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

Shifu Hou, Yanfang Ye ∗
Department of CSEE
West Virginia University, 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
- Artificial Intelligence → General; - Database applications → Data mining; - Security and Protection → Invasive Software;

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

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 smartphones, is currently dominating the smartphone 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 (malicious software) that deliberately fulfills the harmful intent to the smartphone 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
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., Dalvik 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: f836f5c6267f13bf9f6109ab8d79175) 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.

![Figure 1: Screen shots of the ransomware “Locker.apk”](image)

**Listing 1: An example of smali code**

```
1 method protected
2 readLibs(/android/content/Context)/Y
3 locals 4
4 try_start
5 new-instance v0, Ljava/io/BufferedReader;
6 new-instance v1, Ljava/io/InputStreamReader;
7 invoke-virtual ()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;)
11 move-result-object v2
12 invoke-virtual (v2) Ljava/lang/Process->getInputStream(Ljava/io/InputStream)
13
14 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 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.

- **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
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/Runtime; → 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 $R_1$, we generate the API-CodeBlock matrix $B$ where each element $b_{ij} = b_{ij} \in \{0, 1\}$ 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/HttpReques” 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 $R_2$, we generate the API-Package matrix $P$ where each element $p_{ij} = p_{ij} \in \{0, 1\}$ 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) invokevirtual: invokes a static method with parameters; (2) invokevirtual: invokes a virtual method with parameters; (3) invokespecial: invokes a method with parameters without the virtual method resolution; (4) invokesuper: invokes the virtual method of the immediate parent class; and (5) invokeinterface: 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 $R_3$, we generate
the API-InvokeMethod matrix $I$ where each element $I_{ij} = i_{ij} \in \{0, 1\}$ 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; \rightarrow \text{exec (Ljava/lang/String;)} Ljava/lang/Process"
and "Ljava/lang/Process; \rightarrow \text{getInputStream()} Ljava/io/InputSteam"oth 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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;

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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;

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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;

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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;

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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;

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
nodes as entity types from $A$ to use heterogeneous information network to represent the
Andriod entity set and different types of relations. Thus, in this section, we introduce how
apply machine learning algorithms, it is also be/t/ter to distinguish
given the analysis of rich relationship types of API calls for Andriod
tacklers add several non-associated API calls in the same code
evade the detection (e.g., it may result in execution collapse if
belong to the same code block, are with the same package name, or
both use $Ljava/lang/Process$;
provide us different similarities with different semantic meanings, 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 \( P_k, k = 1, \ldots, K \). We can compute the corresponding commuting matrices \( M_{P_k}, k = 1, \ldots, K \), where \( M_{P_k} \) 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:

