Structural Detection of Android Malware using Embedded Call Graphs

Hugo Gascon
University of Göttingen
Göttingen, Germany

Fabian Yamaguchi
University of Göttingen
Göttingen, Germany

Daniel Arp
University of Göttingen
Göttingen, Germany

Konrad Rieck
University of Göttingen
Göttingen, Germany

ABSTRACT

The number of malicious applications targeting the Android system has literally exploded in recent years. While the security community, well aware of this fact, has proposed several methods for detection of Android malware, most of these are based on permission and API usage or the identification of expert features. Unfortunately, many of these approaches are susceptible to instruction level obfuscation techniques. Previous research on classic desktop malware has shown that some high level characteristics of the code, such as function call graphs, can be used to find similarities between samples while being more robust against certain obfuscation strategies. However, the identification of similarities in graphs is a non-trivial problem whose complexity hinders the use of these features for malware detection. In this paper, we explore how recent developments in machine learning classification of graphs can be efficiently applied to this problem. We propose a method for malware detection based on efficient embeddings of function call graphs with an explicit feature map inspired by a linear-time graph kernel. In an evaluation with 12,158 malware samples our method, purely based on structural features, outperforms several related approaches and detects 89% of the malware with few false alarms, while also allowing to pin-point malicious code structures within Android applications.

Categories and Subject Descriptors

C.2.0 [**Computer-Communication Networks**]: General—*Security and Protection*; K.6.5 [**Computing Milieux**]: Management of Computing and Information Systems—*Security and Protection - Invasive software*; I.5.1 [**Pattern Recognition**]: Models—*Statistical*

Keywords

Malware Detection; Graph Kernels; Machine Learning

AISec'13, November 4, 2013, Berlin, Germany.

ACM 978-1-4503-2488-5/13/11 ...$15.00.
http://dx.doi.org/10.1145/2517312.2517315

1. INTRODUCTION

The growing popularity of smartphones and tablet computers has made mobile platforms a prime target for attack. In a recent study conducted by Juniper Networks, the number of malicious mobile applications observed in the wild is reported to have grown exponentially at a rate of 614% between March 2012 and March 2013 [23]. Moreover, 92% of these applications are found to target the Google Android platform, creating the need for effective defense mechanisms to protect Android systems.

To account for this threat, the official Android market is regularly scanned for possibly malicious code. Unfortunately, this process is often ineffective [22] and the market is abused to deliver malware on a daily basis. Furthermore, Android applications are also made available in third party repositories, which often implement no methods for malware detection at all. As a result, Android malware detection has become a vivid area of research in both industry and academia in recent years.

Several researchers have proposed to detect malware based on manually specified expert features, such as the permissions requested by applications [10, 14, 31] or the usage of specific API functions [16, 44]. While these approaches add an additional security layer to the Android platform, several of the proposed mechanisms are easy to circumvent, for example using kernel-based exploits or API-level rewriting [18]. Moreover, the designed expert features often target a specific type of malware and hence fail to provide a generic protection from malicious applications.

Fortunately, the vast majority of newly discovered malware samples are variations of existing malware, and thus detecting similarities to known malware has shown to be a promising approach [see 7, 16, 42]. Identifying variations of code, however, is an involved task as small changes at the source code may already have drastic effects on compiled code: instructions may be reordered, branches may be inverted or the allocation of registers may change [see 9]. To make matters worse, such changes are often introduced deliberately by malware to evade detection.

Researchers dealing with the detection of malware targeting desktop systems have discovered that high-level properties of code, in particular *function call graphs*, offer a suitably robust representation to account for these variations [21, 25]. By contrast, existing approaches for Android malware detection mostly focus on contextual information, such as permission and API usage, and do not model the structural composition of code.

In this paper, we therefore present a method for structural detection of Android malware on a large scale. To this end, we extract function call graphs from Android applications and employ an explicit mapping inspired by a linear-time graph kernel to efficiently map call graphs to an explicit feature space. A support vector machine is then trained to distinguish between benign and malicious applications. In an empirical evaluation on a total of 12,158 malware samples, this approach is shown to be highly effective, enabling a detection of 89% of the malware with only 1% false positives, by analyzing the structure of the underlying code only.

In summary, our main contributions are the following.

- **Generic labeling of Dalvik functions.** We present a generic labeling scheme for Dalvik code that enables us to construct labeled function calls graphs without information about function names.
- **Explicit embedding of call graphs.** We derive a feature map inspired by graph kernels that allows for embedding function call graphs in a vector space capturing structural relationships.
- **Structural detection of Android malware.** The vectorial representation of function call graphs finally enables us to detect Android malware with high accuracy using machine learning techniques.

The rest of this paper is structured as follows: we introduce our method for analyzing function call graphs and detecting Android malware in Section 2 and present a detailed evaluation in Section 3. Limitations of our approach and related work are discussed in Section 4 and 5 respectively. Section 6 concludes the paper.

2. STRUCTURAL MALWARE DETECTION

Our method for the detection of Android malware is based on two key observations. First, malicious functionality of an Android application often concentrates on only a small number of its functions and second, similar malicious code is often found throughout the malware landscape as attackers reuse existing code to infect different applications. Our method therefore strives to identify subgraphs of the function call graph representing known malicious code. This is, however, not trivial, in particular since no polynomial time solution exists to test whether two graphs are isomorphic.

In consequence, several solutions have been developed for inexact matching. Some of these methods rely on suboptimal strategies such as the graph edit distance [21] or the identification of maximum common subgraphs, while other ad hoc solutions propose the serialization of the graph structure [5] as a way to measure similarity. In most setups, this similarity metric is later used in a neighbor search to identify close candidates to a test sample.

