Leveraging Interprocess Communication Activity for Characterizing Android Software

Master’s Thesis

Stamatis Volanis

February 2014
Heraklion, Greece
Leveraging Interprocess Communication Activity for
Characterizing Android Software

Thesis submitted by
Stamatis Volanis
in partial fulfillment of the requirements for the
Master of Science degree in Computer Science

THESIS APPROVAL

Author: ____________________________________________
Stamatis Volanis

Committee approvals: ___________________________________
Evangelos P. Markatos
Professor, Thesis Supervisor

_______________________________
Dimitris Plexousakis
Professor

_______________________________
Sotiris Ioannidis
Principal Researcher

Departmental approval: ___________________________________
Angelos Bilas
Professor, Chairman of Graduate Studies

Heraklion, February 2014
Abstract

Smartphones are used by millions of users, while the mobile markets are being flooded with new software every day. Recent studies attempt to estimate the amount of illegitimate software for Android – one of the two most popular mobile architectures – with insufficient results. Unfortunately, there is Android malware out there, which seeks to compromise or take advantage of end-users. Malware performs malicious activities, without the user knowing, such as exfiltrating sensitive information (e.g. the user’s address book) or stealing money (e.g. forcing a mobile phone to call premium numbers). The research community has identified the threat and has proposed many static-based techniques for malware identification. While this is a step forward there are difficulties in handling code obfuscation or native code embedded in proprietary libraries.

In this work, we observe that Android is service oriented, that is, applications exchange Interprocess Communication (IPC) messages for accessing the system’s resources. For example, an application sends an SMS by making an IPC call to the telephony service. We argue that the IPC traffic, which is sent and received by a particular Android application can be useful enough for creating an accurate profile of the high-level actions performed by the under analysis application. We create a system that passively monitors all IPC activity exports application profiles based solely on that information. We analyze known malware and legitimate applications, and store their profiles in a library. We finally use the library to classify unknown software. Our classifier successfully distinguishes legitimate applications from malware with low false positive and false negative rates. However, we stress that our main goal in this work is to develop a system that assists the security analyst, rather than creating a purely unsupervised detector.

Apart from malware identification, our system can be also used for generic application profiling and data tracking. For example, we can passively identify premium numbers or address book information in IPC messages. Finally, we can graphically visualize all collected IPC activity in application graphlets; graphs depicting how an Android application is communicating with other applications and services. In this way, our system can be utilized for discovering colluding applications, which try exfiltrate sensitive
information by evading Android’s permission model by permission-sharing among many collaborating applications.
Ανάλυση της επικοινωνίας μεταξύ διεργασιών για το χαρακτηρισμό λογισμικού σε λειτουργικό σύστημα Android.

Σταμάτης Βολάνης

Μεταπτυχιακή Εργασία

Τμήμα Επιστήμης Υπολογιστών
Πανεπιστήμιο Κρήτης

Περίληψη

Τα 'έξυπνα' κινητά τηλέφωνα (smartphones) χρησιμοποιούνται σήμερα απο εκατομμύρια χρήστες, ενώ καθημερινά προστίθενται νέες εφαρμογές στα mobile markets. Πρόσφατες μελέτες προσπαθούν να εκτιμήσουν τον αριθμό κακόβουλου λογισμικού στο λειτουργικό σύστημα κινητών συσκευών Android, με αμφιλεγόμενα αποτελέσματα. Δυστυχώς υπάρχει κακόβουλο λογισμικό για Android, το οποίο προσπαθεί είτε να υποκλέσει στοιχεία (π.χ. τις επαφές του χρήστη), είτε να υποκλέσει χρήματα (π.χ. αποστολή sms σε αριθμούς υψηλής χρέωσης), είτε να παρουσιάζει διαφημίσεις επιθετικά στο χρήστη. Η ερευνητική κοινότητα έχει αναγνωρίσει το πρόβλημα και έχει προτείνει τεχνικές στατικής ή δυναμικής ανάλυσης για αναγνώριση κακόβουλου λογισμικού.

