DENDROID: A Text Mining Approach to Analyzing and Classifying Code Structures in Android Malware Families

Guillermo Suarez-Tangil[a,*], Juan E. Tapiador[a], Pedro Peris-Lopez[a], Jorge Blasco Alis[a]

[a] *Computer Security (COSEC) Lab*
Department of Computer Science, Universidad Carlos III de Madrid
28911 Leganes, Madrid, Spain.

Abstract

The rapid proliferation of smartphones over the last few years has come hand in hand with and impressive growth in the number and sophistication of malicious apps targetting smartphone users. The availability of reuse-oriented development methodologies and automated malware production tools makes exceedingly easy to produce new specimens. As a result, market operators and malware analysts are increasingly overwhelmed by the amount of newly discovered samples that must be analyzed. This situation has stimulated research in intelligent instruments to automate parts of the malware analysis process. In this paper, we introduce DENDROID, a system based on text mining and information retrieval techniques for this task. Our approach is motivated by a statistical analysis of the code structures found in a dataset of ANDROID OS malware families, which reveals some parallelisms with classical problems in those domains. We then adapt the standard Vector Space Model and reformulate the modelling process followed in text mining applications. This enables us to measure similarity between malware samples, which is then used to automatically classify them into families. We also investigate the application of hierarchical clustering over the feature vectors obtained for each malware family. The resulting dendograms resemble the so-called phylogenetic trees for biological species, allowing us to conjecture about evolutionary relationships among families. Our experimental results suggest that the approach is remarkably accurate and deals efficiently with large databases of malware instances.

Keywords: Malware analysis, software similarity and classification, text mining, information retrieval, smartphones, Android OS

*Corresponding author
Email addresses: guillermo.suarez.tangil@uc3m.es (Guillermo Suarez-Tangil), jestevez@inf.uc3m.es (Juan E. Tapiador), pperis@inf.uc3m.es (Pedro Peris-Lopez), jbalis@inf.uc3m.es (Jorge Blasco Alis)

1. Introduction

The past few years have witnessed a rapid proliferation of smartphones as popular portable devices with increasingly powerful computing, networking and sensing capabilities. In their current generation, most existing smartphones are far more powerful than early personal computers (PCs). But perhaps the key feature of these devices is that they offer the possibility to easily incorporate third-party applications ("apps", for short) through online markets. The popularity of smartphones has been repeatedly corroborated by recent commercial surveys, showing that they will very soon outsell the number of PCs worldwide [1]. For example, the number of smartphone users has swiftly increased over the past few years. In 2011, global mobile handset shipments reached 1.6 billion units [2], and the total smartphone sales reached 472 million units (58% percent of all mobile devices sales in 2010) [3]. According to a report by Nielsen [4], the number of ANDROID OS and iOS users alone increased from 38 to 84 million between 2011 and 2012. Specifically, the global mobile operating system market share shows that ANDROID OS reached 69.7% at the end of 2012, racing past other platforms such as Symbian, BlackberryOS and iOS. The same report also indicates that the average number of apps per device increased from 32 to 41, and the proportion of time spent by users on smartphone applications almost equals the time spent on the Web (73% vs. 81%). Furthermore, the number of worldwide smartphone sales saw a record of 207.7 million units during 2012, rising up 38.3% with respect to the same period in the previous year [5].

In many respects, smartphones present greater security and privacy issues to users than traditional PCs [6]. For instance, many of such devices incorporate numerous sensors that could leak highly sensitive information about users location, gestures, moves and other physical activities, as well as recording audio, pictures and video from their surroundings. Furthermore, users are increasingly embedding authentication credentials into their devices, as well as making use of on-platform micropayment technologies such as NFC [7].

One major source of security and privacy problems for smartphone users is precisely the ability to incorporate third-party applications from available online markets. Many market operators carry out a revision process over submitted apps, which presumably also involves some form of security testing to detect if the app includes malicious code. So far such revisions have proven clearly insufficient for several reasons. First, market operators do no give details about how (security) revisions are done. However, the ceaseless presence of malware in official markets reveals that operators cannot afford to perform an exhaustive analysis over each submitted app. Second, determining which applications are malicious and which are not is still a formidable challenge. This is further complicated by a recent rise in the so-called grayware [8], namely apps that are not fully malicious but that entail security and/or privacy risks of which the user is not aware. And finally, a significant fraction of users rely on alternative markets to get access for free to apps that cost money in official markets. Such unofficial and/or illegal markets have repeatedly proven to be fertile ground for malware, particularly in the form of popular apps modified (*repackaged*) to

include malicious code.

1.1. Motivation

The reality is that the rapid development of smartphone technologies and its widespread user acceptance have come hand in hand with a similar increase in the number and sophistication of malicious software targeting popular platforms. Malware developed for early mobile devices (e.g., Palm platforms) and featured mobile phones was identified prior to 2004. The proliferation of mobile devices in the subsequent years translated into an exponential growth in the presence of malware specifically developed for them (mostly Symbian), with more than 400 cases between 2004 and 2007 [9, 10]. Later on that year, iOS and Android OS were released and shortly became the predominant platforms. This gave rise to an alarming escalation in the number and sophistication of malicious software targetting these platforms, particularly Android OS. For example, according to the mobile threat report published by Juniper Networks in 2012, the number of unique malware variants for Android OS increased by 3325.5% during 2011 [2]. A similar report by F-Secure reveals that the number of malicious Android OS applications received during the first quarter of 2012 increased from 139 to 3063 when compared to the first quarter of 2011 [11], and by the end of 2012 it already represents 97% of the total mobile malware according to McAfee [12].

The main factors driving the development of malware have swiftly changed from research, amusement and the search for notoriety to purely economical –and political, to a lesser extent. Current malware industry already generates substantial revenues [13], and emergent paradigms such as Malware-as-a-Service (MAAS) paint a gloomy forecast for the years to come. In the case of smartphones, malware is a profitable industry due to (i) the existence of a high number of potential targets and/or high value targets; and (ii) the availability of reuse-oriented development methodologies for malware that make exceedingly easy to produce new specimens. Both points are true for the case of Android OS and explain, together with the open nature of this platform and some technical particularities, why it has become such an attractive target to attackers.

Malware analysis is a thriving research area with a substantial amount of still unsolved problems (see, e.g., [14] for an excellent survey). In the case of smartphones, the impressive growth both in malware and benign apps is making increasingly unaffordable any human-driven analysis of potentially dangerous apps. This state of affairs have consolidated the need for intelligent analysis techniques to aid malware analysts in their daily functions. For instance, when confronted with a continuously growing stream of incoming malware samples, it would be extremely helpful to differentitate between those that are minor variants of a known specimen and those that correspond to novel, previously unseen samples. Grouping samples into families, establishing the relationships among them, and studying the evolution of the various known "species" is also a much sought after application.

1.2. Overview and Contributions

Problems similar to those discussed above have been successfuly attacked with Artificial Intelligence and Data Mining techniques in many application domains. In this paper, we explore the use of text mining approaches to automatically analyze smartphone malware samples and families based on the code structures present in their software components. Such code structures are representations of the Control Flow Graph (CFG) of each method found in the app classes [15, 16]. A high level overview of DENDROID's main building blocks and salient applications is provided in Fig. 1. During the modeling phase, all different code structures are extracted from a dataset of provided malware samples. A vector space model is then used to associate a unique feature vector with each malware sample and family. This vector representation is then used two illustrate two main applications:

- Automatic classification of unknown malware samples into candidate families based on the similarity of their respective code structures. Our classification scheme involves a preparatory stage where the sample is transformed into a query in the text mining sense. Thus, a slight variation of this process can be used to search for a set of given code structures in a database of known specimens, a task that could be remarkably useful for malware analysts and app market operators.

- We show how it is possible to perform an evolutionary analysis of malware families based on the dendograms obtained after hierarchical clustering. The process is almost equivalent to the analysis of the so-called phylogenetic trees for biological species [17], although using software code structures rather than physical and/or genetic features. This enables us to conjecture about evolutionary relationships among the various malware families, including the identification of common ancestors and studying the diversification process that they may have gone through as a consequence of code reuse and malware re-engineering techniques.

DENDROID is novel in two separate ways. On the one hand, to the best of our knowledge using code structures to characterize ANDROID OS malware families has not been explored before. One major advantage of focussing on the internal structure of code units (methods) rather than on their specific sequence of instructions is an improved resistance against obfuscation (i.e., deliberate modifications of the code aimed at evading pattern-based recognition [18]). Furthermore, such structures prove to be particularly useful for the case of smartphone malware, where rapid development methodologies heavily based on code reuse are prevalent. On the other hand, the idea of using text mining techniques to automate tasks such as classifying specimens, searching for code components, or studying evolutionary relationships of malware families is, to our knowledge, novel too. Besides, text mining techniques were developed to efficiently deal with massive amounts of data, a feature which turns out to be very convenient for the problems that we address here.

