David Marcus Thierry
DSC180A - Winter 2020
A13873297 - Malware

HinDroid Replication: Malware Classification Final Report

The problem being investigated…
Mobile operating system security has become a very important field of privacy and security due to the explosive growth of the Android operating system and its respective applications. As a result of this operating system gaining popularity recently, there are many more apps that an individual can choose from. A consequence of this growth, there is now more incentive for people to publish malicious applications because these malicious applications now have a higher chance of getting discovered and downloadeded. The severity of the malicious app may differ from app to app, and may differ in implementation, but each app is Malware because it does something other than what the client believes it should do. Malicious apps disrupt the everyday lives of smartphone users, thus the detection of Android application malware has become increasingly relevant. The HinDroid paper provides a solution for detecting Android malware. Traditional approaches of classifying malware consist of analyzing Application Programming Interface (API) calls, however, in this case we can further analyze the different relationships between them to create higher-level semantics which require more efforts for attackers to evade the detection. We will be reconstructing the Android Application Malware Detection System based on Structured Heterogeneous Information Network (HinDroid Paper).

Data being used to approach the problem…
Unlike traditional desktop based Portable Executable files, Android apps are 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 fille format which contains compiled code written for Android, but is uninterpretable by default. In order to analyze the Android apps, we need to convert the dex file to a readable format. APKTool is an assembler/disassembler for the dex format, which provides us readable code in smali language. Smali code is the readable interpreted code between Java and Dex for Android Applications.
The problem at hand is as follows, given a dataset of both benign and malicious apps, classify these apps correctly as either being a Malware App or a Benign App.

Benign Android Application Data
We will be obtaining our benign training data by scraping android applications from www.APKPure.com. Apkpure and similar free download websites simply download the apps from the Play Store and then upload the APKs on their websites. This data source is fairly reliable for scraping benign data because APKPure takes certain measures to ensure the quality of the apps they host.

“ How do we make sure all Apps are authentic and created by the respective developers? ”

- All APKPure.com apps are verified prior to publishing.
- We make sure that the cryptographic signatures for new versions of all previously published apps match the original ones, which means we know if the new version APKs were signed by the real devs or someone pretending to be them.
- For new apps that have never been published on APKPure.com, we try to match the signatures to other existing apps by the same developer. If there’s a match, it means that the same key was used to sign a previously known legitimate app, therefore validating the new upload. If we’re unable to verify the legitimacy of a new APK, we will simply not publish it on APKPure.com.

- Source: www.apkpure.com/faq-safe.html

Malware Android Application Data
The ability to identify and combat Malware requires understanding how Malware works. Such knowledge is divorced from how it gets used by the practitioner; As the practitioner, it is imperative that I understand the ethical concerns with the usage of this knowledge and behave accordingly. We will be obtaining our malicious training data from our professor and domain leader, Prof. Aaron Fraenkel. He will distribute this malicious training data via University of California San Diego data science machine learning platform (DSMLP Server). The malware data is contained on UCSD DSMLP servers and requires root access to obtain the data. The Malware dataset is only provided under very secured circumstances because of the range of potential consequences when dealing with raw malicious data.

The HinDroid paper's dataset consisted of a real sample collection of Android apps privately collected and provided by Comodo Mobile Security. It is safe to assume that the dataset which HinDroid trained it's models on was a (50/50) sample of benign and malicious apps. For this replication, the data ingestion approach consists of benign Android application scraping from www.APKpure.com as well as malware provided to us through secure school servers. By following these methods of data ingestion stated, I believe that the dataset I am collecting will be somewhat representative of the dataset used by the authors of HinDroid. Given the representative sample of Android Applications, this data is appropriate to address the problem at hand because the classifier will be trained in the same exact way.

Data Privacy / Concerns

In order to replicate the results of the HinDroid paper, and since we do not have access to the Android Application Database provided by Comodo Mobile Security, we will obtain our dataset of Android Applications using the previously mentioned methods. It is important to remember that most applications (any platform) are not open source and cannot be redistributed without the developers permission. Each app may have different Terms of Use and End User License Agreements and legality should be evaluated on an app by app basis.