Στη συγκεκριμένη μεταπτυχιακή εργασία, παρατηρούμε ότι το Android είναι προσανατολισμένο στις υπηρεσίες, επομένως οι εφαρμογές ανταλλάσσουν μήνυμα μεταξύ τους αλλά και με το σύστημα μέσα απο ένα κανάλι επικοινωνίας (Interprocess Communication - IPC). Για παράδειγμα, μια εφαρμογή αποστέλλει ένα SMS κάνοντας μια IPC κλήση στην υπηρεσία τηλεφωνίας του λειτουργικού συστήματος. Εξετάζουμε ότι η συγκεκριμένη κίνηση, η οποία αφορά αποστολή και λήψη μιας συγκεκριμένης εφαρμογής ύπο εξέταση, είναι αφετέρου όταν δημιουργήθηκε αυτό αρκετές προφίλ από τις ενέργειες της. Δημιουργούμε ένα σύστημα το οποίο παρακολουθεί παθητικά όλη την IPC επικοινωνία και δημιουργεί προφίλ εφαρμογών βασιζόμενο μόνο σε αυτή την πληροφορία. Έπειτα, αναλύουμε γνωστές κακόβουλες αλλά και νόμιμες εφαρμογές και αποθηκεύουμε το προφίλ τους σε μια βιβλιοθήκη. Τελικά χρησιμοποιούμε τη βιβλιοθήκη για να κατατάξουμε όγνοστο λογισμικό. Το σύστημα μας διαχείρισε επιτυχώς νόμιμες εφαρμογές από κακόβουλο λογισμικό με χαμηλά
ψευδώς θετικά και ψευδώς αρνητικά ποσοστά. Ο κύριος στόχος της εργασίας είναι να αποτελεί ένα βοηθητικό εργαλείο για την ανάλυση λογισμικού.

Επόπτης Μεταπτυχιακής Εργασίας: Ευάγγελος Π. Μαρκάτος
The research outlined in this thesis has been conducted in the Distributed Computing Systems Laboratory (DCS), which is hosted in the Institute of Computer Science (ICS) of Foundation for Research and Technology - Hellas (FORTH), N. Plastira 100 Vassilika Vouton, GR-700 13 Heraklion, Crete, Greece.
Acknowledgments

First of all, I would like to extend my sincerest thanks and appreciation to my supervisor Prof. Evangelos Markatos and my co-supervisor Dr. Sotiris Ioannidis, for guiding me through my graduate studies and providing me their continuous support.

I also feel grateful to Dr. Elias Athanasopoulos for his valuable contributions in this work.

I would also like to express my best thanks to my friends and colleagues Iasonas Polakis, Eleni Gessiou, Giorgos Chinis and Kostas Plelis for working together in various projects.

Special thanks to all DCS members (Distributed Computing Systems) at FORTH-ICS, Lazaros Koromilas, Antonis Papadogiannakis, Giorgos Vasiliadis, Christos Papachristos, Nikos Tsikoudis, Panagiotis Papadopoulos, Antonis Krithinakis, Melitini Christodoulaki, Zaharias Tzermias, Thanasis Petsas, Symeon Meichanetzoglou, Panagiotis Ilia, Vaggelis Ladakis, Laertis Louitsis, Despoina Antonakaki, Mixalis Athanasakis, Mixalis Diamantaris, Giannis Voyatzis and all others, as well as non-DCS members Giorgos Salonstros and Panagiotis Garefalakis for providing a pleasant working environment and for being friends.

I would like to thank my friends Nikos, Manolis, Manolis Maria, Margarita, Manos, Katerina, Katerina, Panagiotis for their support and understanding me for being a stranger some times.

Finally, special recognition and love goes to my family. My parents Manolis and Evaggelia and my siblings George and Elli, for their support and encouragement. My beloved wife Eva, for always being supportive and understanding and my little daughter who every night welcomed me with a big smile.
Στους γονείς μου, Μανώλη και Ευαγγελία,
στα αδέρφια μου, Γιώργο και Έλλη,
στη σύζυγο μου, Εύα και την κορούλα μου.

