Detecting Android application malicious behaviors based on the analysis of control flows and data flows

Peter Zegzhda
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
zeg@ibks.ftk.spbstu.ru

Dmitry Zegzhda
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
dmitry.zegzhda@ibks.ftk.spbstu.ru

Evgeny Pavlenko
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
pavlenko@ibks.spbstu.ru

Andrew Dremov
Peter the Great
St. Petersburg
Polytechnic University, Russia,
St. Petersburg,
29, Politekhnicheskaya ul.,
+7(812)5527632
dremov@ibks.spbstu.ru

ABSTRACT
This paper explores the problem of identifying malicious code sections in applications for the Android. A method for analyzing Android applications is proposed, based on applying of static analysis using control flow graphs and data flow graphs. The paper formally describes the dependency relationships that are used to construct graphs, and also describes an algorithm that allows to identify malicious sections of code using the received graphs. The results of an experimental evaluation of the effectiveness of the proposed method are presented, demonstrated a high probability of detecting malicious portions of the code of Android applications.

CCS Concepts
Security and privacy→Systems security→Operating systems security→Mobile platform security

Keywords
Information security, Google Android, mobile security, malware, application analysis, control flow, data flow, Android application, Android security, malware detection

1. INTRODUCTION
The most popular operation system (OS) in the mobile technology market is Google Android. According to the latest statistics, about 85% of applications downloaded by users from various application stores contain malicious code. Attackers can also use legitimate applications to introduce malicious code into them, after which the applications are placed in special application stores under the guise of legitimate ones. Thus, the actual task is to determine the parts of the code of Android-applications, which contain malicious functions, the performance of which leads to violation of user security.

We propose to develop a system for analyzing the security of Android applications based on methods for static and dynamic analysis to collect information about application behavior and machine learning methods for decision whether it is malicious or not. This paper reviews the existing methods of static and dynamic analysis, that are two traditional approaches for detecting malicious applications [1].

SIN '17, October 13–15, 2017, Jaipur, IN, India

ACM ISBN 978-1-4503-5303-8/17/10…$15.00
https://doi.org/10.1145/3136825.3140583

The novelty of the research presented in this article is based on the fact that this approach allows us to identify areas for malicious code and remove them. In the future, these code fragments can be used to automatically generate signatures. This is particularly important, because all of the existing methods of the analysis application analyzes it as a whole, without separating parts of the code into legitimate and malicious. Moreover, as a result of their work is only a probabilistic assessment of whether application contains malicious functions, without concrete code fragments that implement malicious functionality.

The paper is organized in the following manner. Chapter 2 reviews the related works kept in the field of static and dynamic analysis methods. Chapter 3 describes the proposed approach, it contains of 3 main steps: splitting an Android application into sections of code, building graphs of control and data flows and identifying malicious sections of code. Working example is presented in Chapter 4, is considered Android application "krep.itmtd.ywtjexf-1.apk", which is disguised as the official client application of one of the banks and carries out a number of malicious actions. Experimental results, that demonstrate effectiveness of proposed approach, are represented in Chapter 5. Finally, the paper conclusion and the scope of the expected future works are presented in Chapter 6. Chapter 7 contains lists the cited and related works.

2. RELATED WORKS
The first malware search technology for devices running Google Android OS was borrowed from anti-virus protection methods and was based on the use of signatures - a sequence of bytes that uniquely identify a particular malicious program. Signature methods analyze the bytecode of Android applications, search for malicious functions, and then generate a signature. The authors of the application DroidAnalytics were the first who proposed to analyze byte-code on three levels for signatures generation: the level of methods, the level of classes and the level of the entire application [2].

The second generation of static analysis tools is equipped with analysis technologies that allow to build control flow and data flow graphs that represent a program execution model and a model of dependencies of some variables from others. In [3] the analysis of applications using graphs of control flows and data flows is considered in more detail, algorithms for constructing these graphs are presented, and the shortcomings of considered methods are determined. The authors of [4] and [5] used the structure of control flow graphs as the attributes required for the classification of applications.

Dynamic analysis methods, based on detection of anomalies, use application's monitoring results and extract all system calls, generated network traffic, logs and other parameters. The data

obtained is used to find the anomalous application impacts on the system. The authors of the CrowDroid application used the Strace utility to analyze system calls. When installed on the phone, the application collects the necessary information and sends it to the server for further analysis [6]. Other methods of dynamic analysis are based on using of a virtual environment in which the execution of the application is emulated, while monitoring of system calls and accesses to the OS, which provides information about malicious effects at the core level of the system.