\[
M = \sum_{k=1}^{K} \beta_k M_{P_k},
\]

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:

\[
\min_{w>0, \xi >0, \beta_k \geq 0} \frac{1}{2} \sum_{k} ||w_k||^2/\beta_k + C \sum_{i} \xi_i + \frac{\lambda}{2} \left( \sum_{k} \beta_k^p \right)^{2/p},
\]

s.t. \( y_i \left( \sum_{k} w_k^T \phi_k(x_i) + b \right) \geq 1 - \xi_i, \)

where for each kernel we learn a parameter vector \( w_k \). For each data \( \{x_i, y_i\} \), the slack parameter \( \xi_i \) 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 \( \phi_k(x_i)^T \phi_k(x_i) = M_{P_k}(i, j) \). Then by applying the representation theorem, we have \( w_k = \sum a_i \phi_k(x_i) \). \( a_i \) can be solved using the dual formulation, and non-zero \( a_i \)'s lead to the support vectors.

In multi-kernel learning framework, another set of parameters besides \( w_k \) is \( \beta_k \). Here the \( p \)-norm \( \left( \sum_{k} \beta_k^p \right)^{2/p} \) is used to regularize the optimization of \( \beta_k \)'s. Empirically we found 2-norm is the best, and apply it to our problem throughout the paper. After the optimization, the weights \( \beta_k \)'s are optimized to reveal the importance of the meta-paths serving as kernels. For a new app \( x \) coming, \( \sum_k w_k \phi_k(x) + b \) 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>
<thead>
<tr>
<th>Indices</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>TP</td>
<td># of apps correctly classified as malicious</td>
</tr>
<tr>
<td>TN</td>
<td># of apps correctly classified as benign</td>
</tr>
<tr>
<td>FP</td>
<td># of apps mistakenly classified as malicious</td>
</tr>
<tr>
<td>FN</td>
<td># of apps mistakenly classified as benign</td>
</tr>
<tr>
<td>Precision</td>
<td>( TP/(TP + FP) )</td>
</tr>
<tr>
<td>Recall</td>
<td>( TP/(TP + FN) )</td>
</tr>
<tr>
<td>ACC</td>
<td>( (TP + TN)/(TP + TN + FP + FN) )</td>
</tr>
<tr>
<td>F1</td>
<td>( 2 \times \text{Precision} \times \text{Recall} / (\text{Precision} + \text{Recall}) )</td>
</tr>
</tbody>
</table>

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: \( P_{121} \rightarrow P_{126} \rightarrow P_{125} \rightarrow P_{123} \rightarrow P_{121} \rightarrow P_{129} \rightarrow P_{122} \rightarrow P_{128} \rightarrow P_{127} \rightarrow P_{1213} \rightarrow P_{1214} \rightarrow P_{1215} \rightarrow P_{1210} \rightarrow P_{124} \rightarrow P_{121} \). We select the top five meta-paths (i.e., \( P_{1216}, P_{1216}, P_{126}, P_{123}, \) and \( P_{125} \)) and use their Laplacian scores as the weights to construct a new kernel (i.e., \( P_{1217} \)) 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., \( P_{1219} \)). Another set of comparison is using all the meta-paths (i.e., \( P_{1218} \)) with combined similarity and \( P_{1219} \) 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

<table>
<thead>
<tr>
<th>PID</th>
<th>Method</th>
<th>F1</th>
<th>$\beta$</th>
<th>ACC</th>
<th>TP</th>
<th>FP</th>
<th>TN</th>
<th>FN</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>AA$^T$</td>
<td>0.9529</td>
<td>0.1069</td>
<td>94.40%</td>
<td>283</td>
<td>19</td>
<td>189</td>
<td>19</td>
</tr>
<tr>
<td>2</td>
<td>ABA$^T$</td>
<td>0.9581</td>
<td>0.0900</td>
<td>95.00%</td>
<td>286</td>
<td>9</td>
<td>189</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>APA$^T$</td>
<td>0.9495</td>
<td>0.0858</td>
<td>94.20%</td>
<td>273</td>
<td>0</td>
<td>198</td>
<td>29</td>
</tr>
<tr>
<td>4</td>
<td>AIA$^T$</td>
<td>0.9183</td>
<td>0.0623</td>
<td>90.40%</td>
<td>270</td>
<td>16</td>
<td>182</td>
<td>32</td>
</tr>
<tr>
<td>5</td>
<td>ABPB$^T$A$^T$</td>
<td>0.9479</td>
<td>0.0670</td>
<td>94.00%</td>
<td>273</td>
<td>1</td>
<td>197</td>
<td>29</td>
</tr>
<tr>
<td>6</td>
<td>APBP$^T$A$^T$</td>
<td>0.9502</td>
<td>0.0565</td>
<td>94.20%</td>
<td>277</td>
<td>4</td>
<td>194</td>
<td>25</td>
</tr>
<tr>
<td>7</td>
<td>ABIB$^T$A$^T$</td>
<td>0.8683</td>
<td>0.0369</td>