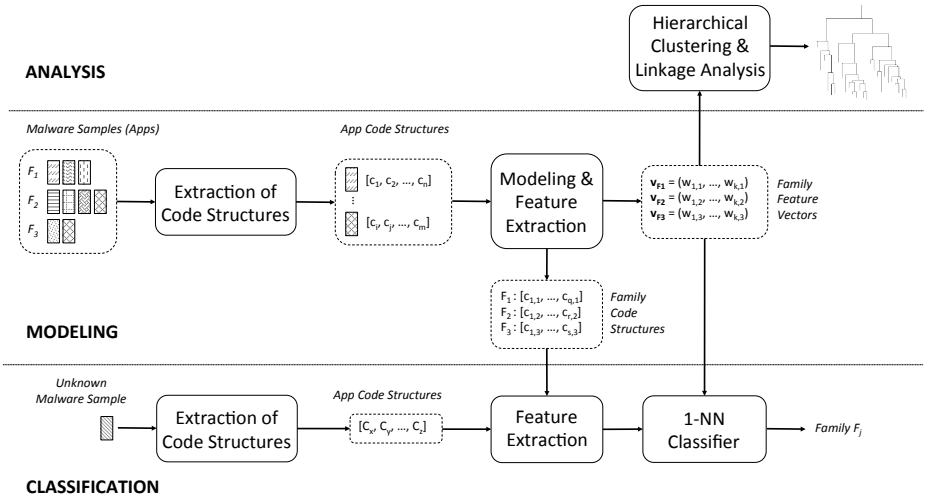


Figure 1: Overview of DENDROID's architecture.

The remaining of this paper is organized as follows. In Section 2 we describe the dataset of Android malware families used in this paper, together with the tools and methodology followed to extract code structures from each app. In Section 3 we analyze and discuss various statistical features of the code structures found in the malware instances. Based on our findings from this analysis, in Section 4 we propose DENDROID, a text mining approach to classify and analyze malware families according to the code structures present in their apps. We first introduce a suitable vector space model and report experimental results related to classifying instances into families, measuring similarity among families, and using dendrograms to analyze the evolutionary relationships among families. In Section 5 we provide an overview of related work in this area. Finally, Section 6 concludes the paper and discusses our main contributions and future research directions.

2. Dataset and Experimental Setting

The work presented in this paper is largely based on a sizeable dataset of real-world ANDROID OS malware samples. The dataset, known as the *Android Malware Genome Project*[1] was collected, characterized and discussed by Zhou and Jian in [19], and covers the majority of malware families for ANDROID OS up to late 2011. It consists of 1247 malicious apps grouped into 49 different families that include a variety of infection techniques (repackaging, update at-

[1] Available at `http://www.malgenomeproject.org`

tacks, and drive-by-download) and payload functionalities (privilege escalation, remote control, financial charge, and private information exfiltration). For the purposes of this paper, we discarded 16 out of the 49 families as they only contain one specimen each, resulting in a final dataset of 1231 malware samples grouped into 33 families. More details on this will be later provided in Section 3.

2.1. Extracting Code Structures

One key aspect of our work is the decomposition of an app into a number of constituent code elements referred to as *code chunks*. Each code chunk corresponds to a method associated with a class within the app. Thus, an app will be fragmented into as many code chunks as methods contained in it. Rather than focusing on the specific sequence of instructions contained in a code chunk, we extract a high-level representation of the associated Control Flow Graph (CFG). CFGs use graphs as a representation of the paths that a program might traverse during its execution. Each node in a CFG represents a "basic block", i.e., a piece of code that will be sequentially executed without any jumps. The CFG of a piece of code is explicit in the source code, is relatively easy to extract, and has been extensively used in static analysis techniques [20].

Each malware instance contained in the dataset described above has been first disassembled into Dalvik instructions. We then used Androguard [21] to extract the code chunks of all malicious apps and compute their structure. Androguard is an open source tool that implements a number of static analysis functions over ANDROID OS apps. CFGs provided by Androguard are based on a grammar proposed by Cesare and Xiang [15] and shown in Fig. 2. The sequence of instructions contained in a code chunk is thus replaced by a list of statements defining its control flow, such as a block of consecutive instructions (B), and bifurcation determined by an "if" condition (I), an unconditional go-to jump (G), and so on. After parsing each code chunk with this grammar, the resulting structure is a sequence of symbols of varying length such as those shown in Fig. 2.

After this process, each malware sample a is represented by a sequence:

$$a = \langle c_1, c_2, \ldots, c_{|a|} \rangle \tag{1}$$

where c_i is a string describing the code structure of the i-th method in a, and $|a|$ is the total number of methods contained in a. In the remaining of this paper, we will refer to c_i's indistinctly as code chunks or code structures. The resulting dataset of code chunks, grouped by app and family as in the original Android Malware Genome Project, has been made publicly available[2].

[2] `http://www.seg.inf.uc3m.es/~guillermo-suarez-tangil/dendroid/codechunks.zip`

Grammar:

```
    Procedure   ::=  StatementList
StatementList   ::=  Statement | Statement StatementList
    Statement   ::=  BasicBlock | Return | Goto | If | Field | Package | String
       Return   ::=  'R'
         Goto   ::=  'G'
           If   ::=  'I'
   BasicBlock   ::=  'B'
        Field   ::=  'F'0 | 'F'1
      Package   ::=  'P' PackageNew | 'P' PackageCall
   PackageNew   ::=  '0'
  PackageCall   ::=  '1'
  PackageName   ::=  Epsilon | Id
       String   ::=  'S' Number | 'S' Id
       Number   ::=  \d+
           Id   ::=  [a-zA-Z]\w+
```

Examples:

CC_1 `B[P0P1]B[I]B[P1R]B[P1P1I]B[P0SP1P1P1]B[P1G]|B[F1P1R]`

CC_2 `B[SSF1F0P1SF0SP1P1I]B[SP1P1F1SP1F1F0I]B[F0P1I]B[F0SP1]B[]`
`B[P1SP1SP1F1SF0P1I]B[F0I]B[F0P1I]B[F1F0P1P1I]B[F0P1I]B[]B[F0P1]`
`B[F0I]B[S]B[P1I]B[F0P1]B[I]B[P1F0P1P1F0P1I]B[F0P1P1I]B[F0P1I]`
`B[]B[F0P1F0P1]B[P0F0P1P1SP1F0P1SP1F0P1SP1F0P1P1F0P1F0P1S]`

CC_3 `B[P1SF1R]`

Figure 2: CFG grammar used by Androguard to extract code structures.

3. Analysis of Code Structures in Android Malware Families

In this section, we analyze and discuss various statistical features of the code structures found in the malware apps and families of the dataset described above. Our findings will subsequently motivate the use of text-mining techniques for tasks such as, for example, the classification of new apps into candidate malware families or the analysis of similarities among families.

3.1. Definitions

We are interested in exploring questions such as how large, in terms of number of code chunks (CCs), apps are; what the distribution of CCs across apps and families is; or how discriminant a subset of CCs is for a given family. We next introduce a number of measures that will be later used to perform this analysis.

Definition 1 (CC)**.** *We denote by* $\texttt{CC}(a)$ *the set of all different CCs found in app* a*. We emphasize that* $\texttt{CC}(a)$ *is a set and, therefore, it does not contain repeated elements.*

Definition 2 (Redundancy)**.** *The redundancy, $R(a)$, of an app a is given by:*

$$R(a) = 1 - \frac{|\texttt{CC}(a)|}{|a|} \tag{2}$$

where $|a|$ is the total number of CCs (possibly with repetitions) in a.

Note that redundancy measures the fraction of repeated CCs present in an app, with low values indicating that CCs do not generally appear multiple times in the app, and vice versa.

Definition 3 (FCC)**.** *The set of family CCs for a family $\mathcal{F}_i$ is given by:*

$$\texttt{FCC}(\mathcal{F}_i) = \bigcup_{a \in \mathcal{F}_i} \texttt{CC}(a) \tag{3}$$

Definition 4 (CCC)**.** *The set of common CCs for a family $\mathcal{F}_i$ is given by:*

$$\texttt{CCC}(\mathcal{F}_i) = \bigcap_{a \in \mathcal{F}_i} \texttt{CC}(a) \tag{4}$$

In short, the set $\texttt{CCC}(\mathcal{F}_i)$ contains those CCs found in all apps of $\mathcal{F}_i$. Even though this can be certainly seen as a distinctive feature of family $\mathcal{F}_i$, it does not imply that all those CCs are unique to $\mathcal{F}_i$. For instance, code reuse –which is a recurrent feature of malware in general and, particularly, of smarphone malware– will make the same CCs appear in multiple families.

Definition 5 (FDCC)**.** *Given a set of malware families $\mathcal{M} = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$, a set $C = \{c_1, \ldots, c_n\}$ of CCs is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$ iff:*

(i) $C \subseteq \texttt{CCC}(\mathcal{F}_i)$*, and*

(ii) $\forall \mathcal{F}_k \in \mathcal{M}, \mathcal{F}_k \neq \mathcal{F}_i : C \cap \texttt{FCC}(\mathcal{F}_k) = \emptyset$

We denote by $\texttt{FDCC}(\mathcal{F}_i|\mathcal{M})$ the maximal set of fully discriminant CCs for $\mathcal{F}_i$ with respect to $\mathcal{M}$; that is, $C = \texttt{FDCC}(\mathcal{F}_i|\mathcal{M})$ iff C is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$, and for all C' such that C' is fully discriminant for $\mathcal{F}_i$ with respect to $\mathcal{M}$, $C' \subseteq C$.