To my parents, Manoli and Evangelia,
my siblings, George and Elli,
my wife, Eva and my little daughter.
# Contents

1 Introduction ........................................... 1  
   1.1 Android internals and proposal ..................... 2  
   1.2 Microscopic vs. Macroscopic ....................... 3  
   1.3 Why IPC ............................................ 4  
   1.4 Requirements ....................................... 4  
   1.5 Contributions ...................................... 5  
   1.6 Thesis Outline ..................................... 6  

2 Related Work ......................................... 7  
   2.1 Android IPC ......................................... 7  
   2.2 BLINC ................................................. 8  
   2.3 Android malware ..................................... 9  

3 Architecture .......................................... 11  
   3.1 Background ......................................... 11  
   3.2 BinderProfiler Overview ............................ 12  
   3.3 Implementation ..................................... 13  
   3.4 Application Graphlets .............................. 14  
   3.5 Blind and Deep Mode ............................... 18  

4 Malware Classification ............................... 19  
   4.1 Training ............................................. 19  
   4.2 Classification Algorithm ......................... 23  

5 Evaluation ............................................. 25  
   5.1 Experimental Setup ................................ 25  
   5.2 Accuracy ............................................ 26  
   5.3 Overhead ............................................ 28  
   5.4 Further Applications .............................. 30
5.4.1 Private Information Exfiltration 30
5.4.2 Real-time Detection 30
5.4.3 Colluding Applications 31

6 Deployment 35
6.1 Overview of operation 35
6.2 Future Work 38

7 Limitations and Considerations 41
7.1 Evading Detection 41
7.2 Performance Overhead 42
7.3 Message Parsing 42
7.4 Device Rooting 42

8 Conclusion 45
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.1</td>
<td>An example application graphlet of <code>com.keji.danti</code> malware.</td>
<td>15</td>
</tr>
<tr>
<td>3.2</td>
<td>Partial graphlet of RZStudio as generated by BinderProfiler.</td>
<td>15</td>
</tr>
<tr>
<td>3.3</td>
<td>IPC activity of each analyzed application, excluding events related to GUI.</td>
<td>16</td>
</tr>
<tr>
<td>3.4</td>
<td>IPC activity for three randomly selected malicious applications for a running period of 30 minutes.</td>
<td>17</td>
</tr>
<tr>
<td>4.1</td>
<td>Taxonomies expressing paths of different level.</td>
<td>22</td>
</tr>
<tr>
<td>4.2</td>
<td>Schematic overview of the systems’ architecture.</td>
<td>23</td>
</tr>
<tr>
<td>5.1</td>
<td>Receiver Operating Characteristic (ROC) curve for the two scoring functions.</td>
<td>27</td>
</tr>
<tr>
<td>5.2</td>
<td>Latency overhead of BinderProfiler.</td>
<td>28</td>
</tr>
<tr>
<td>5.3</td>
<td>CDF of the average throughputs needed to transmit a compressed IPC trace.</td>
<td>29</td>
</tr>
<tr>
<td>5.4</td>
<td>An example scenario of two colluding applications.</td>
<td>32</td>
</tr>
<tr>
<td>6.1</td>
<td>The home page screen of our experimental BinderProfiler prototype.</td>
<td>36</td>
</tr>
<tr>
<td>6.2</td>
<td>The page hosting the analysis results of our experimental BinderProfiler prototype.</td>
<td>37</td>
</tr>
<tr>
<td>6.3</td>
<td>Data gathering progress dialog of our experimental application prototype.</td>
<td>38</td>
</tr>
</tbody>
</table>
List of Tables

2.1 Comparison of Android Analysis tools. . . . . . . . . . . . . . . 8

4.1 Example payload entities used for creating taxonomies. . . . 20

5.1 Summary of the experimental setup. . . . . . . . . . . . . . . 26
Today, Android is a leading software platform for smartphones. Designed and developed by Google and others, it ships installed in a variety of mobile devices. It is estimated that Android users number in the hundreds of millions [7]. Unfortunately, this great success of the platform has been followed by a simultaneous rise of malware specifically targeting Android. According to F-Secure, the growth of Android malware is exponential [9]. We expect to see even more sophisticated Android malware in the near future mainly for two reasons. First, Android is a rapidly growing platform, since it ships with a diverse variety of devices, many of them low-cost. Second, unlike iOS, which is heavily monitored and controlled by Apple, installing software on Android from untrusted sources, such as malicious web sites [34], is trivial. In this work, we present a novel low-overhead technique for identifying malware for Android, which can be applied both off-line and real-time. Our system performs a short time-window analysis – in the order of a few minutes – for classifying an Android application, either as malware or as legitimate. This makes deployment as an on-line service ideal.
1.1 Android internals and proposal