When it comes to classification tasks, kernel-based support vector machines (SVM) have been proven extremely powerful. Graph kernels have emerged as a solution to let SVM operate efficiently in the graph space. A graph kernel is, in short, a kernel function that computes an inner product on graphs. These kernels have been proposed at several occasions to address graph classification problems in chemistry and bioinformatics, however, their applicability to static malware analysis remains largely unexplored.

Some graph kernels are designed to operate only on unlabeled graphs or are unable to be evaluated on graphs with more than a few hundreds of nodes. Moreover, many of these kernels induce only an implicit feature space, which makes it impossible to determine the features predominantly responsible for the classification of a sample.

The use of graph kernels for the task of malware detection allows to abstract the code into a representation that enables learning its underlying structure. However, function call graphs have thousands of nodes and are characterized as directed labeled graphs. Therefore, it is necessary to apply a graph kernel that can not only deal with these specificities, but can also operate on a large number of nodes efficiently. In this paper, we show how the combination of a graph kernel that meets these requirements and a convenient embedding in an equivalent explicit vector space can be applied to the problem of malware detection successfully.

The resulting method comprises the following major steps:

1. *Call graph extraction and labeling.* The function call graph for an application is extracted, which contains a node for each function of the application. Nodes are labeled according to the instructions contained in their corresponding functions (Section 2.1).
2. *Hashing of neighborhoods.* For each node in the function call graph, a hash-value is calculated over the node and direct neighboring nodes, allowing occurrences of graph substructures to be efficiently and explicitly enumerated (Section 2.2).
3. *Feature space embedding.* Samples are embedded using an explicit map inspired by the neighborhood hash graph kernel introduced by Hido and Kashima [20]. The map is designed such that evaluating an inner product in the feature space is equivalent to computing the respective graph kernel (Section 2.3).
4. *Learning and feature analysis.* A linear support vector machine is trained to learn a detection model that is able to classify applications as benign or malicious. The explicit feature space is then used to highlight subgraphs in an application likely to contain malicious functionality (Section 2.4).

In the following sections, each of these steps is described in greater detail.

2.1 Call Graph Extraction and Labeling

In the first step of our method, applications are disassembled and their function call graphs are extracted using the Androguard framework [8]. In addition, nodes of the function call graph are labeled to characterize their content conveniently by short bit sequences.

Intuitively, the extracted function call graphs are directed graphs containing a node for each of the application's functions and edges from callers to callees. Moreover, a *labeled* function call graph can be constructed by attaching a label to each node. Formally, this graph can be represented as a 4-tuple $G = (V, E, L, \ell)$, where V is a finite set of nodes and each node $v \in V$ is associated with one of the application's functions. $E \subseteq V \times V$ denotes the set of directed edges, where an edge from a node v_1 to a node v_2 indicates a call from the function represented by v_1 to the function represented by v_2. Finally, L is the multiset of labels in the

graph and $\ell : V \to L$ is a labeling function, which assigns a label to each node by considering properties of the function it represents.

The design of the labeling function ℓ is crucial for the success of our method. While in principle, a unique label could be assigned to each node, this would not allow the method to exploit properties shared between functions. By contrast, a suitable labeling function maps two nodes onto the same label if their functions share properties relevant to the detection task. Moreover, labeling must be robust against small changes in the code such as identifier renaming or branch inversion.

To meet these requirements, we propose to label nodes according to the type of the instructions contained in their respective functions. Reviewing the Dalvik specification, we define 15 distinct categories of instructions based on their functionality as shown in Table 1. Each node can thus be labeled using a 15-bit field, where each bit is associated with one of the categories.

Category	Bit	Category	Bit
`nop`	1	`branch`	9
`move`	2	`arrayop`	10
`return`	3	`instanceop`	11
`monitor`	4	`staticop`	12
`test`	5	`invoke`	13
`new`	6	`unop`	14
`throw`	7	`binop`	15
`jump`	8		

Table 1: List of instruction categories and their corresponding bit in the label assigned to each node.

Formally, the function ℓ can be defined as follows: We denote the set of categories by $\mathcal{C} = \{c_1, c_2, \cdots, c_m\}$ and the function associated with a node v by f_v. The label $\ell(v)$ of a node $v \in V$ is then a bit field of length m, i.e., $\ell(v) = [b_1(v), b_2(v), \cdots, b_m(v)]$ where

$$b_c(v) = \begin{cases} 1 & \text{if } f_v \text{ contains an instruction from category } c \\ 0 & \text{otherwise.} \end{cases}$$

Consequently, the set of labels L is given by a subset of all possible 15-bit sequences. As an example, Figure 1 shows the disassembled code of a function named `openWebURL` and the categories assigned to each of its instructions. Note that the function contains the `new-instance` instruction, which is used to instantiate an object. This instruction is part of a larger set of instructions denoted as `new` and associated with the sixth bit of the label. The sixth bit is therefore set in the resulting function label.

2.2 Hashing of Neighborhoods

Upon labeling nodes in the function call graph, each function is characterized by the instructions it contains. However, our method strives to model the composition of functions and thus the neighborhood of a function must be taken into account. To this end, for each node, we compute a *neighborhood hash* over all of its direct neighbors in the function call graph, a procedure inspired by the neighborhood hash graph kernel (NHGK) originally proposed by Hido and Kashima [20].

Instructions	Category	Bit
`#LX;->openWebURL(Ljava/lang/String;)V`		
`new-instance v0, Landroid/content/Intent;`	new	6
`const-string v1, android.intent.action.VIEW`	move	2
`invoke-static v4, Landroid/net/Uri;-> parse(Ljava/lang/String;)Landroid/net/Uri;`	invoke	13
`move-result-object v2`	move	2
`invoke-direct v0, v1, v2, Landroid/content/Intent;-> <init>(Ljava/lang/String; Landroid/net/Uri;)V`	invoke	13
`invoke-virtual v3, v0, Lru/getcorvin/mettir/MetrDotel;-> startActivity(Landroid/content/Intent;)V`	invoke	13
`return-void`	return	3

Bit	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
Label	0	1	1	0	0	1	0	0	0	0	0	0	1	0	0

Figure 1: Labeling example of a method from its code. Every opcode belongs to a category, which is a associated to a certain bit in the label.