Put simply, a set of CCs is fully discriminant for a family $\mathcal{F}_i$ if and only if every CC in the set appears in every app of $\mathcal{F}_i$ and, furthermore, no CC in the set appear in any app of any other family. Consequently, such a set unequivocally identifies the family, provided that it is not the empty set.

3.2. Results and Discussion

We computed the various measures and sets described above over all the apps and families in our dataset. Table 1 and Figure 3 summarize the most relevant results.

The entire dataset contains 84854 different CCs. In terms of number of unique CCs, apps do not display a uniform behavior, neither within the same

		App stats		Family stats		
Family $\mathcal{F}_i$	$\lvert\mathcal{F}_i\rvert$	$Avg\{\lvert\texttt{CC}(a)\rvert\}$	$Avg\{R(a)\}$	$\lvert\texttt{FCC}(\mathcal{F}_i)\rvert$	$\lvert\texttt{CCC}(\mathcal{F}_i)\rvert$	$\lvert\texttt{FDCC}(\mathcal{F}_i\lvert\mathcal{M})\rvert$
ADRD	22	416	0.59	2726	21	8
AnserverBot	187	367	0.64	17635	44	9
Asroot	8	78	0.57	462	1	0
BaseBridge	122	433	0.53	9918	5	0
BeanBot	8	746	0.68	3081	61	34
Bgserv	9	384	0.53	487	67	34
CruseWin	2	82	0.53	82	82	40
DroidDream	16	302	0.51	2545	10	0
DroidDreamLight	46	529	0.54	3339	40	13
DroidKungFu1	34	501	0.58	7609	10	0
DroidKungFu2	30	295	0.51	2418	9	0
DroidKungFu3	309	872	0.58	19092	48	11
DroidKungFu4	96	936	0.56	9239	19	2
DroidKungFuSapp	3	351	0.66	411	310	0
FakePlayer	6	6	0.73	7	10	2
GPSSMSSpy	6	13	0.44	23	9	3
Geinimi	69	430	0.58	12141	77	37
GingerMaster	4	223	0.64	297	159	108
GoldDream	47	513	0.54	9129	13	3
Gone60	9	35	0.41	56	26	5
HippoSMS	4	148	0.67	262	8	1
KMin	52	502	0.50	795	120	42
NickySpy	2	65	0.71	84	47	34
Pjapps	45	1160	0.58	15128	6	0
Plankton	11	133	0.52	876	14	2
RogueLemon	2	962	0.54	1441	483	321
RogueSPPush	9	365	0.60	633	114	60
SndApps	10	28	0.55	54	20	11
Tapsnake	2	33	0.57	55	12	2
YZHC	22	316	0.48	1704	33	11
Zsone	12	365	0.40	535	338	1
jSMSHider	16	113	0.46	266	64	52
zHash	11	1348	0.56	2344	645	390

Table 1: Statistical indicators obtained for all apps and families in the dataset.

family nor across families. Apps in some malware families have, on average, only a few different CCs: see for example `FakePlayer` (6), `GPSSMSSpy` (13), or `SndApps` (28). In contrast, others are quite large, such as for example `zHash` (1348), `Pjapps` (1160), or `DroidKungFu4` (936).

The variance, both of apps' length and redundancy within each family, is generally large, as illustrated by the boxplots shown in Figures 3(a) and 3(b). This can be explained by a number of factors, including the fact that in many cases malware belonging to the same family appears in very different apps, each one with its own set and distribution of CCs. In general, however, all apps display a redundancy between 0.4 and 0.7 regardless of their size.

The sizes of the `FCC`, `CCC`, and `FDCC` sets for each family reveal some remarkable details. The number of family CCs (`FCC`) varies quite significantly across families. Furthermore, such variability seems uncorrelated with the aver-

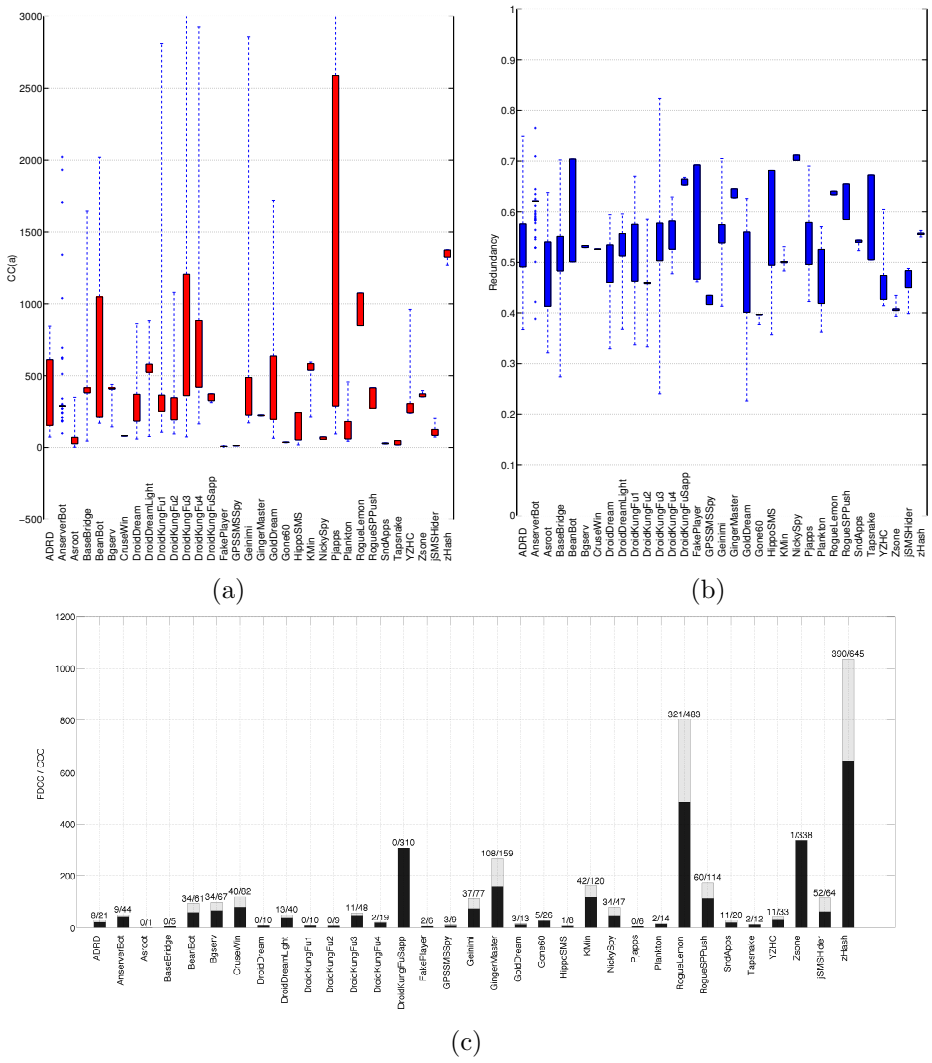


Figure 3: Distribution of (a) unique CCs (`CC`); (b) redundancy (R); and (c) common and fully discriminant CCs for each family (`CCC`/`FDCC`).

age number of CCs in the apps. The most likely explanation for this has to do with the proliferation and prevalence of each malware family. Families such as `AnserverBot`, `Geinimi`, `Pjapps`, and `DroidKungFu` appeared in a variety of very popular repackaged apps and infected a significant number of devices. Thus, finding the same malware in very different apps induces a sharp increase in the size of `FCC`.

The `CCC` set removes this diversity and identifies code structures common

to all available apps within a family. The size of this set varies across families, being quite low in families where the malware code has undergone significant evolution, possibly after being included in different apps. For example, only 6 CCs appear in each of the 45 samples of `Pjapps`. On the contrary, apps in unpopular or rare families share essentially the same version of the malware: see for example `zHash`, where all its 11 apps share 645 CCs.

Finally, the rightmost column in Table 1 shows the number of fully discriminant CCs for each family. Surprisingly, The `FDCC` set is non-empty for 26 out of the 33 families. This suggest that, in principle, those CCs might be used as a "signature" to perfectly classify an app into one of those families. We believe, however, that such a scheme would be extremely weak for a number of reasons. One of the most important shortcomings of using `FDCC` as the basis to represent malware family features is that it is very fragile: the addition of a new app to a family such that it does not share any CCs with those already in the family automatically makes the `CCC` set empty, which in turn makes `FDCC` empty too. Such an app might have actually been incorrectly labeled as belonging to the family, or perhaps carefully constructed to avoid sharing CCs with all other apps. In either case, the characterization of the family would not be useful anymore.

We next study the distribution of CCs across families, which will motivate a more robust representation of family features.

3.3. Distribution of Code Structures

Fig. 4 shows the distribution of CCs as a function of the number of families where they appear. This plot is obtained by iterating over all different code structures and computing, for each one of them, the number of different families where they appear. (A CC appear in a family if it appears in at least one app of that family.) The results reveal that 78.9% of all code structures appear in just one family. Note that this does not mean that such a family is the same, as different code structures may appear in different families. Rather, this value indicates that if a code structure is found in one family, it is unlikely to find that same code structure in an app belonging to a different family. Similarly, the number of code structures that appear in 2, 3, 4, and 5 different families drops to 12.6%, 3.5%, 1.5% and 1.1%, respectively. Consequently, less than 1% of all available code structures appear in 6 or more different families.