Considered ways to analyze Android-applications could be easily bypassed by attackers. Typical ways to bypass static analysis are associated with the use of obfuscation and dynamic code generation. To bypass dynamic analysis, methods of detecting the environment are applied, in this environment dynamic analysis and program behavior modification are performed, depending on runtime system [7]. Currently, the most popular approach to analyzing the security of applications, including the Android operating system, is the use of machine learning methods, while input vector-signs are compiled from the results of static and dynamic analysis of applications [8].

To solve the above problem, a method is proposed to extract malicious sections of code whose properties can be used to automatically create signatures. The resulting signatures allow you to identify malicious code or remove malicious parts from the application, thereby eliminating the risk of malicious functions.

3. Approach

3.1 Splitting an Android application into sections of code

At present, static analysis methods based on control flow graphs and data streams are used to search for code sections. As a result of the construction of these graphs, the chains of method calls and dependencies between application components are defined. Using the obtained dependencies in conjunction with the attributes that affect the legitimacy of the application, you can identify malicious portions of the code. The disadvantage of the described method is the method of constructing these graphs. Most static analysis tools use standard algorithms to build graphs of control flows and data streams that do not take into account the characteristics of Android-applications.

Based on the results of the conducted studies, the task was to divide the application into sections of code, this task can be formulated as follows: $App = \{Class_1, Class_2, \dots, Class_n\}, n \in N$ – the set of all classes of the application. We introduce a family of non-empty sets $\{A_i^{Class}\}_{i \in I}$, where I – is a finite set of indices. A portion of the code A_i^{Class} is a subset of App if the following conditions are satisfied:

$$A_i^{Class} \cap A_j^{Class} = \emptyset, \forall i, j \in I: i \neq j, App = \{A_i^{Class}\}_{i \in I} \quad (1)$$

Thus, Android-application is considered as a collection of code sections, each of which consists of a unique set of classes [9]. The absence of links between classes means that the sections of the code are not connected in terms of management flows and data flows. If for any sections of code, you have the signs that negatively affect the security properties of applications, then such a piece of code is a malware code.

To solve this problem, it is proposed to use an approach based on the use of two types of graphs: class dependency graphs and method call graphs, the algorithms of construction of which take into account the features of Android applications considered earlier. For a formal definition of the graph of class dependencies, we introduce the following types of relations between classes:

1. Method calls dependencies. If the method mtd_1 of class $Class_1$ calls a method mtd_2 of another class $Class_2$, then there is a binary relation between the method calls between $Class_1$ and $Class_2$.

2. Dependencies of the data being processed. If the class $Class_1$ uses a variable var_2 of another class $Class_2$, then there is a binary relation of the data to be processed between $Class_1$ and $Class_2$.

3. Dependence of the intercomponent interaction. If a class $Class_1$ uses a class $Class_2$ through Explicit Intents, then there is a binary relation between the intercomponent interaction between $Class_1$ and $Class_2$.

4. For a formal definition of the method call graph, we introduce the relationship between the method calls. If method mtd_1 calls method mtd_2, then there is a binary relation between method calls between mtd_1 and mtd_2. It is worth noting that in this case the methods can be defined both in one class and in different classes, and the classes can be in different Java or Android API libraries.

On the basis of the dependence relations obtained, we introduce formal definitions of graphs. The application classes dependency graph is the directed graph $G_{Class} = \{V, E\}$, where V is the set of vertices, and E is the set of edges. Each vertex $n \in V$ represents the class of the Android application. The edge $e = (n_1, n_2) \in E$, directed from the vertex n_1 to the vertex n_2, denotes one of the three types of dependency relations between classes. Then sections of the code on the resulting classes dependency graph G will be called subsets of classes satisfying the following properties:

Relationships between classes of one section of code form an oriented subgraph, and $\forall Class_j \in A_m^{Class}\ \exists Class_i \in A_m^{Class}: \exists\, \vec{p} = (Class_i = C_0, C_1, \dots, C_k = Class_j)$, where A_m^{Class} – section of code, $\vec{p}$ – path connecting two classes $Class_i$ and $Class_j$.

There is no relationship of dependency between any two classes from different parts of the code. Formal entry: $\forall (Class_i, Class_j), Class_i \in A_i^{Class}, Class_j \in A_j^{Class}\ \exists!\, \vec{p} = (Class_i = C_0, C_1, \dots, C_k = Class_j)$, where A_i^{Class}, A_j^{Class} – sections of code, $\vec{p}$ – path connecting two classes $Class_i$ and $Class_j$.