Data Generation Process

Benign Data Ingestion Process

- **Step 1:** Using www.apkpure.com/sitemap.xml, download all xml files. Each xml file contains the download link of each app documented by the xml file instance.
- **Step 2:** The xml files are categorized, thus we initialize a sitemap dictionary which consists of:
    - Key --> Category
    - Value --> Compressed XML Links
- **Step 3:** Obtain a sample of app category xmls (50 Categories, N apps from each category)
    - Which apps and how many apps can be specified in config/data-params.json.
- **Step 4:** Obtain a list of app download links from step 3
- **Step 5:** Download each apk file
- **Step 6**: Temporarily save each apk file in data directory
- **Step 7:** Decompile each downloaded app to smali
- **Step 8:** Remove the recently downloaded apk file.
    - No reason to save .apk files; Saving .apk files both wastes memory and throttles runtime.
- **Step 9:** Save the Smali folder containing the removed .apk file code in /data directory.
- **Step 10:** Benign App Ingestion Complete!

Malware Data Ingestion Process

*** Remember that all Malware data is stored locally on dsmlp-servers ***

- **Step 1:** Initialize malware directory filepath.
- **Step 2:** Search entire directory for smali files, populate a malware dictionary:
    - Key --> Malware App Directory File Path
    - Value --> Malware App Smali File Paths
- **Step 3:** Sample the same amount of Malware Apps as Benign Apps from malware_dictionary
- **Step 4:** Malware App Ingestion complete!

Graph Data Generation

These are the graphs that will be used to understand statistics about the Android applications described in the HinDroid Paper. We will be using 4 different graphs to represent each app. Each graph is explained below and is given the inputs required in order to calculate

- The first graph (A) will be a graph where the nodes will be APIs.
    - :param: all unique apis found in Malware and Benign smali files.
    - :param: all apis found per app.
- The edges for the second graph (B) will be pairs of APIs that occur in the same code block (method).
    - :param: Need all unique apis found in Malware and Benign smali files.
    - :param: Need all apis found within the same code block per app.
- The edges for the third graph (P) will be pairs of APIs that belong to the same package.
    - :param: Need all unique apis found in Malware and Benign smali files.
    - :param: Need all apis found in the same package per app.
- And finally, the edges for the last graph (I) will be pairs of APIs that are called with the same invocation in the app.
    - Upon further inspection, we do not need to calculate the I Matrix for this assignment

Text processing techniques were utilized in order to obtain the nodes and edges smali file data for all of the above graphs. Because of the amount of possible files that can potentially be used for creating each graph, I found that using a dictionary where keys are api calls and values are a unique identifier from 0, … , n is a much more efficient container for indexing the api calls as opposed to a list because it allows us to find the index of an api call in approximately O(n) time. We need to be able to find the index of an API call so that we can update each respective graph as needed. These graphs/matrices can be used for exploratory data analysis and for training the model, thus this is the end of the data generated from ingested smali files.

Exploratory Data Analysis
This HinDroid Replication Classifier will be run on a dataset of 1000 apps. 500 apps are downloaded (from www.APKPure.com) and are labeled as benign. The other 500 apps come from navigating through DSMLP servers to obtain 500 malware apps and their respective smali files. Each application has a number of smali files associated with them, ranging from 0 to n. For each smali file found inside of an app, and for each individual app itself, statistics were calculated on the features extracted from the respective smali files.