This distribution of code structures across malware families suggests that each family can be sufficiently well characterized by just a few code structures, possibly accompanied by some extra information such as the frequency of that code structure in each app of the family, the fraction of apps where it appears, etc. We next elaborate on this.

4. DENDROID: Mining Code Chunks in Malware Families

Based on the findings discussed in the previous section, we next describe DENDROID, our approach to analyzing malware samples and families based on

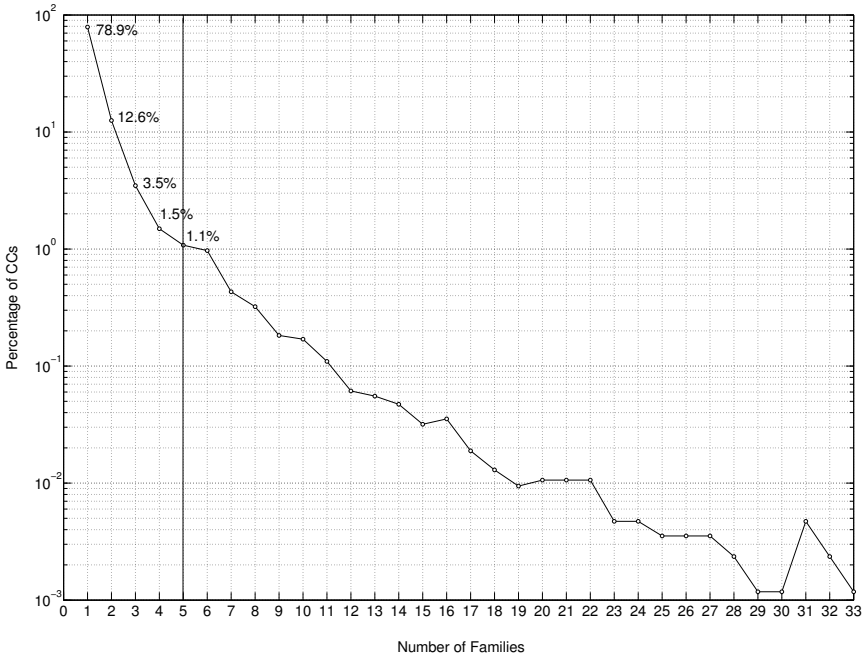


Figure 4: Distribution of CCs as a function of the number of families where they appear.

mining code structures. We first present the vector space model used and describe the main features of our prototype implementation. Subsequently we present two main applications –classifying unknown malware apps and analyzing possible evolutinary paths of malware families– and discuss the experimental results obtained.

4.1. Vector Space Model

In this section, we adapt to our problem various numerical indicators well researched in the field of information retrieval and text mining. One central concept in those fields is the so-called Vector Space Model (VSM) [22], sometimes known as Term Vector Model, where each object d_j of a corpus is represented as a vector of identifiers

$$\mathbf{d_j} = (w_{1,j}, \ldots, w_{k,j}) \tag{5}$$

Each identifier $w_{i,j}$ is a measure of the relevance that the i-th term, t_i, has in object d_j. In the most common setting, objects and terms are documents and words, respectively. Thus, $w_{i,j}$ is an indicator of the importance of word t_i in document d_j.

Many interesting problems related to information retrieval and text mining can be easily reformulated in the VSM in terms of vector operations. For example, the cosine of the angle between two vectors is a good measure of the similarity between the associated documents. Such vector operations are the basis for a number of interesting primitives, such as comparing two documents or ranking various documents according to their similarity to a given query (after appropriately representing queries as vectors too).

One popular statistical indicator used in the VSM is the term frequency-inverse document frequency (tf-idf). Using the notation introduced above, the tf-idf $w_{i,j}$ of term t_i in d_j is the product of two statistics: (1) the term frequency (tf), which measures the number of times t_i appears in d_j; and (2) the inverse document frequency (idf), which measures whether t_i is common or rare across all documents in the corpus. Thus, a high tf-idf value means not only that the corresponding term appears quite often in a document, but also that it is not frequent in other documents. As a result, one important effect is that the tf-idf tends to filter out terms that are common across documents.

Our proposal essentially mimics the model discussed above. Each family $\mathcal{F}_j$ is represented by a vector $\mathbf{v_j} = (I_{1,j}, \ldots, I_{k,j})$, where $I_{i,j} = \mathtt{I}(c_i, \mathcal{F}_j, \mathcal{M})$ is computed as

$$\mathtt{I}(c_i, \mathcal{F}_j, \mathcal{M}) = \mathtt{ccf}(c_i, \mathcal{F}_j) \cdot \mathtt{iff}(c_i, \mathcal{M}) \tag{6}$$

The indicators $\mathtt{ccf}(c, \mathcal{F}_j)$ and $\mathtt{iff}(c, \mathcal{M})$ are approximately equivalent to the tf and idf statistics, respectively, and can be computed as follows.

Definition 6 (CCF)**.** *The frequency of a CC c in a family $\mathcal{F}_j$ is given by*

$$\mathtt{ccf}(c, \mathcal{F}_j) = \frac{\sum_{a \in \mathcal{F}_j} \mathtt{freq}(c, a)}{\max\{\mathtt{freq}(c, a) \ : \ a \in \mathcal{F}_j\}} \tag{7}$$

where $\mathtt{freq}(c, a)$ is the number of occurrences of CC c in app a.

Definition 7 (IFF)**.** *The inverse family frequency of a CC c with respect to a set of malware families $\mathcal{M} = \{\mathcal{F}_1, \ldots, \mathcal{F}_m\}$ is given by*

$$\mathtt{iff}(c, \mathcal{M}) = \log \frac{|\mathcal{M}|}{1 + |\{\mathcal{F}_i \in \mathcal{M} \ : \ c \in \mathtt{FCC}(\mathcal{F}_j)\}|} \tag{8}$$

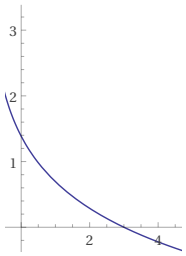
4.2. An example

We next illustrate the model presented above with a numerical example and discuss some relevant features. Assume two different datasets, $\mathcal{M}_1$ and $\mathcal{M}_2$, of malicious apps, with $|\mathcal{M}_1| = 4$ and $|\mathcal{M}_2| = 400$. Given a CC c_i, we can easily see how each family feature vector varies according to the relevance of c_i.

On the one hand, when c_i is a rather common CC (see Fig. 5a), i.e., it appears in most families, the `iff` value quickly vanishes (see Fig. 5b). Similarly, it can also be observed how the components of a family vector grow when the frequency of a CC increases, as shown in 5a. On the other hand, when c_i is a very uncommon CC, the `iff` value grows significantly: see, e.g., Fig. 5 where

	$\mathcal{F}_1$			$\mathcal{F}_2$		$\mathcal{F}_3$	$\mathcal{F}_4$
Apps	a_1	a_2	a_3	a_4	a_5	a_6	a_7
Is c_i in a_k?	✓	×	✓	✓	×	×	×
$\texttt{ccf}(c_i, \mathcal{F}_j)$	2/3			1/2		0	1
$\texttt{iff}(c_i, \mathcal{M}_1)$	$\log \frac{4}{1+2} = 0.288$						
$I(c_i, \mathcal{F}_j, \mathcal{M}_1)$	0.192			0.144		0.000	0.288

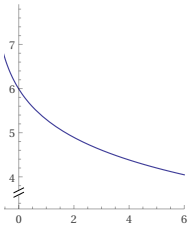
(a) Rather common CC with $|\mathcal{M}_1| = 4$.



(b) $\texttt{iff}(c_i, \mathcal{M}_1)$

	$\mathcal{F}_1$			$\mathcal{F}_2$		$\cdots$	$\mathcal{F}_{400}$
Apps	a_1	a_2	a_3	a_4	a_5	$\cdots$	a_n
Is c_i in a_k?	✓	×	×	✓	×	×	✓
$\texttt{ccf}(c_i, \mathcal{F}_j)$	1/3			1/2		0	1
$\texttt{iff}(c_i, \mathcal{M}_2)$	$\log \frac{400}{1+3} = 4.605$						
$I(c_i, \mathcal{F}_j, \mathcal{M}_2)$	1.535			2.302		0.000	4.605

(c) Very uncommon CC with $|\mathcal{M}_2| = 400$



(d) $\texttt{iff}(c_i, \mathcal{M}_2)$

Figure 5: Computation of $I(c_i, \mathcal{F}_j, \mathcal{M})$ and distribution of the `iff` value depending on the popularity of the CC in two different malware datasets: *tiny* (a) and (b), and *large* (c) and (d).

$\texttt{iff}(c_i, M_2)$ is 16 times larger than $\texttt{iff}(c_i, M_1)$. The overall result is that the relevance of a CC is strongly influenced by its frequency across families. Thus, CCs that are common to many families have a low influence in the family feature vector, even if they are very frequent.

4.3. Implementation

We have built a Java implementation of the VSM discussed above and applied it over all families in our dataset to obtain a family feature vector for each of them. The process is described by the algorithm shown in Fig. 6 and outputs one vector $\mathbf{v_j}$ for each malware family $\mathcal{F}_j$, with each vector component representing the relevance of a CC in F_j.