There are many frameworks proposed for identifying malware in mobile devices, each of them exploiting particular methodologies. Some are based on static analysis [23, 21] or dynamic analysis [16, 40], on tainting [22, 33], on masking out sensitive information [28], on application repackaging [31], or on virtual machines [15]. In this work, we build a new system based on two observations. First, we observe that Android is a service-oriented operating system. Each application has to communicate with a service for accessing most of the capabilities of the device. This holds both for trivial tasks, such as requesting information associated with the hardware of the device, and complicated tasks, such as sending an SMS message to contacts listed in the device’s address book. In fact, many tasks, such as the aforementioned example with the SMS, require the application to communicate with multiple services. Second, a significant amount of malware performs legitimate operations, but towards malicious resources [46]. These include sending sensitive information to malicious servers, and calling or texting to premium numbers.

In this work, we propose a novel system for identifying Android malware based solely on Interprocess Communication (IPC) activity. We passively monitor all messages exchanged through Binder [13], the IPC mechanism of Android, and we construct application profiles. We visualize all collected profiles as graphlets [29], which encapsulate intrinsic signatures for distinguishing malware from legitimate applications. Our method has many advantages. First, it is fast compared to tainting, source analysis or utilizing virtual machines. We passively log all traffic produced by Binder, which can be done with low overheads. Second, we rely on high-level actions, i.e. requesting a particular service, and not on the actual code, which can be easily obfuscated or embedded in native libraries that are hard to decompile. Third, we do not need extra storage, except for storing the signatures, which take negligible space. Forth, we need a short time period for analyzing and classifying an application sample. This makes our method ideal for deploying it as a public service. Finally, we can identify applications, which collude for exporting sensitive information, since we can capture inter-application communication. Although we have not, yet, observed malicious applications colluding in the wild, we expect to see this trend in the near future. This is especially true for Android, as the permission model of the platform
suggestions that an over-privileged application might be suspicious and raise concerns. Researchers have presented an application model, where multiple applications can collude in order to export sensitive information from the device [38]. Our methodology can, in principle, detect such attempts.

Our technique utilizes graphlets. We were initially inspired by BLINC [29], which uses graphlets for classifying hosts in the network, without actually inspecting the payload of the network traffic. In this work, we adopt the term graphlet and apply it to a new context. We argue that Android applications can be accurately classified in taxonomies by simply observing all traffic produced by Binder. BLINC passively monitors all network traffic and blindly decides, based solely on packet headers. We have two modes of operation, one blind and one deep. In blind mode, we rely only on the communication of end-points. In deep mode we take into account the payload of all messages. Both modes have advantages and disadvantages. In blind mode, it is easy to quickly resolve application communities, and in deep mode it is easy to exploit offensive patterns that imply malicious activity or black-list suspicious resources (e.g. premium numbers).

1.2 Microscopic vs. Macroscopic

Many techniques for anomaly detection and malware identification are based on statically analyzing the source, or dynamically executing it in a virtual machine and recording all system calls. We argue that these techniques view the under analysis application microscopically. On the other hand, our technique attempts something different. We examine applications macroscopically. We capture the high-level actions that an application will inevitably perform. Both views, the microscopic and the macroscopic one, have their own merit. The first one collects low-level information, which, if correctly analyzed, can lead to an accurate classification. However, many times, the microscopic view is so complex, that is hard to understand the actual behavior of the application. On the other hand, the macroscopic view captures high-level actions that is hard to hide. For example, there is no way to send an SMS from a smartphone, without sending an IPC message to the SMS service. This message must also embed the recipient cell’s number and the SMS’s body.

In this work, we argue that we need to combine both views, microscopic and macroscopic, for delivering better security. Android is an example plat-
form for applying this combination.