The NHGK is a so called decomposition kernel as defined by Haussler [19]. As such, it is a kernel operating over an enumerable set of subgraphs in a labeled graph. It has low computational complexity and high expressiveness of the graph structure, but its main advantage is that it is able to run in time linear in the number of nodes and can therefore process graphs with thousands of nodes such as the function call graphs of Android applications.

The main idea behind the NHGK is to condense the information contained in a neighborhood into a single hash value. This value is calculated over the labels of a neighborhood and represents the distribution of the labels around a central node. It thus allows us to enumerate all neighborhood subgraphs in linear time without running an isomorphism test over all pairs of neighborhoods.

The computation of the hash for a given node v and its set of adjacent nodes V_v is defined by the operation

$$h(v) = r(\ell(v)) \oplus \left(\bigoplus_{z \in V_v} \ell(z) \right) \tag{1}$$

where $\oplus$ represents a bit-wise XOR on the binary labels and r denotes a single-bit rotation to the left. This computation can be carried out in constant time for each node, more specifically in $\Theta(md)$ time where d is the maximum outdegree and m the length of the binary label.

The *neighborhood hash* of a complete graph G denoted by $G_h = h(G) = (V, E, L_h, h(\cdot))$ is then obtained by calculating hashes for each node individually and replacing the original labels with the calculated hash values. This creates an additional linear dependence of the computation time on the number of nodes in the graph. Furthermore, it can be noted that the hash values have the same length m as the original label, however, they aggregate information spread across neighboring nodes. Moreover, the hash values are independent of the actual order in which children are processed, and thus sorting is not necessary.

Hido and Kashima also consider applying the neighborhood hash iteratively to aggregate information across neighbors up to a path length p. The neighborhood hash of order p can then be defined recursively as $G^{(p+1)} = h(G^p)$. Choosing p larger than one still allows to construct a valid decomposition kernel, however, higher values of p also lead to an increased number of overlapping substructures.

Since we are particularly interested in designing an explicit representation of the kernel feature space that is easy to interpret by analysts, we thus fix $p = 1$ in order to limit the complexity of the feature space.

2.3 Feature Space Embedding

The neighborhood hash graph kernel function, evaluates the count of common identical substructures in two graphs, which after the hashing, is the number of shared node labels. Considering that several nodes can be labeled with the same hash, the kernel value can be represented as the size of the intersection of the multisets L_h and L'_h for two function call graphs G_h and G'_h, that is,

$$K(G_h, G'_h) = |L_h \cap L'_h| \tag{2}$$

For the specific application of malware analysis, our goal is to find an explicit representation that is equivalent to that of the graph kernel. In this vector space, a linear SVM can be used to learn a model that is able to (a) classify samples into benign and malicious applications and (b) allows for an interpretation of its decisions. In order to achieve this, we abstain from using the implicit kernel function K, but instead embed every sample in a feature space whose inner product is equivalent to the graph kernel.

To this end, we start by considering the histogram of the multiset L_h as $H = \{a_1, a_2, \cdots, a_N\}$, where $a_i \in \mathbb{N}$ indicates the occurrences of the i-th hash in G_h. The number of shared elements between two multisets can be calculated by sorting all elements of a certain type and counting the minimum number of elements of this type that are present in both multisets. This is known as the multiset intersection. If the size of the intersection of two histograms H and H' of length N is defined as

$$S(H, H') = \sum_{i=1}^{N} \min(a_i, a'_i) \tag{3}$$

it becomes apparent that the kernel defined in Eq. (2) can be also phrased using the intersection of the histograms for two graphs G_h and G'_h as

$$K(G_h, G'_h) = S(H, H'). \tag{4}$$

Barla et al. [2] show that this histogram intersection can be indeed adopted in kernel-based methods and propose a feature mapping, such that S is an inner product in the induced vector space. For this purpose, each histogram H is mapped to a P-dimensional vector $\phi(H)$ as follows

$$\phi(H) = \left(\underbrace{\overbrace{1, \cdots, 1}^{a_1}, \overbrace{0, \cdots, 0}^{M-a_1}}_{\text{bin } 1}, \cdots, \underbrace{\overbrace{1, \cdots, 1}^{a_N}, \overbrace{0, \cdots, 0}^{M-a_N}}_{\text{bin } N} \right) \tag{5}$$

where M is the maximum value of all bins in the dataset, N is the number of bins in each histogram and $P = NM$ is the dimension of the vector.

In this representation, each bin i of the histogram is associated with M dimensions in the vector space. These dimensions are filled with 1's according to the value of a_i, whereas the remaining $M - a_i$ dimensions are set to 0. As a result, the sum of the M dimensions associated with the i-th bin is equal to a_i and moreover the sum of all dimensions in $\phi(H)$ is equal to the sum of all bins in the histogram.

By putting the different steps together, we can finally show that the inner product in the vector space induced by Eq. (5) indeed resembles the neighborhood hash graph kernel given in Eq. (2). That is, we have

$$K(G_h, G'_h) = S(H, H') = \langle \phi(H), \phi(H') \rangle. \tag{6}$$

The interested reader is referred to the original work of Barla et al. [2], which provides a more detailed analysis of histogram intersections and this mapping.

The mapping ϕ finally allows us to embed every call graph in a feature space, where a linear SVM can be used for efficiently learning with hundreds of thousands of graphs each containing thousands of nodes and edges.