The graph of method calls is a directed graph $G_{mtd} = \{V, E\}$, where V is the set of vertices, and E is the set of edges. Each vertex $n \in V$ represents the Android application method. The edge $e = (n_1, n_2) \in E$, directed from the vertex n_1 to the vertex n_2, denotes the relation of the dependence of the method calls.

3.2 Identify malicious sections of code

To determine sections of the application code, it is necessary to analyze the reachability of vertices in the class dependency graph. An adjacency matrix of an oriented graph $G = \{V, E\}$ is a binary matrix A whose unit elements describe all the arcs in an oriented graph-path of length equal to one.

To search for all arcs of length k, it is necessary to find a composition of relations of the set E with itself. By definition, the matrix of compositions of relations is a matrix of size $n \times n$, where n is the number of vertices of the graph. To analyze all existing graph paths, it is necessary to find the reachability matrix.

Based on the obtained D matrix, for each C_k class, a set of achievable R_k classes is defined, that is, classes to which there are paths from the C_k class. To find code sections, it is necessary to perform a number of transformations over the reachability matrix.

As a result of transformations of the attainability matrix, a matrix D' is obtained, the unit elements in each row of which correspond to classes belonging to the application code field. The number of sections of the code is equal to the number of rows of the resulting matrix D'.

The final step to determine the malicious portions of the code is to determine the correspondence between the graphs. To do this, the resulting sections of the code (subgraphs in the graphs of class dependencies) are one-to-one correspondence with the set of method call graphs. We denote this operation by projection and give it a formal definition.

The projection operation is the mapping of a section of code into a set of method call graphs. Formally: $Prj_{A_i}: A_i \to G_m = \{G_m^{C_1}, G_m^{C_2}, \dots\}$, where A_i is the code section in the class dependency graph, and G_m is the set of method call graphs. Projecting is performed on the basis of the tables of the correspondence of classes and the methods constructed when searching for the relationship of methods. After determining the correspondence, each section of the code is put in correspondence with a set of features obtained during the analysis of the corresponding set of method call graphs.

For the experimental evaluation of the possibility of using these characteristics to detect malicious code sections, an analysis was made of 1000 instances of malicious applications received from various resources (open databases and repositories of the GitHub service). Also, 1,000 copies of applications from the official Google Play store were downloaded. The result of our experiments, all the signs can be divided into three groups: in the first group obtained ranges of values for legitimate and malicious applications do not overlap, in the second group obtained ranges of values for legitimate and malicious applications are overlapped, but have differences not less than 0.1 up or down, in the third group obtained ranges of values for legitimate and malicious applications are virtually identical.Thus, the criteria for determining sections of the code are the characteristics of the first two groups, that presented in Table 1.

Table 1: Criteria for identifying malicious code sections

Group	Criterion	The range of possible values for a legitimate application	The range of possible values for a malicious application	The condition of satisfaction
I	$R_4^{D_{rate}}$	[0.174;0.531]	[0.621;1]	Values in one of these ranges
	$U_2^{V_{rate}}$	[0.784;0.889]	[0.921;1]	
	$U_3^{I_{rate}}$	[0.078;0.113]	[0.139;0.256]	
	$M_1^{I_{rate}}$	[0.431;0.557]	[0.632;0.881]	
	$M_5^{Def_{rate}}$	[0.032;0.091]	[0.185;0.298]	
II	$C_2^{S_{rate}}$	[0;0.298]	[0;0.667]	Values in one of these ranges and does not fall into the intersection area
	$C_3^{B_{rate}}$	[0;0.329]	[0;0,876]	
	$R_3^{L_{rate}}$	[0.223;0.785]	[0;0.434]	
	$R_6^{F_{rate}^{max}}$	[0.419;0.631]	[0.271;0.443]	
	$U_1^{M_{rate}}$	[0.098;0.226]	[0.003;0.183]	
	$M_3^{N_{rate}}$	[0.239;0.421]	[0.356;0.672]	
	$M_6^{S_{rate}}$	[0.176;0.359]	[0.231;0.821]	

Here, $C_2^{S_{rate}}$ is a service utilization rate, $C_3^{B_{rate}}$ - utilization ratio of broadcast message receivers, $R_3^{L_{rate}}$ - coefficient of using secure permissions, $R_4^{D_{rate}}$ - coefficient of using danger permissions, $R_6^{F_{rate}^{max}}$ - cover ratio of signal types for the receiver of broadcasting events with the maximum number of filters, $U_1^{M_{rate}}$ - coefficient of using user activity tracking methods, $U_2^{V_{rate}}$ - coefficient of use of the most significant methods of tracking user actions, $U_3^{I_{rate}}$ - coefficient of workload of methods for tracking user actions, $M_1^{I_{rate}}$ - utilization factor of the device information collecting methods, $M_3^{N_{rate}}$ - coefficient of use of methods of access to communication channels, $M_5^{Def_{rate}}$ - coefficient of use of methods of protection against detection, $M_6^{S_{rate}}$ - coefficient of using system methods.