The algorithm comprises three main steps: (i) initialization, (ii) inverse family frequency computation, and (iii) CC frequency computation. First, we extract the frequency $\texttt{freq}(\texttt{c}, \texttt{a})$ for every CC $c \in \texttt{CC}(a)$ of each app $a \in \mathcal{M}$ (lines 2–5). The inverse family frequency is then computed for each extracted CC using Eq. (8) (lines 8–10). Finally, the frequency of each CC is computed by applying Eq. (8), and the associated indicator for the CC is obtained (lines 11–16).

4.4. Modelling Families and Classifying Malware Instances

In our first experiment, we have tested the ability to correctly predict the family of a malware instance. To do this, we have split our dataset in two subsets of approximately equal number of malware instances and the same family

Algorithm 1. Computing Family Vectors

Input:

Dataset of labelled malware apps (sequences of code chunks):

$$\mathcal{M} = \{(a_1, \mathcal{F}_{a_1}), (a_2, \mathcal{F}_{a_2}), \ldots, ((a_p, \mathcal{F}_{a_m}))\}$$

where $\mathcal{F}_{a_i} \in \{\mathcal{F}_1, \ldots, \mathcal{F}_q\}$

Output:

Vectors $\mathbf{v_j} = (I_{1,j}, \ldots, I_{k,j})$ for each $\mathcal{F}_j \in \{\mathcal{F}_1, \ldots, \mathcal{F}_q\}$

Algorithm:

1 $\texttt{FCC}(\mathcal{F}_j) = \emptyset \ \forall j = 1, \ldots, q$
2 **For each** $(a, \mathcal{F}_a) \in \mathcal{M}$ **do**
3 $\quad \texttt{FCC}(\mathcal{F}_a) = \texttt{FCC}(\mathcal{F}_a) \ \cup \ \texttt{CC}(a)$
4 $\quad$ Update $\texttt{freq}(c, a)$ for each $c \in \texttt{CC}(a)$
5 **end-for**
6 $\texttt{C}(\mathcal{M}) = \bigcup_{j=1}^{q} \texttt{FCC}(\mathcal{F}_j)$
7 $k = |\texttt{C}(\mathcal{M})|$
8 **For each** $i = 1, \ldots, k$ **do**
9 $\quad$ Compute $\texttt{iff}(c_i, \mathcal{M})$ according to (8)
10 **end-for**
11 **For each** $\mathcal{F}_j$ **do**
12 $\quad$ **For each** $i = 1, \ldots, k$ **do**
13 $\quad\quad$ Compute $\texttt{ccf}(c_i, \mathcal{F}_j)$ according to (7)
14 $\quad\quad$ $\mathbf{v_j}[i] = I_{i,j} = \texttt{ccf}(c_i, \mathcal{F}_j) \cdot \texttt{iff}(c_i, \mathcal{M})$
15 $\quad$ **end-for**
16 **end-for**
17 **return** $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$

Figure 6: Algorithm for obtaining each family vector.

distribution. This has been simply carried out by randomly picking from each family half of the malware instances (or the closest integer number when the family had an odd number of members). The process resulted in two datasets with 621 and 610 malware instances, respectively.

The first dataset (621 instances) was used to derive a vectorial representation for each malware family as described in Section 4.1. A total number of 84854 CC were found across all instances in the dataset, so each family is represented by a vector with this dimensionality, as specified in (6). We note, however, that such vectors are very sparse (as expected by the analysis given in Section 3), which in practice makes very efficient to store and manipulate them. For illustration purposes, the largest family vectors correspond to `DroidKungFu3` (19091 non-null components), `AnserverBot` (17634), `Pjapps` (15127), and `Geinimi` (12140). On average, only around 11% of each feature vector contains discriminant information.

The second dataset (610 instances) was processed in a similar way, obtaining a vectorial representation for each malware instance. We then implemented a

Algorithm 2. 1-NN malware classifier

Input:
Family vectors $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$ and data structures $\langle \mathtt{C}(\mathcal{M}), \mathtt{iff}(c_i, \mathcal{M})\rangle$
Malware instance a
Output:
Predicted family $\mathcal{F}_j$
Algorithm:
1 **for each** $c_i \in \mathtt{C}(\mathcal{M})$ **do**
3 $\quad \mathbf{u}[i]) = \mathtt{freq}(c_i, a) \cdot \mathtt{iff}(c_i, \mathcal{M})$
4 **end-for**
5 $j = \arg\min_i\{\mathrm{dist}(\mathbf{u}, \mathbf{v_i})\}$
6 **return** $\mathcal{F}_j$

Figure 7: 1-NN malware classification algorithm.

1-NN (nearest neighbor) classifier [23] to compute the predicted family for each malware instance under test. Such a prediction is the family whose vector is closest to the instance's vector (see Fig. 7). 1-NN is a widely used method in data mining that only requires to compute n distances and one minimum. To compute distances between vectors, we relied on the well-known cosine similarity:

Definition 8 (Cosine similarity)**.** *The cosine similarity between two vectors* $\mathbf{u} = (u_1, \ldots, u_k)$ *and* $\mathbf{v} = (v_1, \ldots, v_k)$ *is given by*

$$\mathrm{sim}(\mathbf{u}, \mathbf{v}) = \cos(\theta_{\mathbf{u},\mathbf{v}}) = \frac{\mathbf{u} \cdot \mathbf{v}}{\parallel \mathbf{u} \parallel \parallel \mathbf{v} \parallel} = \frac{\sum_{i=1}^{k} u_i v_i}{\sqrt{\sum_{i=1}^{k} u_i^2}\sqrt{\sum_{i=1}^{k} v_i^2}} \tag{9}$$

The cosine similarity, which measures the cosine of the angle between vectors $\mathbf{u}$ and $\mathbf{v}$, has been extensively used to compare documents in text mining and information retrieval applications. Besides, it is quite efficient to evaluate in domains such as ours, since vectors are sparse and, therefore, only a few non-zero dimensions need to be considered in the computation. As for our purposes a distance, and not a similarity, is required, we use:

$$\mathrm{dist}(\mathbf{u}, \mathbf{v}) = 1 - \mathrm{sim}(\mathbf{u}, \mathbf{v}) \tag{10}$$

The overall classification error per family attained in this experiment is shown in Table 2. In overall terms, 35 out of the 610 malware instances are misclassified, resulting in a global classification error around 5.74%. A closest inspection reveals that the classification error is not uniform across families. On the contrary, errors concentrate on 6 out of the 33 malware families studied

Classification Error: Incorrectly classified / total instances (%)			
ADRD	0/11 (00.00%)	GingerMaster	0/2 (00.00%)
AnserverBot	4/93 (04.30%)	GoldDream	0/23 (00.00%)
Asroot	0/4 (00.00%)	Gone60	0/4 (00.00%)
BaseBridge	5/61 (08.20%)	HippoSMS	0/2 (00.00%)
BeanBot	0/4 (00.00%)	KMin	0/26 (00.00%)
Bgserv	0/4 (00.00%)	NickySpy	0/1 (00.00%)
CruseWin	0/1 (00.00%)	Pjapps	0/22 (00.00%)
DroidDream	0/8 (00.00%)	Plankton	0/5 (00.00%)
DroidDreamLight	0/23 (00.00%)	RogueLemon	0/1 (00.00%)
DroidKungFu1	2/17 (11.76%)	RogueSPPush	0/4 (00.00%)
DroidKungFu2	3/15 (20.00%)	SndApps	0/5 (00.00%)
DroidKungFu3	13/154 (08.44%)	Tapsnake	0/1 (00.00%)
DroidKungFu4	8/48 (16.67%)	YZHC	0/11 (00.00%)
DroidKungFuSapp	0/1 (00.00%)	Zsone	0/6 (00.00%)
FakePlayer	0/3 (00.00%)	jSMSHider	0/8 (00.00%)
GPSSMSSpy	0/3 (00.00%)	zHash	0/5 (00.00%)
Geinimi	0/34 (00.00%)	**Global**	**35/610 (5.74%)**

Table 2: Malware classification error per family using 1-NN.

(`AnserverBot`, `BaseBridge`, and `DroidKungFu1` through `DroidKungFu4`), while instances belonging to the remaining 27 families are perfectly classified.

Interestingly, `DroidKungFu` has been considered a milestone in Android OS malware sophistication [19]. After the release of its first version, a number of variants rapidly emerged, including `DroidKungFu2` through `DroidKungFu4` or `DroidKungFuApp`. A common feature shared by all these variants is the use of encryption to hide their existence. In fact, some of them embedded their payloads within constant strings or even resource files (e.g., pictures, asset files, etc.). Furthermore, `DroidKungFu` aggressively obfuscates the class name and uses native programs (Java Native Interface, or JNI) precisely to difficult the analysis. Similarly, `AnserverBot` use sophisticated techniques to obfuscate all internal classes, methods, and fields. Moreover, instead of enclosing the payload within the app, `AnserverBot` dynamically fetches and loads it at runtime (this is known as *update attacks*). In this regard, some authors (e.g., [19]) believe that `AnserverBot` actually evolved from `BaseBridge` and inherited this feature from it. Our results seem to confirm this hypothesis.