<td>86.40%</td>
<td>254</td>
<td>29</td>
<td>169</td>
<td>48</td>
</tr>
<tr>
<td>8</td>
<td>AIBI$^T$A$^T$</td>
<td>0.8722</td>
<td>0.0437</td>
<td>84.60%</td>
<td>256</td>
<td>29</td>
<td>169</td>
<td>46</td>
</tr>
<tr>
<td>9</td>
<td>APBP$^T$A$^T$</td>
<td>0.8373</td>
<td>0.0445</td>
<td>81.20%</td>
<td>242</td>
<td>34</td>
<td>164</td>
<td>60</td>
</tr>
<tr>
<td>10</td>
<td>APIP$^T$A$^T$</td>
<td>0.8761</td>
<td>0.0572</td>
<td>86.60%</td>
<td>237</td>
<td>2</td>
<td>196</td>
<td>65</td>
</tr>
<tr>
<td>11</td>
<td>ABIP$^T$B$^T$A$^T$</td>
<td>0.9184</td>
<td>0.0616</td>
<td>90.80%</td>
<td>259</td>
<td>3</td>
<td>195</td>
<td>43</td>
</tr>
<tr>
<td>12</td>
<td>APBPB$^T$P$^T$A$^T$</td>
<td>0.8597</td>
<td>0.0617</td>
<td>84.60%</td>
<td>236</td>
<td>11</td>
<td>187</td>
<td>66</td>
</tr>
<tr>
<td>13</td>
<td>ABIB$^T$B$^T$A$^T$</td>
<td>0.9284</td>
<td>0.0426</td>
<td>91.80%</td>
<td>266</td>
<td>5</td>
<td>193</td>
<td>36</td>
</tr>
<tr>
<td>14</td>
<td>AIBP$^T$B$^T$A$^T$</td>
<td>0.8237</td>
<td>0.0426</td>
<td>82.60%</td>
<td>218</td>
<td>3</td>
<td>195</td>
<td>44</td>
</tr>
<tr>
<td>15</td>
<td>APIP$^T$B$^T$A$^T$</td>
<td>0.8597</td>
<td>0.0458</td>
<td>84.60%</td>
<td>236</td>
<td>11</td>
<td>187</td>
<td>66</td>
</tr>
<tr>
<td>16</td>
<td>Combined-kernel (5)</td>
<td>0.9214</td>
<td></td>
<td>91.20%</td>
<td>258</td>
<td>0</td>
<td>198</td>
<td>44</td>
</tr>
<tr>
<td>17</td>
<td>Combined-kernel (16)</td>
<td>0.9740</td>
<td></td>
<td>96.80%</td>
<td>300</td>
<td>14</td>
<td>184</td>
<td>2</td>
</tr>
<tr>
<td>18</td>
<td>Multi-kernel (5)</td>
<td>0.9834</td>
<td></td>
<td>98.00%</td>
<td>297</td>
<td>5</td>
<td>193</td>
<td>5</td>
</tr>
<tr>
<td>19</td>
<td>Multi-kernel (16)</td>
<td>0.9884</td>
<td></td>
<td>98.60%</td>
<td>299</td>
<td>4</td>
<td>194</td>
<td>3</td>
</tr>
</tbody>
</table>

Some other meta-paths do not perform well on their own, such as $\text{ABPB}^T\text{B}^T\text{A}^T$, 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 $\text{PID}_12$, $\text{PID}_16$, $\text{PID}_6$, $\text{PID}_3$, and $\text{PID}_5$, the meta-paths for $\text{PID}_12$, $\text{PID}_3$, and $\text{PID}_5$ 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 $\beta_k$ 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 $\beta_k$’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.
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), Naïve 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 $1.62 \times 10^{-13}$. 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 Android app is around 3-5 seconds on average, including the feature extraction.

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 (January 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.
The 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 Program (No.2014CB340304).

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
[11] Jiawei Han, Yizhou Sun, Xifeng Yan, and Philip S. Yu. 2010. Mining Knowledge from Databases: An Information Network Analysis Approach. In SIGMOD.
[17] Yizhou Sun, Rick Barber, Manish Gupta, Chara 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. PathSelChau: Integrating Meta-Path Selection with User-Guided Object Clustering in Heterogeneous Information Networks. TKDD 7, 3 (2013), 11.
[24] Chengwang 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 SIGOD 2015. 1228–1224.
[25] Chengwang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. 2015. Knowledge Similarity Search on Heterogeneous Information Networks. In IJCAI. 1015–1020.
[26] Chengwang Wang, Yangqiu Song, Haoran Li, Ming Zhang, and Jiawei Han. 2016. Text Classification with Heterogeneous Information Network Kernels. In AAAI.
[27] Chengwang 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 Relationships. In IJCAI. 3882–3889.
[30] Chengwang 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 SIGOD 2015. 1228–1224.
[35] Peixiang Zhao, Jiawei Han, and Yizhou Sun. 2009. P-Rank: a comprehensive structural similarity measure over information networks. In CIKM. 553–562.