some other meta-paths do not perform well on their own, such as $\mathbf{AIBPB}^T\mathbf{I}^T\mathbf{A}^T$, which may be because the semantics of the meta-path does not reflect the problem of Android malware detection well. However, when we combine these meta-paths to others, they still help to improve the classification results.

Laplacian score indeed helps us select some important meta-paths. For example, among *PID*12, *PID*16, *PID*6, *PID*3, and *PID*5, the meta-paths for *PID*6, *PID*3, and *PID*5 are very good. However, the weights used for combining the meta-paths are the Laplacian scores which may not be the best to reflect classification property. From the result we can see that "Combined-kernel (5)" for test set is with 91.20% detection accuracy and "Combined-kernel (16)" is 96.80%. This shows that by combining different meta-paths using Laplacian score, it can also improve the performance.

Finally, the method using multi-kernel learning successfully outperforms the single meta-paths and the unsupervised meta-path selection algorithm, i.e., Laplacian score. From the results we can see that, both "Multi-kernel (5)" over the five selected ones by Laplacian score and "Multi-kernel (16)" performed very well. To demonstrate the effectiveness of multi-kernel learning, we further show the correlation between the learned parameter β_k weighting each meta-path in multi-kernel learning algorithm, shown in Eq. (1), and the actual performance of each meta-path in Table 3 and Figure 3. We can see that β_k's can successfully filter out the meta-paths that do not perform well on the malware prediction problem while maintaining the "good" meta-paths for final decision of malware detection.

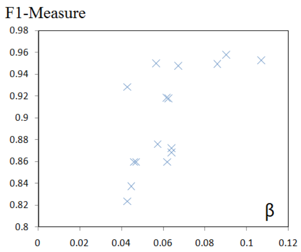


Figure 3: β_k and F1 correlation.

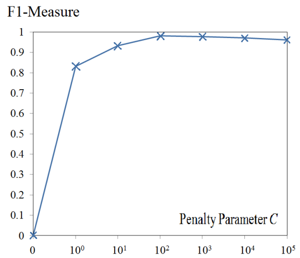


Figure 4: Parameter sensitivity evaluation.

Remark: In Figure 3, β_k is the parameter learned by multi-kernel learning, shown in Eq. (1). F1 is the actual performance of SVM using each meta-path as kernel.

Since HIN construction in our framework can be fully automated, the only parameter we need to tune is in multi-kernel learning. In this experiment, based on five-fold cross validation, we show the results using different values of the penalty parameter C ranging from 1 to 10^5. Figure 4 shows the stability of *HinDroid* with different parameters. From the figure, we can see that in a wide range of numbers, our algorithm is stable and not very sensitive to the penalty parameter. This indicates that for practical use, we can simply tune a parameter using some training data based on cross-validation, and apply that parameter to the test set without concerning the change of the parameter affecting the online performance.

4.3 Comparisons of HinDroid and other Alternative Detection Methods

In this set of experiments, based on the first sample set described in Section 4.1, we compare *HinDroid* (i.e., the system integrating multi-kernel learning based on all the constructed 16 meta-paths) with four other typical classification methods, i.e., Artificial Neural Network (ANN), Naive Bayes (NB), Decision Tree (DT), and Support Vector Machine (SVM) based on the extracted API calls as well as feature engineering based on the discussion in Section 3.1. For ANN, we use 3 hidden layers (500 neurons in each hidden layer) and train the network using back propagation. The learning rate is set to 0.3 and the momentum is set as 0.5. For SVM, we use LibSVM in our experiment and the penalty is empirically set to be 1,000.

The experiment results are shown in Table 4. From the results we can see that feature engineering helps the performance of machine learning, since the rich semantics encoded in different types of relations can bring more information. However, the use of this information for traditional machine learning algorithms is simply flat features, i.e., concatenation of different features altogether. From Table 4, we can see that *HinDroid* further outperforms these alternative classification methods with feature engineering in Android malware detection. In Table 4, we also show detailed information of true-positive and false-positive numbers for different algorithms. It's shown that our *HinDroid* algorithm can significantly reduce the numbers. To check whether the overall improvement is significant, we also run 30 random trials of training and testing examples to compare *HinDroid* and SVM with feature engineering, and the probability associated with a paired t-Test with a two-tailed distribution is 1.62×10^{-13}. This shows that *HinDroid* is significantly better than the best baseline method we compared. 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. This again demonstrates that using *HinDroid* can reduce the work of feature engineering, and significantly improve the Android malware detection performance.