More insights can be gained by observing the confusion matrix given in Table 3. Each cell (x, y) in the matrix shows the number of instances belonging to family x whose predicted family is y. Here, for instance, we can observe that 5 out of the 61 samples of `BaseBridge` have been predicted as `AnserverBot`. Similarly, we can observe that a few samples of `DroidKungFu1` have been classified as `DroidKungFu2` and, in a similar way, there is some missclassifications between `DroidKungFu3` and `DroidKungFu4`. Thus, the aforementioned classification error

Actual \ Predicted	ADRD	AnserverBot	Asroot	BaseBridge	BeanBot	Bgserv	CruseWin	DroidDream	DroidDreamLight	DroidKungFu1	DroidKungFu2	DroidKungFu3	DroidKungFu4	DroidKungFuSapp	FakePlayer	GPSSMSSpy	Geinimi	GingerMaster	GoldDream	Gone60	HippoSMS	KMin	NickySpy	Pjapps	Plankton	RogueLemon	RogueSPPush	SndApps	Tapsnake	YZHC	Zsone	jSMSHider	zHash	
ADRD	11	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**11**
AnserverBot	0	89	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**93**
Asroot	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
BaseBridge	0	5	0	56	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**61**
BeanBot	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
Bgserv	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
CruseWin	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
DroidDream	0	0	0	0	0	0	0	8	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**8**
DroidDreamLight	0	0	0	0	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**23**
DroidKungFu1	0	0	0	0	0	0	0	0	0	15	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**17**
DroidKungFu2	0	0	0	0	0	0	0	0	0	3	12	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**15**
DroidKungFu3	0	0	0	0	0	0	0	0	0	0	0	141	13	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**154**
DroidKungFu4	0	0	0	0	0	0	0	0	0	0	0	8	40	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**48**
DroidKungFuSapp	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**1**
FakePlayer	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
GPSSMSSpy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	3	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**3**
Geinimi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	34	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**34**
GingerMaster	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**2**
GoldDream	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	23	0	0	0	0	0	0	0	0	0	0	0	0	0	0	**23**
Gone60	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	0	0	0	0	0	0	0	**4**
HippoSMS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	0	**2**
KMin	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	26	0	0	0	0	0	0	0	0	0	0	0	**26**
NickySpy	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	**1**
Pjapps	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	22	0	0	0	0	0	0	0	0	0	**22**
Plankton	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	0	0	0	**5**
RogueLemon	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0	0	**1**
RogueSPPush	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	**4**
SndApps	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	0	0	0	0	0	**5**
Tapsnake	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	**1**
YZHC	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	11	0	0	0	**11**
Zsone	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	6	0	0	**6**
jSMSHider	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	8	0	**8**
zHash	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	5	**5**
	11	**94**	**4**	**60**	**4**	**4**	**1**	**8**	**23**	**18**	**14**	**149**	**53**	**1**	**3**	**3**	**34**	**2**	**23**	**4**	**2**	**26**	**1**	**22**	**5**	**1**	**4**	**5**	**1**	**11**	**6**	**8**	**5**	**610**

Table 3: Confusion matrix for malicious app classification.

is actually justified by the evolutionary relationships of these particular malware strands.

4.5. Evolutionary Analysis of Malware Families

In this section, we discuss the application of hierarchical clustering to the feature vectors that model samples and family. The resulting dendrograms are then used to conjecture about their evolutionary phylogenesis, giving a valuable instrument to discover relationships among families. We first describe the hierarchical clustering algorithm currently included in DENDROID. Subsequently we discuss the results obtained with our dataset.

4.5.1. Single Linkage Hierarchical Clustering

Single Linkage Clustering, also known as nearest neighbour clustering, is a well-known method to carry out an agglomerative hierarchical clustering process over a population of vectors. The algorithm, shown in Fig. 8, keeps a set of

Algorithm 3. Single-linkage hierarchical clustering of malware families

Input:

Family vectors $\{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$

Output:

Proximity matrices $D^{(t)} = [d_{ij}]$ and linkages at each level $L(k)$

Algorithm:

1 $\mathcal{K} = \{\mathbf{v_1}, \ldots, \mathbf{v_q}\}$
2 $D^{(0)} = [d_{ij}] = \text{dist}(\mathbf{v_i}, \mathbf{v_j})$ for all $\mathbf{v_i}, \mathbf{v_j} \in \mathcal{K}$
3 $m = 0, L(m) = 0$
4 **while** $|\mathcal{K}| \neq 1$ **do**
5 Find $\mathbf{r}, \mathbf{s} \in \mathcal{K}$ such that $\text{dist}(\mathbf{r}, \mathbf{s}) = \min_{\mathbf{a},\mathbf{b} \in \mathcal{K}} \{\text{dist}(\mathbf{a}, \mathbf{b})\}$
6 Merge $\mathbf{r}, \mathbf{s}$ into new cluster $\mathbf{v_{rs}}$
7 $m = m + 1$
8 $L(m) = \text{dist}(\mathbf{r}, \mathbf{s})$
9 $D^{(m)} = D^{(m-1)}$ deleting the rows and columns corresponding to $\mathbf{r}$ and $\mathbf{s}$
10 Add to D a new row and column for $\mathbf{v_{rs}}$
11 $D[\mathbf{v_{rs}}, \mathbf{x}] = \text{dist}(\mathbf{v_{rs}}, \mathbf{x}) = \min\{\text{dist}(\mathbf{r}, \mathbf{x}), \text{dist}(\mathbf{s}, \mathbf{x})\}$ for all $\mathbf{x}$
12 $\mathcal{K} = \mathcal{K} \cup \{\mathbf{v_{rs}}\} \setminus \{\mathbf{r}, \mathbf{s}\}$
13 **end-while**
14 **return** $\langle D^{(0)}, \ldots, D^{(m)}, L \rangle$

Figure 8: Single linkage hierarchical clustering algorithm for malware families.

clusters, $\mathcal{K}$, which is initialized to the set of family vectors. At each iteration m, the two closest clusters $\mathbf{r}, \mathbf{s} \in \mathcal{K}$ are combined into a larger cluster $\mathbf{v_{rs}}$. The distance matrix between each pair of clusters is then updated by removing both $\mathbf{r}$ and $\mathbf{s}$, adding the newly created $\mathbf{v_{rs}}$, and finally computing the distances from $\mathbf{v_{rs}}$ to each remaining cluster $\mathbf{x}$ through a linkage function. In our case, such a function is simply the shortest between the distance from $\mathbf{x}$ to $\mathbf{r}$ and the distance $\mathbf{x}$ to $\mathbf{s}$. Furthermore, the algorithm keeps a list $L(m)$ with the distances at which each fusion takes place. The process is iterated until the set of clusters $\mathcal{K}$ is reduced to one element.

4.5.2. Results and Discussion

The results of a hierarchical clustering can be visualized in a dendrogram as the one depicted in Fig. 10 for the dataset used in this work. The dendrogram represents a tree diagram where links between the leaves (malware families) illustrate the parental relationships (ancestors and descendants) in a hierarchy. Thus, clusters (denoted as v_{rs} in Fig. 8–line 6) are tree nodes representing merged families, i.e., a common ancestor. The paths that group together different families illustrate the phylogenetic evolution of the "species." Furthermore, the distance $\mathcal{D}^{(t)}$ between an ancestor and its descendants is a measure of their similarity and, therefore, can be interpreted as an evolutionary (or diversifica-

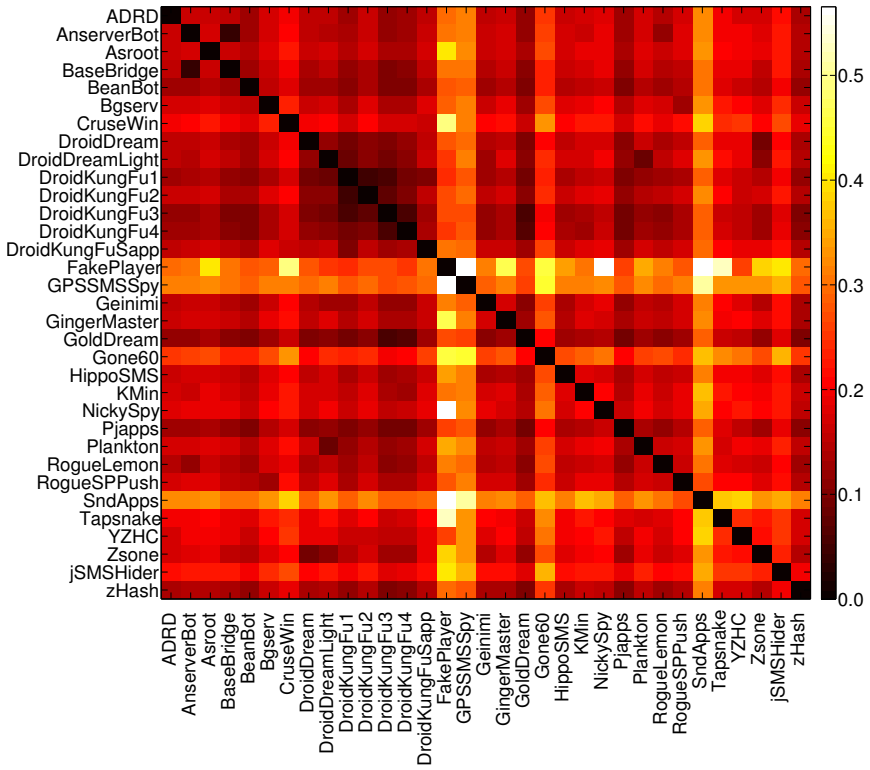


Figure 9: Distance matrix between pairs of malware families.

tion) distance. Note that the sequence of such distances is provided as an output by the algorithm in Fig. 8.