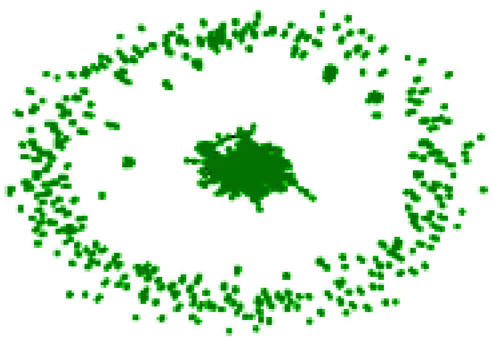
An example of these statistics that were calculated are shown below. The features extracted include the following:

- Num_Methods : The total number of methods found inside of an app
- Num_Files: The number of smali files found inside of the app
- Reflections: The number of reflections found inside an app
- Unique_apis: The number of unique apis found inside an app
- Unique_packages: The number of unique packages found inside of an app
- API_call_count: The total number of apies

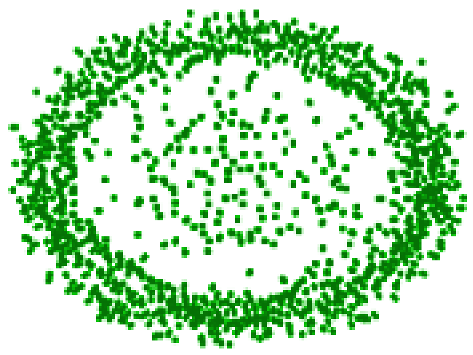
Type	num_methods	num_files	Reflections	Unique APIs	Unique Packages	API Call Count
Mal	3486	5936	148	10202	4312	15021
Ben	5366	8124	669	13870	3071	27018
Ben	4721	6352	431	12122	3021	19321

Based on these extracted features, we can learn alot about what types of apps have certain kinds of patterns. By doing exploratory data analysis, we can discover more about our data which will allow us to make smarter implementation decisions.
Visualizing Graphs

B Graph



P Graph



B Graph - The B matrix is based on which api calls are contained in the same code blocks, and can be interpreted as follows: The center of the B graph represents the api which are used together in many apps often whereas the outer ring represents apis in code blocks that occur together in the same context. This implies that there might be a type of hierarchy with api calls, where some apis are considered to be core apis, and others apis not found in the middle acting supplemental apis.

P Graph - The P graph is much more dispersed and decentralized than that of the B graph. The P graph captures whether or not two apis are contained in the same package. A package is defined as a namespace that organizes a set of related classes and interfaces. Conceptually you can think of packages as being similar to different folders on your computer. Because packages are usually imported to deal with a certain task, it makes sense that there is less overlap seen in P than A.

Benign App Reliability - The data (apps) downloaded from www.APKPure.com are reliable in the sense that they mostly contain smali files. There are some instances of apps that do not contain any smali files after being decompiled. This is plausible in practice, it would make sense that an api with no smali files is representative of an app not being malicious.

Malicious App Reliability - The malware that has been provided to us is reliable because it has been provided to us by our class instructor specifically for this HinDroid Replication. Thus, all the malicious smali files that we obtained through our data ingestion will be following the same schema as the benign apps.

Cleaning and Formatting
There was no need for much cleaning or formatting of the parsed data that was ingested. Each Smali Converted App (Benign or Malicious) that comes from the android app decompiler (apktool) has a consistent output file schema. By using apktool, we can utilize the fact that apktool has a predetermined output, so I adapted a helper function specifically for obtaining statistics on smali code that works on any smali file..

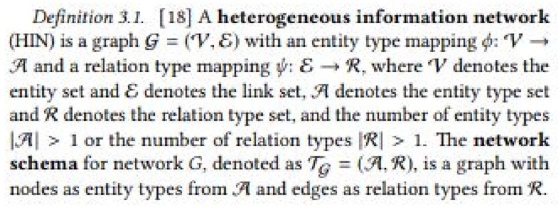
HinDroid's Approach to Classifying Malware
Matrix Generation

The graphs that you saw before were to be utilized to understand underlying statistics about the Android applications from the HinDroid Paper. Luckily, creating a graph is essentially the same as creating a matrix. As a result, we can conveniently compute the following matrix interpretations from the graphs we previously had. We will be using 4 different matrices to capture relationships found in each app. Each matrix is explained below along with the required inputs to initialize the specified matrix.