1.3 Why IPC

All operating systems developed in the last decades have implemented a form of IPC. However, in Android IPC has some intrinsic properties that makes it ideal for characterizing applications. In contrast with other mobile platforms, such as iOS, Android builds its security model by isolating applications from the rest of the system. Each process runs on a sandbox under a unique user id. The process has a predefined permission set, which is declared in the application’s manifest file at install time. If the user agrees on this permission set, then all processes of this particular application can exercise all services associated with this permission set.

The fact that processes are isolated from the rest of the system elevates the role of IPC. Each process has to communicate with a service for performing basic operations. Any process acquiring information stored in the address book will produce IPC traffic, which reveals the intention of the process accessing sensitive information. This is not the case for iOS or Linux-based OSs for PCs, Windows, and Mac OS X.

To the best of our knowledge, Android is the first massively adopted platform that experiences these characteristics. This is the main reason we chose to create a framework for analyzing applications based solely on their IPC activity.

1.4 Requirements

We specify the following requirements for our system:

1. **Lightweight operation.** We seek of a low-cost analysis technique that can either be implemented as an on-line service or directly inside the operating system. Our system is based on passively monitoring IPC in Android, which is considered a low-computation task compared to tainting and incorporating special VMs.

2. **Hard to avoid detection.** Many source-based analysis systems suffer due to code obfuscation or native code relying in proprietary libraries, which is hard to decompile. We, on the other hand, examine high-level operations, exposed through IPC messaging. An attacker can evade detection by making the malware stealthier (i.e., decreasing the rate
of texting to premium numbers). However, this raises the bar for the attacker. We discuss such trade-offs in Chapter 7.

3. **Accuracy.** We ideally want a classification technique that does not suffer from high-rates of false positives and false negatives. Malware identification techniques have been recently criticized and the results of published methodologies were questioned by the research community [37]. We stress here that our system aims more at assisting the security analyst in the classification process, which will be potentially enhanced with other sources of information, rather than operating as an isolated, non-supervised detector.

For realizing these requirements, we perform the following. Our system is based on a training and a running phase. During training, we run a series of applications in the Android emulator for a short period of time (in the scale of minutes). We export all Binder traffic associated with the under analysis application and form its profile. In our training set we have already classified malicious applications [46], as well as legitimate ones. This phase can be carried out off-line. After all traffic is collected and all profiles are formed, we create a library with application graphlets, which encapsulate signatures that can potentially distinguish legitimate applications from malware. We develop a scoring system (see Chapter 4), which reveals patterns in IPC traffic that imply malicious behavior, based on the information stored in our graphlet library. Our primary goal is malware classification, but our engine can be easily extended for carrying out more complex tasks, such as protecting end-users from exfiltrating sensitive information. For example, we can create a black list of resources (e.g. a premium telephone number), and prohibit any communication or accessing on them.

### 1.5 Contributions

This work makes the following contributions:

1. We present BinderProfiler, a novel system that classifies Android applications based solely on observed traffic produced by IPC activity.

2. We introduce application graphlets for characterizing Android software. Application graphlets are graphs, which encapsulate the high-level behavior of an application.
3. We evaluate our system in terms of accuracy and performance.

4. We design an on-line public service, which utilizes BinderProfiler for profiling Android applications before installation.

1.6 Thesis Outline

This work is organized as follows. We review prior work in Chapter 2. In Chapter 3 we present the architecture of our system and how application graphlets are produced. In Chapter 4 we discuss how our system can classify Android applications in taxonomies, and, then, how malware is distinguished from legitimate applications. We evaluate our methodology in Chapter 5 and we discuss how our technique can be used for implementing further applications in Section 5.4. We discuss how we deployed our system as an on-line public service in Chapter 6 and various limitations of our proposal in Chapter 7. We conclude in Chapter 8.
In this chapter we review related work. Since, we have borrowed ideas from different fields of research, we have structure this chapter in many parts.