The initial proximity matrix, $\mathcal{D}^{(0)}$, for all the families in our dataset is graphically shown in Fig. 9. As anticipated by the results of the previous experiment, the similarity among some groups of families is striking, while in other cases there are substantial differences. The results after applying hierarchical clustering to the datasets are displayed in the dendrogram shown in Fig. 10. There are a number of interesting observations:

- `BaseBridge` and `AnserverBot` are intimately related, hence that they appear as variants of a common ancestor. Besides, their linkage (distance) is very small compared to the rest of the families, which suggest a large share of relevant code structures and, perhaps of functionality too.

- The case of the `DroidKungFu` variants is remarkably captured. It transpires from our results that `DroidKungFu1` and `DroidKungFu2` are alike, and the same occurs with the pair `DroidKungFu3` and `DroidKungFu4`. Furthemore, both pairs descend from a common ancestor, say `DroidKungFuX`,

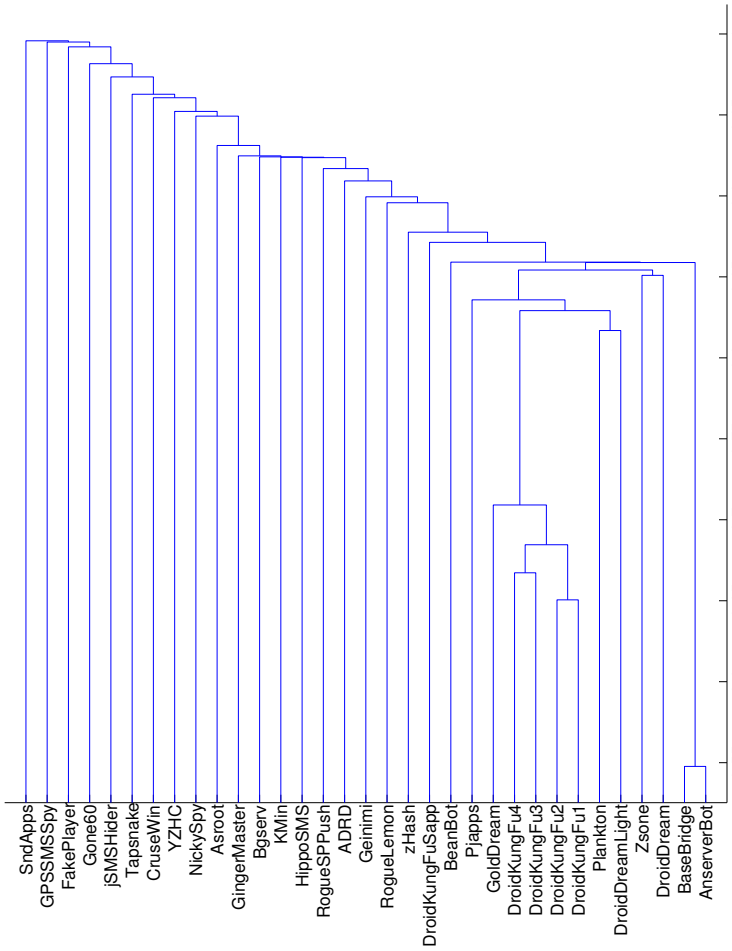


Figure 10: Dendrogram obtained after hierarchical clustering over the dataset.

which in turn is connected with `GoldDream`. This branch connects with another one formed by the pair `Plankton-DroidDreamLight`, and both groups relate to `Pjapps`, which is among the oldest examples of sophisticated ANDROID OS malware. Finally, the relationship between this group, `Zsone-DroidDream`, and `BaseBridge-AnserverBot` could be explained by a number of reasons, including the fact that they probably share common engines.

- The remaining malware families seem rather unrelated, and no significant evolutionary relationship can be inferred. Note, too, that distances approach 1 in this area of the dendrogram, which suggest a very weak connection.

5. Related Work

Tools for automatically detecting and classifying malware have proliferated over the last years. There are two basic types of malware detection techniques according to how code is analyzed: *static* and *dynamic* analysis. Static analysis techniques attempt to identify malicious code by unpacking and disassembling (or decompiling) the specimen and looking into the resulting code. Contrarily, dynamic analysis seeks to identify malicious behaviors after deploying and executing the sample in a controlled and instrumented environment.

Both static and dynamic analysis extract and analyze a number of features from the sample being scrutinized. In this regard, several techniques have been proposed to assist the analyst in classifying the malware, including approaches based on machine learning [24], data mining [25, 26], expert systems [27], and clustering [28]. We refer the reader to [14] for an excellent survey on automated malware analysis techniques.

Malicious applications targeting smartphones, particularly ANDROID OS, have rocketed over the last few years [2], evolving from relatively simple apps causing annoyance to complex and sophisticated pieces of code designed for profit, sabotage or espionage [8]. Current trends in malware engineering suggest that malicious software will continue to evolve its sophistication [29], in part due to the availability of reuse-oriented development methodologies. This is particularly important when analyzing *piggyback attacks*. In this regard, one of the most common distribution strategy for smartphone malware consists of repackaging popular applications and distributing them through alternative markets with additional malicious code attached (i.e., piggybacked) [19]. In DENDROID, these two properties are exploited to facilitate analysis and detection.

A substantial number of research works have been recently proposed to enhance malware detection and classification based on a variety of techniques [29]. Static analysis techniques have recently gained momentum as efficient mechanisms for market protection; see, e.g., [30], [31], [32], [16], [33], [34], [35], and [36] to name a few. More precisely, researchers have explored different ways to detect piggybacked malware [37], [38], [39] by clustering malware instances intro classes according to some similarity dependencies. While early approaches use syntactic analysis such as string-based matching [37], recent approaches elaborate on semantic analysis [38], e.g., program dependency graphs, as they are resilient to code obfuscation. In this regard, Desnos [37] apply several compression algorithms to compute normalized information distances between two applications based on `Kolmogorov` complexity measurement. Their algorithm first identifies which methods are identical, and calculates the similarity of the remainder methods using Normalized Compression Distances (NCD). `DNADroid` [38] focuses on detecting cloned apps by comparing program dependency graphs (PDG) between methods, detecting semantic similarities through graph isomorphisms. A system called DroidMOSS is proposed in [39] for detecting repackaged applications based on a fuzzy hashing technique. Distinguishing features are extracted in the form of fingerprints and compared with other applications to identify similarities. These features are computed by applying traditional hash

functions to pieces of code of variable size. The size of the pieces is bounded by smaller chunks of fixed size called reset points. A chunk is considered a reset point when the resulting hash is a prime number. Then, the edit distance is calculated between two applications by comparing their fingerprints. Finally, authors in [40] present a system for detecting similar ANDROID OS applications. They propose an optimization strategy over the representation of apps as an alternative to k-grams based on feature hashing. Feature hashing reduces the dimensionality of the data analyzed and, therefore, the complexity of computing similarities among their feature sets. In particular, they rely on the `Jaccard` similarity over the set of bit vectors representing each application. More recently, several other related works have studied different strategies to recommend *appropriate* apps to users based on contextual preferences [41], which is particularly relevant due to a recent rise in the so-called grayware [8].

DENDROID shares with some of these works the idea of finding a suitable representation for pieces of code, in particular one that facilitates measuring similarities. However, our use of code structures at the method level is more fine grained, resulting very useful to tell apart new specimens from those that are a minor variant of a known strand. Besides, by breaking samples into structural components we build a large database that can be mined with well-researched techniques such as those currently incorporated in DENDROID. Similarity, classification, and hierarchical clustering rely on such structural information, which is a major difference between our proposal and other existing approaches. For example, Hanna et al. [40] apply hierarchical clustering over the k-gram hashes (and pursuing goals different to ours), rather than on high-level representation of code structures.

In other domains, many works have applied text mining and information retrieval techniques for decision making and classification, such as for example [42] and [43]. Furthermore, recent approaches have also used text mining for detecting similarities [44, 45]. To the best of our knowledge, DENDROID is the first attempt to apply text mining techniques over malicious code structures.

6. Conclusions and Future Work

In this paper, we have proposed a text mining approach to automatically classify smartphone malware samples and analyze families based on the code structures found in them. Our proposal is supported by a statistical analysis of the distribution of such structures over a large dataset of real examples. Our findings point out that the problem bears strong resemblances to some questions arising in automated text classification and other information retrieval tasks. By adapting the standard Vector Space Model commonly used in these domains, we have explored the suitability of such techniques to measure similarity among malware samples, and to classify unknown samples into known families. Our experimental results suggest that this technique is fast, scalable and very accurate. We have subsequently studied the use of hierarchical clustering to derive dendograms that can be understood as phylogenetic trees for malware families. This provides the analyst with a means to analyze the relationships among families,

the existence of common ancestors, the prevalence and/or extinction of certain code features, etc. As discussed in this paper, automated tools such as these will be instrumental for analysts to cope with the proliferation and increasing sophistication of malware.

The work presented in this paper can be improved and extended in a number of ways. At the time of writing this, we are focussing our efforts in four main working directions:

- Address the dimensionality problem. Feature vectors eventually become unmanageably large as a consequence of extending the model with new code structures. In classical text mining, this problem can be easily solved by the so-called Latent Semantic Analysis (LSA) [46]. Roughly speaking, LSA performs a singular value decomposition to identify a reduced set of dimensions (in our case, linear combinations of code structures) that suffice to model the population of instances.