- The first graph (A) will be a graph where the nodes will be APIs.
    - :param: all unique apis found in Malware and Benign smali files.
    - :param: all apis found per app.
- The edges for the second graph (B) will be pairs of APIs that occur in the same code block (method).
    - :param: Need all unique apis found in Malware and Benign smali files.
    - :param: Need all apis found within the same code block per app.
- The edges for the third graph (P) will be pairs of APIs that belong to the same package.
    - :param: Need all unique apis found in Malware and Benign smali files.
    - :param: Need all apis found in the same package per app.

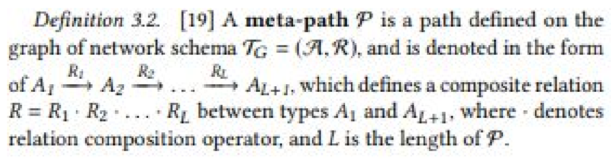
Given the analysis of rich relationship types of API calls for Android apps (Matrices A, B, P), it is important to model them in a proper way so that different relations can be better and easier handled. When we apply machine learning algorithms, it is also better to distinguish different types of relations.

Thus, a heterogeneous information network is used to represent the Android apps by using the features (including API calls and the relationships among them) extracted above. Heterogeneous Information Networks not only provide 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.



Definition 3.1. [18] A **heterogeneous information network** (HIN) is a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with an entity type mapping $\phi: \mathcal{V} \to \mathcal{A}$ and a relation type mapping $\psi: \mathcal{E} \to \mathcal{R}$, 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 $\mathcal{T}_{\mathcal{G}} = (\mathcal{A}, \mathcal{R})$, is a graph with nodes as entity types from $\mathcal{A}$ and edges as relation types from $\mathcal{R}$.

There are four types of relations, e.g., apps containing API calls, 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-paths are used in the concept of HIN to formulate the semantics of higher-order relationships among entities.



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 meta path based approach can be used to characterize the semantic relatedness of apps and APIs. API calls are used by the Android apps in order to access operating system functionality and system resources; they can be used as representations of the behaviors of an Android app. Roughly speaking, we can interpret a meta-path as such; We want to find a way to connect two apps (nodes) in a HIN through the path containing the same API over the HIN. 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. Each different meta path has its own interpretation as well as the specific relationships it wishes to capture. Finally, we can utilize the kernel trick to make our lives easier.

The final tasks (for general results replication) in HinDroid's approach are to fit a Support Vector Classifier with the metapath kernel and to then pass in testing data to that classifier to obtain our results. The graphs that make up HIN are appropriate for approaching the problem shown above because unlike traditional malware classification techniques where API calls were only used for feature representation, a HIN graph allows us further analyze the relationships among them, by creating higher level semantics and requiring more efforts for attackers to evade the detection.

Results of the HinDroid Classification Replication

The following table is the results of this replicated HinDroid project trained on a set of 375 malware and benign apps and tested on a set of 375 separate malware and benign apps. The total number of apps used was 750. The metapath kernels that were used were the following: [AA^T, ABA^T, APA^T, APBP^TA^T`].

Kernel	Accuracy	Sensitivity	Specificity	Precision	TP%	FP%	TN%	FN%	TP	FP	TN	FN
AA^T	.924	.936	.908	.936	.549	.037	.37	.037	206	14	139	16
ABA^T	.949	.951	.953	.968	.572	.018	.378	.032	215	7	141	11
APA^T	.938	.944	.993	.995	.546	.002	.396	.058	204	1	148	12
$ABPB^TA^T$	.931	.936	.924	.944	.546	.032	.394	.058	203	12	146	14

Conclusion

Based on the extracted features, this is a replication project of the 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 (not shown in this replication; only single kernel). This work is based on *HinDroid: An Intelligent Android Malware Detection System Based on Structured Heterogeneous Information Network* by contributors: Shifu Hou, Yangfang Ye, Yangqiu Song, and Melih Abdulhayogul.