2.4 Learning and Feature Analysis

A two-class classification problem can be then posed and solved by means of a linear SVM, which learns a linear separation of the given classes with a maximum margin [13]. The decision function f of the linear SVM is given by

$$f(G_h) = \langle \phi(H), w \rangle + b, \tag{7}$$

where $w \in \mathbb{R}^P$ is the direction of the hyperplane and b the offset from the origin of the vector space. In this setting, a function call graph G_h is classified as malicious if $f(G_h) > 0$ and as benign if $f(G_h) \leqslant 0$.

In order to identify what substructures of G_h contribute to this decision, it is necessary to reverse the expansion performed in Eq. (5). In particular, we compute an aggregated weight $\hat{w}_i$ for each bin i of the histogram (corresponding to the i hash value of the graph G_h). Formally, this is achieved for the i-th bin of the histogram as follows

$$\hat{w}_i = \sum_{j=iM}^{(i+1)M} w_j. \tag{8}$$

The largest of these aggregated weights allows us to highlight those neighborhoods in a given graph G_h that predominantly influence the decision and can be interpreted as typically belonging to the malicious class. That is, if the weight $\hat{w}_i$ of the i-th bin is large, all nodes labeled with the corresponding hash value significantly contribute to the decision of the SVM and thus likely reflect malicious functionality.

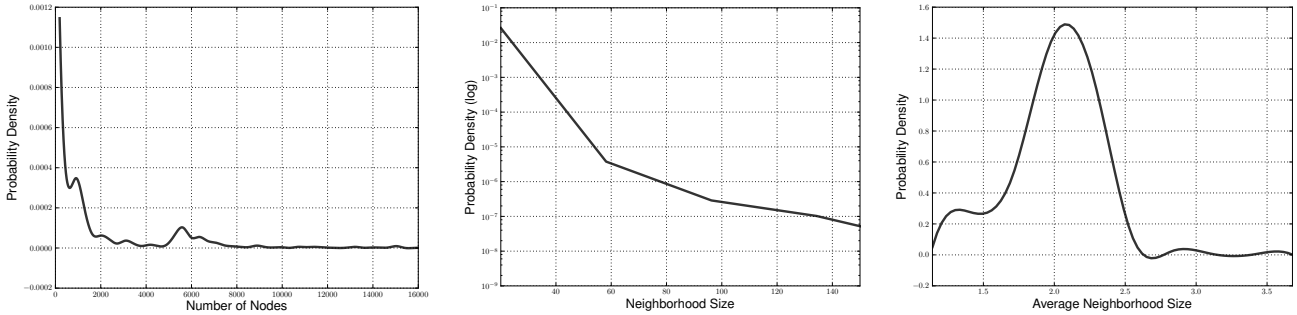
As a last step and as we will see in the evaluation section, a relevance map can be constructed by shading each node in the graph with the sum of the weights of the neighborhoods it belongs to. This will produce a representation that is ready for visual inspection and further feature analysis by a malware analyst.

3. EVALUATION

Our method is evaluated on a large data set of real Android applications. We begin by providing a detailed description of our data set in Section 3.1 and demonstrating that our method scales linearly in the number of functions contained in an application. We then proceed to evaluate our method's ability to detect malicious Android applications in Section 3.2. Finally, a case study on a malware sample from a popular family is given in Section 3.3 to highlight the practical advantages of an explicit feature space.

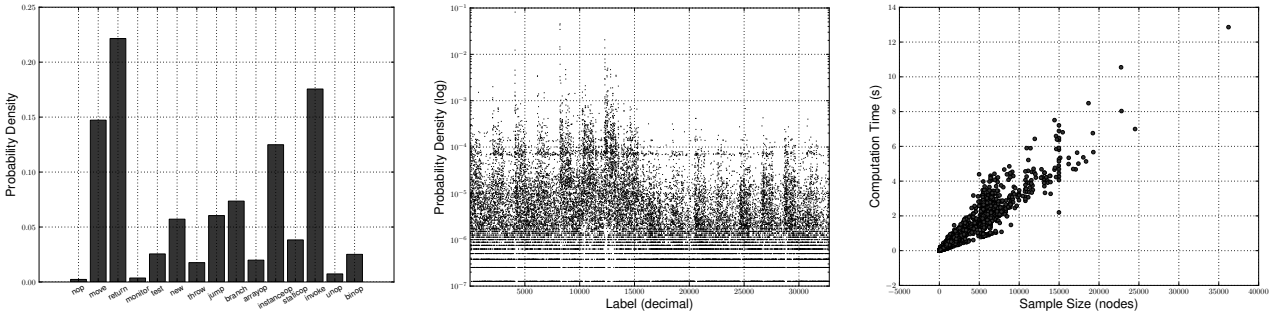
3.1 Data Set

The data set used in our evaluation consists of 135,792 benign and 12,158 malicious Android applications obtained from both the official Google Play store as well as popular third-party markets. To decide whether an application



(a) Sample size distribution (b) Neighborhood size distribution (c) Average neighborhood size

Figure 2: Characteristics of the dataset: (a) distribution of the number of nodes per sample, (b) size of neighborhoods across the dataset and (c) the average size of a neighborhood within an individual sample.



(a) Distribution of Dalvik op-codes (b) Distribution of 15-bit labels (c) Runtime of feature vector

Figure 3: Distribution of (a) instruction categories and (b) labels over all functions in the dataset, and (c) runtime of the embedding as a function of the number of nodes in the graph.

is malicious or not, we employ 10 commercial malware detectors and consider a sample to be benign if none of the scanners triggers an alert while in all other cases we label the application as malicious. We proceed to extract function call graphs from each application to examine the properties of the data set in more detail.