The criteria, the values of which fall within the range of malicious applications, will be called decisive. As a result, the final evaluation of the malicious portions of the code $V_{malvare}$ is calculated using formula (2).

$$V_{malvare} = \frac{\alpha_1 F_{malware}^{I} + \alpha_2 F_{malware}^{II}}{\alpha_3 F_{valid}^{I} + \alpha_4 F_{valid}^{II}} \quad (2)$$

where F_{valid}^{I} – a quantity of criteria of the first group, for which is observed feasibility condition, F_{valid}^{II} – a quantity of criteria of the second group, for which is observed feasibility condition, $F_{malware}^{I}$ – a quantity of characteristic criteria of the first group, $F_{malware}^{II}$ – a quantity of characteristic criteria of the second group, for which is observed feasibility condition, $\alpha_1, \alpha_2, \alpha_3, \alpha_4$ – the weight coefficients. The values of weight coefficients were obtained experimentally: $\alpha_1 = 1.98, \alpha_2 = 0.89, \alpha_3 = 0.95, \alpha_4 = 0.48$. Then with the fulfillment of condition $V_{malvare} > 1$ the section of the code is considered as malicious.

4. WORKING EXAMPLE

For a more detailed description of the proposed approach, in Figure 1 is represented one of the graphs of the dependence of the classes obtained during the analysis of the Android application "krep.itmtd.ywtjexf-1.apk" is presented, which is disguised as the official client application of one of the banks and carries out a number of malicious actions, For example, the theft of credentials from online banking.

As a result of transformations of the attainability matrix, a two-row matrix D' was obtained, presented below. Thus, the class dependency graph under consideration consists of two sections of code: $A_1 = \{Class_1, Class_2, Class_3, Class_4\}$ and $A_2 = \{Class_5, Class_6, Class_7, Class_8, Class_9\}$.

$$D' = \begin{pmatrix} 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 \end{pmatrix}$$

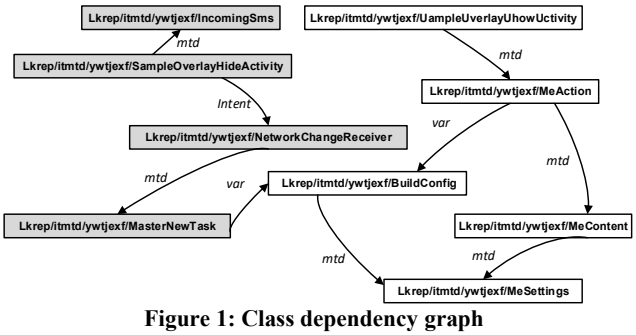

Figure 1: Class dependency graph

For the resulting sections of the code, operations were performed for projecting onto the set of method call graphs, on the basis of which characteristics for section of code were obtained. The criteria values are shown in Table 2.

Table 2: Criteria for the received code sections

Criterion	Section of code		Criterion	Section of code	
	A_1	A_2		A_1	A_2
R_4^{Drate}	0,854	0,478	C_3^{Brate}	0,5	0
U_2^{Vrate}	0,835	0,796	R_3^{Lrate}	0,357	0,671
U_3^{Irate}	0,231	0,101	$R_6^{Fmax}_{rate}$	0,311	0,292
M_1^{Irate}	0,786	0,823	U_1^{Mrate}	0,032	0,219
$M_5^{Defrate}$	0,072	0,054	M_3^{Nrate}	0,632	0,256
C_2^{Srate}	0	0	M_6^{Srate}	0,181	0,663

The results are $V_{malvare}$ values for two sections of code (A_1 and A_2). For A_1 $V_{malvare} = 1{,}361$; for A_2 $V_{malvare} = 0{,}581$. Results showed that malicious functions are contained in the code section of the A_1. To perform additional testing, these classes were decompiled, after which it was established that they contain the logic of the malicious application, which allows to intercept SMS messages from the bank with confirmation codes of operations and create new tasks for manipulating the user's bank account.

5. EXPERIMENTAL FINDINGS

To automate the analysis of Android applications, a system was created that performs the construction of class dependency graphs and method calls, extracting code sections on the derived class dependence graphs, calculating characteristic values, and assessing the legitimacy of code sections.