- Study obfuscation strategies that seek to defeat classification by modifying the code structures of a malware instance while preserving its intended purpose (semantics).

- Enrich code structures with an associated semantic describing its functionality. This could automate even further the task of reasoning about the goals, tactics, etc. of a piece of malicious software.

- Automated identification of countermeasures. If malware sample x can be counteracted by measure m and sample y is similar to x, there is chance that countermeasures for y will be similar to m. Enriching code structures with potential countermeasures would facilitate reasonings such as the one above and can be instrumental in scenarios where a rapid response is required, or just to assist the analyst in engineering solutions to thwart a newly found piece of malware.

Acknowledgements

We are very grateful to Yajin Zhou and Xuxian Jiang from North Carolina State University for providing us with access to the samples contained in the Android Malware Genome Project, which has been essential in this work.

References

[1] H. Dediu, "When will tablets outsell traditional pcs?" March 2012, http://www.asymco.com/2012/03/02/when-will-the-tablet-market-be-larger-than-the-pc-market/.

[2] Juniper, "2011 mobile threats report," Juniper Networks, Tech. Rep., 2012.

[3] L. Goasduff and C. Pettey, "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth," Visited April 2012, http://www.gartner.com/it/page.jsp?id=1924314.

[4] Nielsen, "State of the appnation —a year of change and growth in u.s. smartphones," Nielsen, Tech. Rep., 2012.

[5] R. van der Meulen and J. Rivera, "Gartner says worldwide mobile phone sales declined 1.7 percent in 2012," Visited March 2013, http://www.gartner.com/newsroom/id/2335616.

[6] E. Chin, A. P. Felt, V. Sekar, and D. Wagner, "Measuring user confidence in smartphone security and privacy," in *Symposium on Usable Privacy and Security*. Washington: Advancing Science, Serving Society, March 2012.

[7] J. Fenske, "Biometrics in new era of mobile access control," *Biometric Technology Today*, vol. 2012, no. 9, pp. 9–11, 2012.

[8] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, "A survey of mobile malware in the wild," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*, ser. SPSM '11. New York, NY, USA: ACM, 2011, pp. 3–14.

[9] K. Dunham, *Mobile malware attacks and defense*. Syngress, 2008.

[10] D. Shih, B. Lin, H. Chiang, and M. Shih, "Security aspects of mobile phone virus: a critical survey," *Industrial Management & Data Systems*, vol. 108, no. 4, pp. 478–494, 2008.

[11] F-Secure, "Mobile threat report q1 2012," F–Secure, Tech. Rep., April 2012, "http://www.f-secure.com/weblog/archives/MobileThreatReport_Q1_2012.pdf".

[12] McAfee, "Threats report:fourth quarter 2012," McAfee, Tech. Rep., January 2013, http://www.mcafee.com/us/resources/reports/rp-quarterly-threat-q4-2012.pdf.

[13] M. Schipka, "Dollars for downloading," *Network Security*, vol. 2009, no. 1, pp. 7–11, 2009.

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel, "A survey on automated dynamic malware-analysis techniques and tools," *ACM Comput. Surv.*, vol. 44, no. 2, pp. 6:1–6:42, Mar. 2012.

[15] S. Cesare and Y. Xiang, "Classification of malware using structured control flow," in *Proceedings of the Eighth Australasian Symposium on Parallel and Distributed Computing-Volume 107*. Australian Computer Society, Inc., 2010, pp. 61–70.

[16] M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, 2012.

[17] E. Ruzgar and K. Erciyes, "Clustering based distributed phylogenetic tree construction," *Expert Systems with Applications*, vol. 39, no. 1, pp. 89–98, 2012.

[18] V. Rastogi, Y. Chen, and X. Jiang, "Droidchameleon: evaluating android anti-malware against transformation attacks," in *Proceedings of the 8th ACM SIGSAC symposium on Information, computer and communications security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 329–334.

[19] Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (Oakland 2012)*, May 2012.

[20] F. Nielson, H. Nielson, and C. Hankin, *Principles of Program Analysis*. Springer, 1999.

[21] A. Desnos. (Visited June 2013) Androguard reverse engineering tool. [Online]. Available: http://code.google.com/p/androguard/

[22] G. Salton, A. Wong, and C.-S. Yang, "A vector space model for automatic indexing," *Communications of the ACM*, vol. 18, no. 11, pp. 613–620, 1975.

[23] S. Tan, "Neighbor-weighted k-nearest neighbor for unbalanced text corpus," *Expert Systems with Applications*, vol. 28, no. 4, pp. 667–671, 2005.

[24] Y.-T. Hou, Y. Chang, T. Chen, C.-S. Laih, and C.-M. Chen, "Malicious web content detection by machine learning," *Expert Systems with Applications*, vol. 37, no. 1, pp. 55–60, 2010.

[25] S.-H. Liao, P.-H. Chu, and P.-Y. Hsiao, "Data mining techniques and applications a decade review from 2000 to 2011," *Expert Systems with Applications*, vol. 39, no. 12, pp. 11 303 – 11 311, 2012.

[26] S. Thiruvadi and S. C. Patel, "Survey of data-mining techniques used in fraud detection and prevention," *Information Technology Journal*, vol. 10, no. 4, pp. 710–716, 2011.

[27] S. Sahin, M. R. Tolun, and R. Hassanpour, "Hybrid expert systems: A survey of current approaches and applications," *Expert Systems with Applications*, vol. 39, no. 4, pp. 4609–4617, 2012.

[28] S. J. Delany, M. Buckley, and D. Greene, "Sms spam filtering: methods and data," *Expert Systems with Applications*, vol. 39, no. 10, pp. 9899–9908, 2012.

[29] G. Suarez-Tangil, J. E. Tapiador, P. Peris, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," 2013.

[30] M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Proceedings of the Network and Distributed System Security Symposium*, 2011.

[31] H. Peng, C. Gates, B. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy, "Using probabilistic generative models for ranking risks of android apps," in *Proceedings of the 2012 ACM conference on Computer and communications security.* ACM, 2012, pp. 241–252.

[32] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang, "Riskranker: scalable and accurate zero-day android malware detection," in *Proceedings of the 10th international conference on Mobile systems, applications, and services.* ACM, 2012, pp. 281–294.

[33] S. Rosen, Z. Qian, and Z. M. Mao, "Appprofiler: a flexible method of exposing privacy-related behavior in android applications to end users," in *Proceedings of the third ACM conference on Data and application security and privacy.* ACM, 2013, pp. 221–232.

[34] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang, "Chex: statically vetting android apps for component hijacking vulnerabilities," in *Proceedings of the 2012 ACM conference on Computer and communications security.* ACM, 2012, pp. 229–240.

[35] K. O. Elish, D. D. Yao, B. G. Ryder, and X. Jiang, "A static assurance analysis of android applications," Virginia Polytechnic Institute and State University, Tech. Rep., 2013.

[36] S.-H. Seo, A. Gupta, A. M. Sallam, E. Bertino, and K. Yim, "Detecting mobile malware threats to homeland security through static analysis," *Journal of Network and Computer Applications*, no. 0, 2013.

[37] A. Desnos, "Android: Static analysis using similarity distance," in *System Science (HICSS), 2012 45th Hawaii International Conference on.* IEEE, 2012, pp. 5394–5403.

[38] J. Crussell, C. Gibler, and H. Chen, "Attack of the clones: Detecting cloned applications on android markets," *Computer Security–ESORICS 2012*, pp. 37–54, 2012.

[39] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy.* ACM, 2012, pp. 317–326.

[40] S. Hanna, L. Huang, E. Wu, S. Li, C. Chen, and D. Song, "Juxtapp: A scalable system for detecting code reuse among android applications," in *Proceedings of the 9th Conference on Detection of Intrusions and Malware & Vulnerability Assessment*, 2012.

[41] "Which app? a recommender system of applications in markets: Implementation of the service for monitoring users interaction," *Expert Systems with Applications*, vol. 39, no. 10, pp. 9367 – 9375, 2012.

[42] C. Chibelushi, B. Sharp, and A. Salter, "A text mining approach to tracking elements of decision making: a pilot study." in *NLUCS*. Citeseer, 2004, pp. 51–63.

[43] V. Gadia and G. Rosen, "A text-mining approach for classification of genomic fragments," in *Bioinformatics and Biomeidcine Workshops, 2008. BIBMW 2008. IEEE International Conference on*. IEEE, 2008, pp. 107–108.

[44] A. Y. Rodriguez-Gonzalez, J. F. Martinez-Trinidad, J. A. Carrasco-Ochoa, and J. Ruiz-Shulcloper, "Mining frequent patterns and association rules using similarities," *Expert Systems with Applications*, vol. 40, no. 17, pp. 6823 – 6836, 2013.

[45] G. Oberreuter and J. D. Velsquez, "Text mining applied to plagiarism detection: The use of words for detecting deviations in the writing style," *Expert Systems with Applications*, vol. 40, no. 9, pp. 3756 – 3763, 2013.

[46] D. Thorleuchter and D. V. d. Poel, "Technology classification with latent semantic indexing," *Expert Systems with Applications*, 2012.