Figure 2(a) shows the sample size distribution obtained by determining the number of functions in each application. We observe that most applications are rather small, consisting of less than 2,000 functions, however, applications consisting of up to 16,000 functions exist and must therefore be accounted for by our method. Furthermore, we examine the number of outgoing calls made by functions as illustrated in the distribution shown in Figure 2(b). We find that the majority of functions contain less than 50 outgoing calls. Moreover, Figure 2(c) shows the average number of outgoing calls to be only slightly above two, thus making the computation of neighborhood hashes very efficient.

We further examine the distribution of the different categories of Dalvik instructions as shown in Figure 3(a). We observe that the `return` class has the highest probability, as an instruction of this type is present in every method. On the contrary, certain types of instructions rarely occur. For example, the `nop` instruction, which performs no operation, or the `monitor` instruction, which is used for synchronization. This observation that some instructions are much more common than others seems problematic at first, as it suggests that labels will not be equally distributed, possibly distorting the distribution of updated node labels once the neighborhood hash is computed. However, Figure 3(b) shows how the distribution of 15-bit node labels is not significantly skewed and every possible label appear in the call graphs extracted from the dataset.

Finally, we evaluate the runtime performance of the feature extraction process presented in Section 2.3. Figure 3(c) shows the time required to process a sample plotted against the sample size. We observe that the computation of feature vectors indeed scales linearly in the number of functions contained in a sample.

3.2 Malware Detection

The experiment for malware detection is posed as a supervised two-class classification problem. Random subsets of benign and malicious applications from the dataset are embedded and split into training and testing sets which are then fed to the linear SVM algorithm.

Figure 4 shows a ROC curve which illustrates the performance of the binary classifier. Bounded to a maximum false-positive rate (FPR) of 10% the area under the curve reaches a value of 0.935. The performance measure is depicted along three recently proposed methods for static detection of Android malware that we have also tested on our dataset: *Kirin* [10], *RCP* [35] and the approach proposed by Peng et. al in [31], where we implement the latter using

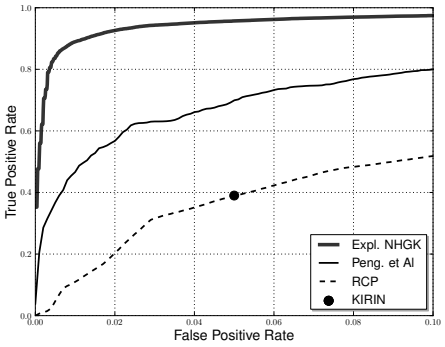


Figure 4: Detection performance as ROC curve for our method and related detection methods.

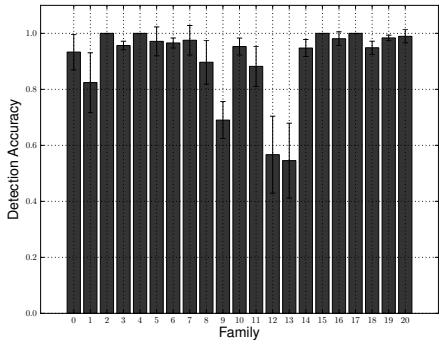


Figure 5: Detection accuracy for the largest malware families in the dataset (see Table 2)

SVM instead of a naive bayes classifier. Our method significantly outperforms these approaches and detects 89% of the malware at a false-positive rate of only 1%—corresponding to only one false alarm when installing 100 applications on a smartphone. The related methods enable a detection between 10%-50% at the same false-positive rate, likely due to their focus on Android permissions which reflect only little of the information inside a malicious application.

Several of the malware samples in our dataset are labeled with a particular malware family. Table 2 shows a list of the 21 largest families in our dataset. Using this labeling, we are able to show in Figure 5 the detection performance of the SVM for the individual families. For several families a detection rate above 90% is achieved. It can be noted that certain malware families, like 12 and 13, are detected with lower accuracy than the rest. However, the standard error for these measures is also considerable larger than that of the families with better performance, what indicates that these measures have a lower confidence as a result of the limited number of samples available.

3.3 Case Study: FakeInstaller

A weakness of many machine learning approaches to malware detection [e.g., 35, 39] is their inability to provide interpretable results. Therefore, it is often difficult for analysts to understand why samples are marked as benign or malicious. Our method addresses this shortcoming by using an explicit feature space representation. The advantage of this representation is that it allows a weight to be attached

Id	Family	#	Id	Family	#
0	DroidDream	71	11	Glodream	67
1	SMSreg	49	12	ExploitLinuxLotoor	46
2	Kmin	145	13	Iconosys	52
3	FakeInstaller	859	14	GinMaster	271
4	FakeRun	58	15	MobileTx	55
5	Yzhc	31	16	FakeDoc	116
6	DroidKungFu	570	17	Hamob	31
7	SendPay	44	18	Opfake	606
8	Adrd	82	19	Plankton	545
9	Adware.Airpush	175	20	Geinimi	84
10	BaseBridge	319			

Table 2: List of malware families with largest number of samples in the dataset.

to each node of the call graph according to its importance for the decision. This allows the analyst to highlight nodes corresponding to functionality deemed malicious, simply by considering only nodes with large weights.

As an example, Figure 6 shows the function call graph of *Android.Trojan.FakeInst.AS*, a sample from the *FakeInstaller* malware family. Note, that disconnected nodes are not displayed to obtain a cleaner presentation. By shading each node in accordance with the sum of the weights $\hat{w}_i$ of the neighborhoods it belongs to, this allows a visual map to be constructed. This map is suitable to guide the analyst during the examination of a sample classified as malicious as it points directly to function neighborhoods considered typical for malware by the classifier.

Figure 7 shows a more concise example, the complete call graph of *Android:RuFraud-C* again from the *FakeInstaller* family. Samples from this malware family are wide spread and are often repackaged versions of popular applications. Malware authors hide the malicious code in these modified packages and upload them to third-party markets. Once a user installs the application, it will send SMS messages to expensive premium services owned by the authors. This specific sample, an SMS Trojan, first determines the Mobile Country Code of the device and, depending on the result, sends an SMS message to a premium number.