2.1 Android IPC

Our work is heavily based on Android’s IPC. The mechanism for providing communication facilities to Android applications and services has its roots in BeOS’ OpenBinder [10], which was developed by Be and Palm. Android’s Binder is considered an evolved version of OpenBinder, which now shares little with its predecessor. Researchers have studied the performance and robustness of Android’s Binder [13]. Perhaps, the first work that raised concerns about security implications in Android’s IPC is the article of Enck et al. in the S&P Magazine. Furthermore, in ComDroid [18] researchers have studied how malicious applications can compromise user’s privacy by looking at Binder traffic. Essentially, this work is very similar with ours, but with a different orientation. ComDroid attempts to detect application vulnerabilities that can be exploited through IPC, where BinderProfiler is using IPC traffic to characterize applications. IPC Inspection [26] addresses the problem of permission re-delegation, where multiple applications can collude for gaining more privileges. They studied this in both the mobile and
web platform and they propose a mechanism that adjusts permissions when a message is delivered. In this work we discussed many times about permission re-delegation. BinderProfiler can efficiently detect colluding applications, but it does not enforce any policies. Finally, Quire [20] proposes chaining of IPC messages and signed message delivery for ensuring trust between Android applications communicating with each other or with a remote endpoint.

### 2.2 BLINC

We are inspired and influenced by BLINC [29], a framework for classifying network hosts and traffic. BLINC was the first attempt for host classification in the network, without taking into account the payload of the traffic exchanged and the port numbers of the communicating end-points. It introduced the idea of graphlets, graphs depicting host communication. In fact, BLINC uses bipartite cliques to visualize hosts communities. Using a set of heuristics BLINC is able to produce graphlets that encapsulate signatures for a web server, a game server, a P2P applications, etc.. In this work we borrowed the term graphlet to depict applications communicating in Android using IPC. However, we do not use bipartite cliques, but graphs which are formed by applications sending IPC messages or receiving to other applications and services. We mainly borrowed the term from BLINC to stress this analogy. BLINC is able to characterize a host in the network with the least information – just the end-points –, and we try to characterize applications just by looking at the IPC traffic, without taking into account all

<table>
<thead>
<tr>
<th>Tool</th>
<th>Static Analysis</th>
<th>Dynamic Analysis</th>
<th>Taint tracking</th>
<th>IPC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Androguard</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Andrubis</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DroidBox</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>DroidScope</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>TaintDroid</td>
<td>✗</td>
<td>✓</td>
<td>✓</td>
<td>*</td>
</tr>
<tr>
<td>TraceDroid</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>CopperDroid</td>
<td>✗</td>
<td>✓</td>
<td>✗</td>
<td>✓</td>
</tr>
<tr>
<td>ForeSafe</td>
<td>✓</td>
<td>✓</td>
<td>✗</td>
<td>✗</td>
</tr>
</tbody>
</table>

Table 2.1: Comparison of Android Analysis tools.
2.3 Android malware

Android malware has received a lot of attention. In this work, we used the Android Malware Genome Project [46], which we consider as the state-of-the-art malware sample. Furthermore, there are many proposals for identifying malware in mobile devices. Ded [23] decompiles Dalvik programs to Java and use the many available Java tools for performing static analysis to the recovered code. They study more 21 millions of source code and they identify potential misuses by legitimate applications. A similar effort has been done in PiOS for iOS [21]. Androguard [1] provides static analysis for Android applications by disassembling and decompiling Dalvik bytecode and computing a similarity value for application repackaging detection. Source analysis is a well studied technique and has many advantages. However, many times, it is hard to analyze the code, which may be either obfuscated or native in a proprietary library. In this work, we argue that source analysis is a microscopic technique for identifying malicious activity, where our approach is macroscopic, since we analyze the high-level actions of an applications. We argue that there are many cases where hiding information at the microscopic level is feasible (obfuscating the code), but hard at the macroscopic level, where concrete actions must take place.

Tainting and dynamic analysis have been also explored for tracking information flow and the behaviour of applications in the mobile environment [22, 33, 16, 40, 17, 42] Andrubis [3], which is an extension to Anubis, provided the first dynamic analysis platform for Android applications as a public service. Droidscope [44] implements an instrumentation interface with taint tracking, API call and native instruction tracing plugins. Another dynamic analysis approach is Droidbox [6] which adds tracing code to core Android opensource framework libraries and also utilizes Taintdroid [22] for tainting analysis. Tracedroid [41] follows a similar approach. Taintdroid [22] is probably the most complete and mature tainting framework for Android at the time of writing. It provides a system-wide tainting system for Android with realistic overhead. However, it suffers from propagating the tainted information in native code, as well as from well known problems associated with tainting [39]. We do not consider BinderProfiler as a replacement of these techniques, but as a complementary tool. BinderProfiler can assist
the security analyst with high-level information, which can be eventually correlated with the fine-grained information flows received from tainting frameworks. Recent work on Android malware analysis also monitor IPC traffic in addition to VMI-based dynamic system call-centric analysis for reconstructing malware behaviors [36].