To assess the quality of the analysis, an additional study of 1000 applications (not previously used in this work) was conducted, of which 475 are malicious. The share of truly recognized malicious Android-applications is 91%, the share of truly recognized samples of secure Android-applications is 93%. The graph of the error curve and its quantitative interpretation of the AUC are shown in Figure 2.

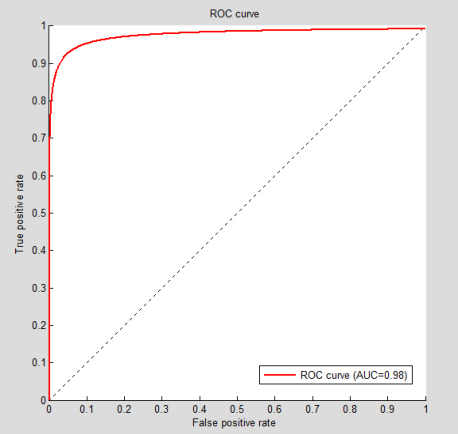

Figure 2: ROC-curve of the developed approach

6. CONCLUSION

As a result of the work, methods for detecting malicious code sections are examined, methods for determining code sections in applications are explored, methods for searching code sections and their subsequent analysis based on the construction of graphs of control flows and data flows are proposed. On the basis of the research, signs of application legitimacy have been obtained, allowing to determine whether the code sections contain malicious functions. Also, software was developed to analyze Android-applications in the manner described.

In future studies, it is planned to classify code sections using machine learning methods for the most accurate division into malicious and legitimate, and perform an analysis of the effectiveness of the algorithms used.

7. REFERENCES

[1] Peter D. Zegzhda, Semen S. Kort and Valery G. Chernenko. 2014. To the issue of security analysis of programs in the app stores of mobile devices. *Information Security Problems. Computer Systems,* 1 (Jan. 2014), 52-58.

[2] Min Zheng, Mingshen Sun, and John C. S. Lui. 2013. Droid Analytics: A Signature Based Analytic System to Collect, Extract, Analyze and Associate Android Malware. *In Proceedings of the 2013 12th IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TRUSTCOM '13).* IEEE Computer Society, Washington, DC, USA, 163-171. DOI=http://dx.doi.org/10.1109/TrustCom.2013.25.

[3] Li Lia, Tegawende F. Bissyand, Mike Papadakisa, Siegfried Rasthoferb, Alexandre Bartela, Damien Octeauc, Jacques Kleina and Yves Le Traon. 2016. *Static Analysis of Android Apps: A Systematic Literature Review.* Technical Report. Interdisciplinary Centre for Security, Reliability and Trust (SnT), University of Luxembourg, Luxembourg, Fraunhofer SIT, Darmstadt, Germany, University of Wisconsin and Pennsylvania State University.

[4] Hugo Gascon, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2013. Structural detection of android malware using embedded call graphs. *In Proceedings of the 2013 ACM workshop on Artificial intelligence and security (AISec '13). ACM, New York, NY, USA, 45-54.* DOI:http://dx.doi.org/10.1145/2517312.2517315.

[5] Mu Zhang, Yue Duan, Heng Yin, and Zhiruo Zhao. 2014. Semantics-Aware Android Malware Classification Using Weighted Contextual API Dependency Graphs. *In Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14).* ACM, New York, NY, USA, 1105-1116. DOI: http://dx.doi.org/10.1145/2660267.2660359.

[6] Iker Burguera, Urko Zurutuza, and Simin Nadjm-Tehrani. 2011. Crowdroid: behavior-based malware detection system for Android. *In Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices (SPSM '11).* ACM, New York, NY, USA, 15-26. DOI:http://dx.doi.org/10.1145/2046614.2046619.

[7] Thanasis Petsas, Giannis Voyatzis, Elias Athanasopoulos, Michalis Polychronakis, and Sotiris Ioannidis. 2014. Rage against the virtual machine: hindering dynamic analysis of Android malware. *In Proceedings of the Seventh European Workshop on System Security (EuroSec '14).* ACM, New York, NY, USA, , Article 5 , 6 pages. DOI:http://dx.doi.org/10.1145/2592791.2592796.

[8] Evgeny Y. Pavlenko, Anastacia V. Yarmak, Dmitry A. Moskvin. 2016. The application of clustering techniques to analyze the security of Android applications. *Information Security Problems. Computer Systems, 3 (July*, 2016), 119-126.

[9] Evgeny Y. Pavlenko, Andrew S. Dremov. 2017. Research of features of malicious Android applications. *Information Security Problems. Computer Systems, 2* (April, 2017), 99-108.