The depicted call graph of this sample includes in every node the class name, the function name and the final weight assigned to each individual node. It can be observed how the learned model assigns a high weight to the neighborhood of the `sendSms` function of the `FileloaderActivity` class. This specific function is called by the malware to send multiple SMS messages to different premium numbers. The numbers are stored in the `SmsClass` class and accessed using the `getSms` function. The `sendSms` function, which has been tagged with the highest weight, is therefore identified as the most insidious element in the sample.

We found similar code structures in other members of the FakeInstaller family and although these methods are often renamed by the malware authors to obfuscate their malicious intent, our method is still able to correctly identify them as malicious and assign a high weight to their neighborhoods.

4. LIMITATIONS

By analyzing the global structure of Android applications, our method is resilient towards typical local obfuscation techniques, such as instruction reordering, branch inversion or the renaming of packages and identifiers. However, as

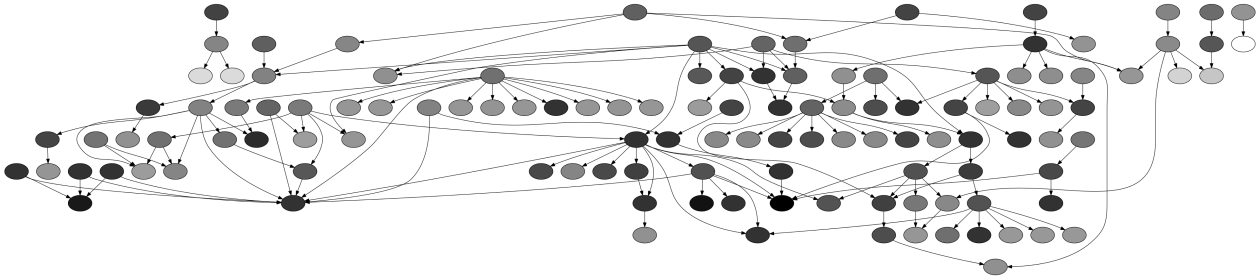


Figure 6: Subgraph from the function call graph of "Android.Trojan.FakeInst.AS" from the malware family FakeInstaller. Dark shading of nodes indicate malicious structures identified by the SVM.

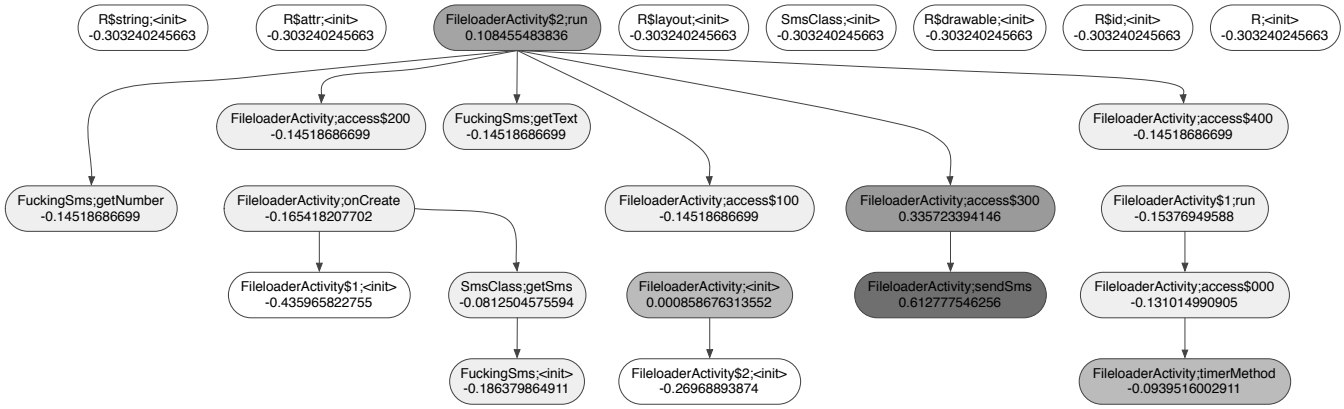


Figure 7: Complete function call graph of "Android:RuFraud-C" from the malware family FakeInstaller. Dark shading of nodes indicate malicious structures identified by the SVM.

a purely static method, it suffers from the inherent limitations of static code analysis. In particular, the construction of static call graphs is undecidable and thus the function call graphs processed by our method are typically over-approximations. In principle, this works towards the attacker's advantage, as the call graph can be obfuscated by adding unreachable calls. Moreover, function inlining can be used to hide the graph structure. While in the extreme case, this allows for the creation of malware with only a single function, Android's event-driven programming model usually prohibits this in practice.

Attackers may also target the decompilation process to evade detection by our method. For example, invalid but unreachable bytecode sequences ("junk code") can be deliberately inserted to hinder successful decompilation. Moreover, bytecode unpacked and loaded at runtime cannot be processed by the decompiler and thus can only be considered if our method is coupled with dynamic analysis techniques. Finally, native code is currently not analyzed, however, this is not a fundamental limitation, as our method can be adapted to process native code by adding a suitable decompilation stage and a corresponding labeling.

While these obfuscation techniques may be employed by malware targeting desktop computers, they are rarely encountered in current Android malware so far. This hypothesis is supported by the high detection performance observed in the empirical evaluation (see Section 3).

5. RELATED WORK

The analysis of malicious code and its structure have been a vivid area of security research in the last years. In the following, we first discuss related work on structural code comparison in general and then proceed to present approaches specifically designed for static detection of Android malware.

Structural Comparison of Code.