4.4 Comparisons of HinDroid and other Commercial Mobile Security Products

In this set of experiments, to evaluate the detection performance of *HinDroid*, based on the first sample set described in Section 4.1, we also compare it with some other popular commercial mobile security products (e.g., Clean Master (CM), Lookout and Norton Mobile Security). For the comparisons, we use all the latest versions of the mobile security products (i.e., Clean Master (CM): 2.08, Lookout: 10.9-7f33b3e, and Norton: 3.17.0.3205).

Table 5 shows different detection results from *HinDroid* and other mobile security products. From Table 5, we can see that *HinDroid* outperforms others in the detection of the most recent collected Android malware from different families (e.g., Lotoor, RevMob, and GhostPush, etc.). The success of *HinDroid* may lie in its novel higher-level semantic feature representations as well as the multi-kernel learning based on the constructed HIN in feature engineering. Besides, *HinDroid* also has high detection efficiency: the prediction of an Andriod app is around 3-5 seconds on average, including the feature extraction.

Table 4: Comparisons between HinDroid and alternative detection methods. "Original" means all the algorithms use original app features (i.e., API calls) as input. "Augmented" means that, we simply put all HIN-related entities and relations as features for different algorithms to learn.

Original	F1	AUC	ACC	TP	FP	TN	FN
ANN-1	0.9173	0.9023	90.20%	272	19	179	30
NB-1	0.8514	0.8511	83.60%	235	15	183	67
DT-1	0.9202	0.9005	90.40%	277	23	175	25
SVM-1	0.9529	0.9458	94.40%	283	9	189	19
Augmented	F1	AUC	ACC	TP	FP	TN	FN
ANN-2	0.9409	0.9316	93.00%	279	12	186	23
NB-2	0.9025	0.8891	88.60%	264	19	179	38
DT-2	0.9539	0.9397	94.40%	290	16	182	12
SVM-2	0.9590	0.9537	95.20%	281	7	191	17
HinDroid	**0.9884**	**0.9849**	**98.60%**	299	4	194	3

Table 5: Comparisons with other mobile security products

Family	Sample #	Norton	Lookout	CM	***HinDroid***
Lotoor	78	75	74	76	78
RevMob	52	46	50	48	52
Malapp	33	29	32	30	33
Fakebank	31	29	30	29	30
Generisk	29	29	29	29	29
GhostPush	19	15	16	18	18
Fakegupdt	16	15	14	14	16
Danpay	21	19	20	20	21
HideIcon	12	11	9	8	12
Idownloader	11	10	9	9	10
Total	302	278	283	281	**299**
DetectionRate	–	92.05%	93.71%	93.05%	**99.01%**

4.5 Evaluations Based on Large and Real Sample Collection from Industry

In this experiment, based on a real and larger data collection from Comodo Cloud Security Center (i.e., 30,000 Android apps obtained within one month (Januray 2017), half of which are benign apps and the half are malicious apps), we systematically evaluate the performance of our developed system *HinDroid*, including the detection effectiveness and scalability.

Figure 5 shows the overall and zoomed-in receiver operating characteristic (ROC) curves for this experiment based on the ten-fold cross validations. From Figure 5, we can see that *HinDroid* achieves an impressive 0.9833 TP rate at the 0.0087 FP rate while labeling the newly collected Android apps.

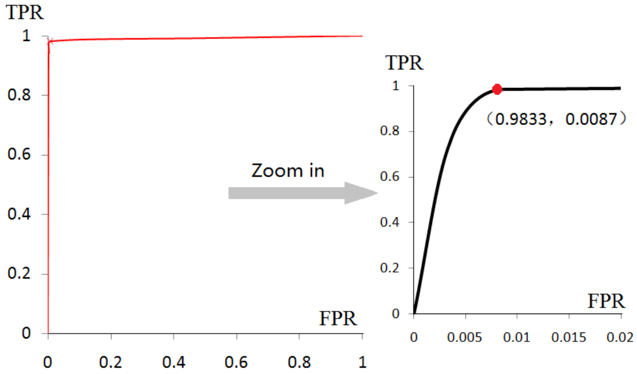


Figure 5: *Left*: ROC curve of HinDroid, *Right*: Zoomed-in.