In Section 5.4 we discuss how BinderProfiler can be applied in real-time. We argue that we can outsource a significant amount of computation and storage requirements in the cloud. This is an approach inspired by Paranoid Android [33].

Finally, there are many research efforts for enhancing and optimizing the permission model of Android, which is fundamental for the security of the platform. Saint [32] enhances Android’s permission model with policies, which is more powerful than static permissions enforced at installation time. Saint policies can assist in trusted communication between applications and components. For a short example, consider an applications which is linked with an Ad framework. The application could outsource fake Ad clicks by hijacking the communication between itself and the component offering the Ads functionality. On the other hand, Kirin [24] attempts to resolve potentially dangerous combinations of permissions at install-time and warn the user. The authors also provide the implementation of a service offering application certification based on Kirin. Aurasium [43] enforces policies through user-level sandboxing. The authors automatically repackage Android applications with custom code, which is able to resolve offensive actions similar to the ones we identify in this work (e.g., calling or texting premium numbers). The great advantage of Aurasium is that it needs no system modifications, since applications are automatically extended to support the framework. However, Aurasium aims more at user protection, while BinderProfiler aims at application analysis. Table 2.1 summarize representative tools which focus on Android malware analysis. Our work uses dynamic analysis of IPC traffic in an attempt to train and classify Android applications.
In this chapter we present BinderProfiler. We begin with some background information about Android and IPC, and then we give a short overview of the system. We discuss how we implemented it for Android, we present application graphlets, and how we use them as descriptors for identifying malicious patterns in application execution. We close this chapter by presenting our system’s two modes of operation, namely blind and deep.

3.1 Background

Instead of using the traditional IPC techniques offered by the Linux Kernel, the Android project implemented Binder, based on OpenBinder [10]. Android models applications as a set of components with distinct roles, namely Activities, Services, Content Providers and Broadcast Receivers. Activities provide the UI to interact with the user. Services stay on the background and perform operations without engaging the user. Content Providers provide a consistent method to store and access data.

Android implements all IPC using Remote Procedure Calls (RPC). The most common technique for issuing an RPC in Android is through intents. An intent is a type of object designed to deliver messages across Android components. Intents are capable of determining the destination of the mes-
sage dynamically. The developer can set the criteria that are needed for the
delivery of the intent. Android, based on these criteria, dynamically resolves
and decides which is the destination of a particular intent.

Services requiring advanced RPC capabilities, for instance multithreading
support, have to create an AIDL specification. The Android Interface
Definition Language (AIDL) is the specification of the API that the service
wishes to expose to other components. Based on that specification, Android
generates the necessary proxy class used by the clients and a stub class used
by the service to implement RPC. Binder uses these two classes to transmit
method calls and parameters from the client process to the server process.
More specifically, Binder initiates a transaction from the client to the server
containing all the necessary information for the RPC in the payload. In
the Java model the transaction data is expressed as a Parcel object. A
parcel can contain Java primitives, objects or references to other interfaces
(IBinder) objects. All these have to be marshalled before being sent across
process boundaries. Another structure, heavily used in IPC, is bundles; a
special type of payload holding key/value pairs and it is designed for type-
safety and improved performance, it is used extensively by the applications
for convenience.

Besides Binder, which is implemented and accessed in Java, there is
middleware written in C++ that mediates the interaction between the Java
objects and the Binder kernel module. Finally, Binder includes a custom
kernel component that passes messages between processes. Binder follows
the “thread migration” model. That is, an IPC call between processes looks
as if the thread issuing the IPC has hopped over to the destination process
to execute the code there, and then hopped back with the result.