Determining similar program code is an important problem encountered in several areas of security research, including the detection of malware [1, 21, 27, 36], software plagiarism [28, 30] and vulnerabilities [9, 40]. To this end, several methods to assess the structural similarity of code have been proposed. For example, Kruegel et al. [27] as well as Cesare and Xiang [6] present methods for polymorphic malware detection based on the comparison of control flow graphs. In particular, Kruegel et al. perform graph labeling to preserve instruction level information in a similar manner as performed by our method. Unfortunately, both approaches are based on sets of control flow graphs and thus ignore the composition of functions entirely. We address this shortcoming by taking into account the function call graph.

Other researchers have also recognized function call graphs as a robust representation for code comparison. For example, Hu et al. [21] as well as Shang et al. [36] define similarity metrics for function call graphs, however, without consider-

ing the use of supervised learning techniques to automatically detect malware. The related problem of clustering known malware into families has been considered by Kinable and Kostakis [24], who use approximations to graph edit-distances to cluster malware by call graph similarity. Efficiency is however, not a primary concern in this setting, whereas it is vital in malware detection, a problem we address using an efficient linear time mapping.

Kernel functions for structured data have been pioneered by Haussler [19] and have been first applied over graphs by Kondor et al. [26]. Graph kernels have since then been applied mainly in bioinformations [e.g., 3, 37] and chemical informatics [e.g., 20, 33]. Unfortunately, the high computational effort involved has prohibited many applications in the past. Researchers have therefore focused on developing efficient approximations of graph kernels. An overview of these approaches is given in [4].

Regardless of these efforts, graph kernels have found little attention in malware detection to date. An exception is the work by Wagner et al. [38] and Anderson et al. [1] who use graph kernels to analyze execution traces obtained from dynamic analysis.

Static Detection of Android Malware.

The increasing popularity of mobile devices has made Android a prime target for attack. Consequently, a unique malware landscape has evolved in recent years as described in surveys by Felt et al. [15] and Zhou and Jiang [43]. To counter this emerging threat, methods based on both static code analysis [e.g., 10, 14, 31, 42] and dynamic code analysis [e.g., 11, 32, 34, 41, 44] have been proposed.

Most existing static approaches have focused on the Android permission system. For example, Enck et al. [10] analyze the permissions requested by applications at install time and manually formulate rules to detect malicious applications. Similarly, Peng et al. [31] analyze permissions to infer probabilistic models for malicious applications. Felt et al. [14] go one step further by correlating requested permissions with statically observable API calls to detect overprivileged applications. Unfortunately, these approaches are unable to detect malware which elevates its privileges by exploiting vulnerabilities [see 12, 29].

A second strain of research focuses on detecting variants of known malware families. For example, Zhou et al. [42] as well as Hanna et al. [17] employ feature hashing on byte code sequences to measure code similarity. Furthermore, Crussel et al. [7] present a method called DNADroid, which compares program dependence graphs. Finally, RiskRanker by Grace et al. [16] compares function call graphs. Unfortunately, RiskRanker requires source and sink functions commonly linked to malicious behavior to be specified manually, thus requiring constant manual adaption to changing trends in malware development. In contrast, the classifier learned by our method can be easily adapted by re-training on more recent malware data sets.

6. CONCLUSION AND FUTURE WORK

In this work, we have presented a learning-based method for the detection of malicious Android applications. Our method employs an explicit feature map inspired by the neighborhood hash graph kernel to represent applications based on their function call graphs. This representation is shown to be both, efficient and effective, for training an SVM that ultimately enables us to automatically identify Android malware with a detection rate of 89% with 1% false positives, corresponding to one false alarm in 100 installed applications on a smartphone.

As the vast majority of mobile malware targets the Android platform, this work focuses on Android malware detection. However, the method presented can be adapted to other platforms with minor changes, given that (a) function call graphs can be extracted and (b) instructions can be suitably categorized. Adapting the method to other platforms, including desktop systems, may thus be an interesting direction for future work. Moreover, combining existing classifiers based on contextual features with our structural detection approach seems promising.

7. ACKNOWLEDGEMENTS

This work has been supported by the German Federal Ministry of Education and Research (BMBF) under the project PROSEC (FKZ 01BY1145). We thank Felix Leder for fruitful discussions and Michael Spreitzenbarth for supplying us with the malware samples.

References

[1] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in Computer Virology*, 2011.

[2] A. Barla, F. Odone, and A. Verri. Histogram intersection kernel for image classification. In *Proc. of International Conference on Image Processing, ICIP*, volume 2, pages III–513–516, 2003.

[3] K. M. Borgwardt, C. S. Ong, S. Schönauer, S. V. N. Vishwanathan, A. J. Smola, and H.-P. Kriegel. Protein function prediction via graph kernels. *Bioinformatics*, 2005.

[4] K. Borhwardt. *Graph Kernels*. PhD thesis, University of Munich, 2007.

[5] S. Cesare and Y. Xiang. Classification of malware using structured control flow. In *Proc. of the Eighth Australasian Symposium on Parallel and Distributed Computing*, 2010.

[6] S. Cesare and Y. Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *Proc. of the International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, 2011.

[7] J. Crussell, C. Gibler, and H. Chen. Attack of the clones: Detecting cloned applications on android markets. In *Proc. of European Symposium on Research in Computer Security (ESORICS)*, 2012.

[8] A. Desnos. Androguard - Reverse engineering, Malware and goodware analysis of Android applications. `http://code.google.com/p/androguard/`, 2013.

[9] T. Dullien and R. Rolles. Graph-based comparison of executable objects, 2005.

[10] W. Enck, M. Ongtang, and P. D. McDaniel. On lightweight mobile phone application certification. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 235–245, 2009.

[11] W. Enck, P. Gilbert, B. gon Chun, L. P. Cox, J. Jung, P. McDaniel, and A. Sheth. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *Proc. of USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, pages 393–407, 2010.

[12] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A study of Android application security. In *Proc. of USENIX Security Symposium*, 2011.