We also further evaluate the training time of *HinDroid* with different sizes of the training data sets. Figure 7 shows the scalability of our proposed method. It is shown that similar to other kernel methods, the multi-kernel learning is quadratic to the number of data samples. When dealing with more data, approximation or parallel algorithms should be developed. However, as shown in Figure 6, for such Android malware detection problem, the need of more labels is not as important as the need of more expressive representations of data. Therefore, for practical use, our approach is still feasible.

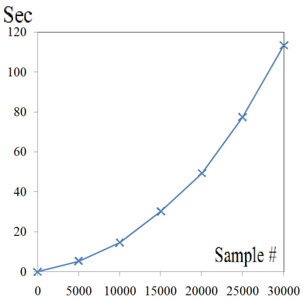


Figure 6: Scalability evaluation of *HinDroid*

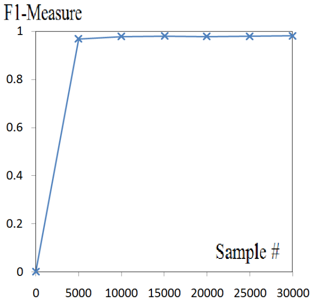


Figure 7: Comparisons when training data sizes vary

5 SYSTEM DEPLOYMENT AND OPERATION

By adopting the proposed methods, 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 have either been trained or tested).

For the development of the system, Comodo has spent over $250K, including hardware equipment and human resource investment. Due to the high detection efficiency and effectiveness, the developed system *HinDroid* can greatly save human labors and reduce the staff cost: over 50 anti-malware analysts at Comodo Cloud Security Center are utilizing the system on the daily basis. In practice, an anti-malware analyst has to spend at least 8 hours to manually analyze 40 Android apps for malware detection. Using the developed system *HinDroid*, the analysis of about 15,000 file samples can be performed within minutes with multiple servers. This would benefit over 10 million smart phone users of Comodo's Mobile Security product.

6 RELATED WORK

In recent years, there have been research studies on developing intelligent Android malware detection systems using machine learning and data mining techniques [6–8, 29, 30]. DroidDolphin [30] used a dynamic analysis framework including DroidBox and APE to record thirteen activity features from the collected Android apps, and then applied Support Vector Machine (SVM) to build a malware prediction model. Crowdroid [6] also performed dynamic analysis for Android malware detection which extracted API system calls as the feature set for k-means clustering. CopperDroid [22], an automatic Virtual Machine Introspection (VMI) based dynamic analysis system, extracted operating system interactions (e.g., file and process creation), as well as intra- and inter-process communications (e.g., SMS reception) as the features to represent the behaviors of the Android apps. Though dynamic extraction is more resilient to low level obfuscation, it is computationally expensive to perform and requires simulation of user interactions. On the contrast, static analysis focuses on analyzing the internal components of an app without executing it. This makes it much cheaper to perform than dynamic analysis. DroidMat [29] performed static analysis on Android apps to extract API calls, permissions and intent messages as the input features for k-means clustering and finally k-NN classification. DroidMiner [32] also extracted API calls, but then transformed them into modalities for associative classifier. Peiravian and Zhu [15] analyzed Android apps creating a feature set consisting of API calls and permission requests that they then fed to SVM, Decision Tree, and ensemble classifiers. Due to its high efficiency in feature extraction, in this paper, we choose to use static analysis for feature representation of Android apps. We first extract API calls from the smali files. Different from the existing works [15, 29, 32], we then further analyze the relationships between them (i.e., whether the extracted API calls belong to the same smali code block, are with the same package names, or use the same invoke method). Based on these extracted features, the Android apps will be represented by a structured heterogeneous information network (HIN), and a meta-path based approach will be used to link the apps.

Heterogeneous information network has been proposed for several years. HIN is a conceptual representation of graph/network with different types of entities and relations [11, 18]. It has been applied to scientific publication network analysis [17, 19, 20, 35], public general social media analysis [14, 33, 34], and document analysis based on knowledge graph [24–27]. Different from traditional graph similarities, such as shortest path, the similarity defined on HIN, i.e., PathSim [19], is more likely a natural extension to dot product. Different from the simple dot product, the similarity defined over HIN considers the semantics of the network metadata. Thus, the similarity can be related to certain topics or relationships. Originally, PathSim [19] is developed for ranking similar researchers in scientific publication data. Only one meta-path is used for each query. Then PathSim is extended by finding important paths for entity clustering by using some user provided seed entities in each entity type [20, 21]. In our application, the problem of Android malware detection is considered as a task of classification, thus we care more about the classification boundary instead of cluster centers to improve the generalization property. Another extension is to develop a similarity based on multiple meta-paths using an unsupervised meta-path weighting mechanism [25, 26]. This approach uses unsupervised feature selection algorithm to rank the meta-paths first, and then combines different meta-paths based on the selection criterion. Since it is a supervised learning task in our case, a better idea is to jointly optimize both the classification boundary and the meta-path weights based on the provided labels (either malicious or benign).

7 CONCLUSION

To combat the Android malware threats, in this paper, instead of using API calls only for feature representation, we further analyze the relationships among them, which create higher-level semantics and require more efforts for attackers to evade the detection. Based on the extracted features, we present a novel Android malware detection framework, *HinDroid*, which introduces a structured heterogeneous information network (HIN) representation of Android apps, and a meta-path based approach to link the apps. We use each meta-path to formulate a similarity measure over Android apps, and aggregate different similarities using multi-kernel learning. Then each meta-path is automatically weighted by the learning algorithm. To the best of our knowledge, this is the first work to use HIN representation for Android malware detection. A comprehensive experimental study on the real sample collections from Comodo Cloud Security Center is performed to compare various malware detection approaches. Promising experimental results demonstrate that *HinDroid* outperforms other alternative Android malware detection techniques as well as popular mobile security products. The system has already 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. National Science Foundation under grant CNS-1618629 and WVU Senate Grants for Research and Scholarship (R-16-043). The work of Y. Song is supported by China 973 Fundamental R&D Program (No.2014CB340304).