[13] R.-E. Fan, K.-W. Chang, C.-J. Hsieh, X.-R. Wang, and C.-J. Lin. LIBLINEAR: A library for large linear classification. *Journal of Machine Learning Research (JMLR)*, 9:1871–1874, 2008.

[14] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android permissions demystified. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 627–638, 2011.

[15] A. P. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A survey of mobile malware in the wild. In *Proc. of ACM Worksgop on Security and Privacy in Smartphones and Mobile Devices (SPSM)*, pages 3–14, 2011.

[16] M. Grace, Y. Zhou, Q. Zhang, S. Zou, and X. Jiang. Riskranker: scalable and accurate zero-day android malware detection. In *Proc. of International Conference on Mobile Systems, Applications, and Services (MOBISYS)*, pages 281–294, 2012.

[17] S. Hanna, E. Wu, S. Li, C. Chen, D. Song, and L. Huang. Juxtapp: A scalable system for detecting code reuse among android applications. In *Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2012.

[18] H. Hao, V. Singh, and W. Du. On the effectiveness of api-level access control using bytecode rewriting in android. In *Proc. of the ACM SIGSAC symposium on Information, computer and communications security*, 2013.

[19] D. Haussler. Convolution kernels on discrete structures. Technical Report UCSC-CRL-99-10, UC Santa Cruz, July 1999.

[20] S. Hido and H. Kashima. A linear-time graph kernel. In *Proc. of International Conference on Data Mining (ICDM)*, pages 179–188, 2009.

[21] X. Hu, T.-c. Chiueh, and K. G. Shin. Large-scale malware indexing using function-call graphs. In *Proc. of the ACM conference on Computer and communications security*, 2009.

[22] X. Jiang. An evaluation of the application ("app") verification service in android 4.2, December 2012. `http://www.csc.ncsu.edu/faculty/jiang/appverify/`.

[23] Juniper Networks. Juniper networks third annual mobile threats report, 2013.

[24] J. Kinable and O. Kostakis. Malware classification based on call graph clustering. *Journal in Computer Virology*, 2011.

[25] C. Kolbitsch, P. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security Symposium*, 2009.

[26] R. I. Kondor and J. D. Lafferty. Diffusion kernels on graphs and other discrete input spaces. In *Proc. of the International Conference on Machine Learning*, 2002.

[27] C. Kruegel, E. Kirda, D. Mutz, W. Robertson, and G. Vigna. Polymorphic worm detection using structural information of executables. In *Recent Adances in Intrusion Detection (RAID)*, 2005.

[28] C. Liu, C. Chen, J. Han, and P. S. Yu. Gplag: detection of software plagiarism by program dependence graph analysis. In *Proc. of the ACM SIGKDD international conference on Knowledge discovery and data mining*, 2006.

[29] L. Lu, Z. Li, Z. Wu, W. Lee, and G. Jiang. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, 2012.

[30] J. Ming, M. Pan, and D. Gao. ibinhunt: Binary hunting with inter-procedural control flow. In *Information Security and Cryptology (ICISC)*, 2012.

[31] H. Peng, C. S. Gates, B. P. Sarma, N. Li, Y. Qi, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Using probabilistic generative models for ranking risks of android apps. In *Proc. of ACM Conference on Computer and Communications Security (CCS)*, pages 241–252, 2012.

[32] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos. Paranoid android: Versatile protection for smartphones. In *Proc. of Annual Computer Security Applications Conference (ACSAC)*, 2010.

[33] L. Ralaivola, S. J. Swamidass, H. Saigo, and P. Baldi. Graph kernels for chemical informatics. *Neural Networks*, 2005.

[34] V. Rastogi, Y. Chen, and W. Enck. Appsplayground: Automatic security analysis of smartphone applications. In *Proc. ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2013.

[35] B. P. Sarma, N. Li, C. Gates, R. Potharaju, C. Nita-Rotaru, and I. Molloy. Android permissions: a perspective combining risks and benefits. In *Proc. of ACM symposium on Access Control Models and Technologies (SACMAT)*, pages 13–22, 2012.

[36] S. Shang, N. Zheng, J. Xu, M. Xu, and H. Zhang. Detecting malware variants via function-call graph similarity. In *Proc. of the International Conference on Malicious and Unwanted Software (MALWARE)*, 2010.

[37] N. Shervashidze, S. Vishwanathan, T. Petri, K. Mehlhorn, and K. Borgwardt. Efficient Graphlet Kernels for Large Graph Comparison. In *Proc. of the International Conference on Artificial Intelligence and Statistics (AISTATS)*, 2009.

[38] C. Wagner, G. Wagener, R. State, and T. Engel. Malware analysis with graph kernels and support vector machines. In *International Conference on Malicious and Unwanted Software (MALWARE)*, 2009.

[39] D.-J. Wu, C.-H. Mao, T.-E. Wei, H.-M. Lee, and K.-P. Wu. Droidmat: Android malware detection through manifest and API calls tracing. In *Proc. of Asia Joint Conference on Information Security (Asia JCIS)*, pages 62–69, 2012.

[40] F. Yamaguchi, M. Lottmann, and K. Rieck. Generalized vulnerability extrapolation using abstract syntax trees. In *Proc. of 28th Annual Computer Security Applications Conference (ACSAC)*, pages 359–368, Dec. 2012.

[41] L.-K. Yan and H. Yin. Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis. In *Proc. of USENIX Security Symposium*, 2012.

[42] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proc. ACM Conference on Data and Application Security and Privacy (CODASPY)*, pages 317–326, 2012.

[43] Y. Zhou and X. Jiang. Dissecting android malware: Characterization and evolution. In *Proc. of IEEE Symposium on Security and Privacy*, pages 95–109, 2012.

[44] Y. Zhou, Z. Wang, W. Zhou, and X. Jiang. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *Proc. of Network and Distributed System Security Symposium (NDSS)*, 2012.