REFERENCES

[1] *APKTool.* http://ibotpeaches.github.io/Apktool/.
[2] *Dalvik Opcodes.* http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.
[3] *DEX.* http://www.openthefile.net/extension/dex.
[4] *IDC.* http://www.idc.com.
[5] *Ransomware.* https://containment.comodo.com/why-comodo/ransomware.php.
[6] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: Behavior-based Malware Detection System for Android. In *SPSM*.
[7] Marko Dimjasevic, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamaric. 2016. Evaluation of Android Malware Detection Based on System Calls. In *IWSPA*.
[8] Marko Dimjašević, Simone Atzeni, Ivo Ugrina, and Zvonimir Rakamarić. 2015. *Android Malware Detection Based on System Calls.* Technical Report.
[9] Adrienne Porter Felt, Matthew Finifter, Erika Chin, Steve Hanna, and David Wagner. 2011. A Survey of Mobile Malware in the Wild. In *SPSM*.
[10] Mehmet Gönen and Ethem Alpaydin. 2011. Multiple Kernel Learning Algorithms. *Journal of Machine Learning Research* 12 (2011), 2211–2268.
[11] Jiawei Han, Yizhou Sun, Xifeng Yan, and Philip S. Yu. 2010. Mining Knowledge from Databases: An Information Network Analysis Approach. In *SIGMOD*.
[12] Xiaofei He, Deng Cai, and Partha Niyogi. 2005. Laplacian Score for Feature Selection. In *Advances in Neural Information Processing Systems 18*.
[13] Shifu Hou, Aaron Saas, Yanfang Ye, and Lifei Chen. 2016. DroidDelver: An Android Malware Detection System Using Deep Belief Network Based on API Call Blocks. In *WAIM*. 54–66.
[14] Xiangnan Kong, Jiawei Zhang, and Philip S. Yu. 2013. Inferring anchor links across multiple heterogeneous social networks. In *CIKM*. 179–188.
[15] N. Peiravian and X. Zhu. 2013. Machine Learning for Android Malware Detection Using Permission and API Calls. In *IEEE ICTAI*. 300–305.
[16] Sören Sonnenburg, Gunnar Rätsch, Christin Schäfer, and Bernhard Schölkopf. 2006. Large Scale Multiple Kernel Learning. *JMLR* 7 (2006), 1531–1565.
[17] Yizhou Sun, Rick Barber, Manish Gupta, Charu C Aggarwal, and Jiawei Han. 2011. Co-author relationship prediction in heterogeneous bibliographic networks. In *ASONAM*. IEEE, 121–128.
[18] Yizhou Sun and Jiawei Han. 2012. Mining heterogeneous information networks: principles and methodologies. *Synthesis Lectures on Data Mining and Knowledge Discovery* 3, 2 (2012), 1–159.
[19] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S. Yu, and Tianyi Wu. 2011. PathSim: Meta Path-Based Top-K Similarity Search in Heterogeneous Information Networks. *PVLDB* (2011), 992–1003.
[20] Yizhou Sun, Brandon Norick, Jiawei Han, Xifeng Yan, Philip S. Yu, and Xiao Yu. 2012. Integrating meta-path selection with user-guided object clustering in heterogeneous information networks. In *KDD*. 1348–1356.
[21] Yizhou Sun, Brandon Norick, Jiawei Han, Xifeng Yan, Philip S. Yu, and Xiao Yu. 2013. PathSelClus: Integrating Meta-Path Selection with User-Guided Object Clustering in Heterogeneous Information Networks. *TKDD* 7, 3 (2013), 11.
[22] K. Tam, S. Khan, A. Fattori, and L. Cavallaro. 2015. CopperDroid: Automatic Reconstruction of Android Malware Behaviors.. In *NDSS*.
[23] S. V. N. Vishwanathan, Zhaonan sun, Nawanol Ampornpunt, and Manik Varma. 2010. Multiple Kernel Learning and the SMO Algorithm. In *NIPS*. 2361–2369.
[24] Chenguang Wang, Yangqiu Song, Ahmed El-Kishky, Dan Roth, Ming Zhang, and Jiawei Han. 2015. Incorporating World Knowledge to Document Clustering via Heterogeneous Information Networks. In *SIGKDD*. 1215–1224.
[25] Chenguang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. 2015. KnowSim: A Document Similarity Measure on Structured Heterogeneous Information Networks. In *ICDM*. 1015–1020.
[26] Chenguang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. 2016. Text Classification with Heterogeneous Information Network Kernels. In *AAAI*.
[27] Chenguang Wang, Yangqiu Song, Dan Roth, Chi Wang, Jiawei Han, Heng Ji, and Ming Zhang. 2015. Constrained Information-Theoretic Tripartite Graph Clustering to Identify Semantically Similar Relations. In *IJCAI*. 3882–3889.
[28] P. Wood. 2015. Internet Security Threat Report 2015. In *Symantec*.
[29] D. J. Wu, C. H. Mao, T. E. Wei, H. M. Lee, and K. P. Wu. 2012. DroidMat: Android Malware Detection through Manifest and API Calls Tracing. In *Asia JCIS*. 62–69.
[30] Wen-Chieh Wu and Shih-Hao Hung. 2014. DroidDolphin: A Dynamic Android Malware Detection Framework Using Big Data and Machine Learning. In *RACS*.
[31] J. Xu, Y. Yu, Z. Chen, B. Cao, W. Dong, Y. Guo, and J. Cao. 2013. MobSafe: cloud computing based forensic analysis for massive mobile applications using data mining. *Tsinghua Science and Technology* 18, 4 (August 2013), 418–427.
[32] Chao Yang, Zhaoyan Xu, Guofei Gu, Vinod Yegneswaran, and Phillip Porras. 2014. DroidMiner: Automated Mining and Characterization of Fine-grained Malicious Behaviors in Android Applications. In *19th European Symposium on Research in Computer Security*. 163–182.
[33] Jiawei Zhang, Xiangnan Kong, and Philip S. Yu. 2013. Predicting Social Links for New Users across Aligned Heterogeneous Social Networks. In *ICDM*. 1289–1294.
[34] Jiawei Zhang, Xiangnan Kong, and Philip S. Yu. 2014. Transferring heterogeneous links across location-based social networks. In *WSDM*. 303–312.
[35] Peixiang Zhao, Jiawei Han, and Yizhou Sun. 2009. P-Rank: a comprehensive structural similarity measure over information networks. In *CIKM*. 553–562.