3.2 BinderProfiler Overview

BinderProfiler is based solely on traffic produced by Binder. Applications
that need extra resources, for example access to the video or to the SMS
functionality of a smartphone, produce messages towards the service that
provides the particular functionality. These messages, along with their re-
sponses, are delivered through Binder. Our system runs on a modified An-
droid kernel, which passively logs all Binder traffic. It then associates a
graph, which we call application graphlet, with each application. As we
show later in the work, graphlets can be used for characterizing an unknown
application. For example, a security analyst can be assisted in deciding whether an application may be considered offensive by searching for particular patterns in the monitored traffic.

The captured Binder traffic expresses high-level application activity. For example, an application that exfiltrates sensitive information, such as the IMEI or the address book, will eventually request this information by sending an IPC message to the System service, which will eventually be delivered through Binder. In the same fashion, an application that issues calls or SMSes towards premium numbers, will acquire the functionality by requesting, through Binder, the telephony or texting service. Our intuition suggests that Android malware aggressively performs such actions in short time-windows, which can be identified solely by monitoring the traffic produced by IPC calls.

3.3 Implementation

We implemented BinderProfiler on the Android emulator. It can be argued that malware can detect whether it is running on the emulator or on an actual device. However, we selected this approach for convenience, since it is easier to build and debug Android on the emulator than on an actual smartphone. Now that our system has been finished and tested, porting BinderProfiler on a device is trivial. We discuss this in Chapter 6.

We use Android emulator version 18.0 (build_id MASTER-306762) running Android 4.0.3 (API level 15). Unfortunately, the variety of different Android versions, makes developing a security testbed hard. Some of the malware we run were crashing or not running at all. We assume that this is because the particular malware was originally written for a different Android version. One can argue that this hardens the task of malware authors, as well. This is evident, since many families of malware exist, presumably translating the same application for different Android versions [46].

All applications that crashed or did not run are excluded from our experiments and analysis. In Chapter 5 we evaluate our system and we explicitly discuss the percentage of application we were not able to capture. Fortunately, our Android modifications can be easily ported to older Android versions. Porting our system to multiple Android versions for supporting a broader range of applications is left as future work.

To collect IPC traffic the application must perform some computation.
CHAPTER 3. ARCHITECTURE

Usually, Android applications incorporate a user interface and almost all actions are delivered through it. We expect that malware may perform malicious activities without the user doing anything special with the application. However, our system can be used more broadly for characterizing applications in general, not simply from a security perspective, and most importantly, our system needs a training phase (see Chapter 4), which incorporates both malware and legitimate applications as samples. We use Android Monkey [12], an automated UI exerciser for triggering functionality in each application. This tool has also been used for similar purposes in various papers [16]. Monkey will exercise application by applying random events by default, or predefined ones. Since some malicious activities may require specific user actions like filling a form and pressing submit button. This is a challenging issue, which also applies to dynamic analysis tools, even independent with Android. To overcome this issue, we modified AndroidViewClient [2] to fill out all available edited text components inside an application with either preset or pseudorandom data, in an attempt to fill out forms. For example, there is a preset which fills non-existent email addresses when identifying that the corresponding field requires email address. Furthermore, the script attempts to press all clickable UI components. It also preserves state, so if pressing a component the activity changes or the application exits, the application will restart and the procedure will be resumed skipping all previously completed actions.

3.4 Application Graphlets

The core idea of BinderProfiler is application graphlets. Initially, we were inspired from BLINC [29], a system for characterizing network hosts by inspecting headers of exchanged network flows. Since Android is heavily based on communicating services over IPC, we seek of a similar framework for characterizing Android applications. An application graphlet is the visual representation of an application interacting with other services over IPC calls. We depict such an example graphlet in Figure 3.1. We present the complete graph as well as a area zoomed in area. This graphlet visualizes the IPC activity of a known malware running on BinderProfiler for three minutes.

Two nodes are connected with each other if they exchange a message through Binder. The connection is directed; the direction is from the node
3.4. APPLICATION GRAPHLETS

Figure 3.1: An example application graphlet. At the top we depict the graphlet of the malware identified as com.keji.danti. At the bottom we zoom in a particular area of the whole graphlet. If two nodes are connected with a solid line then these nodes have exchanged a message we are not able to parse. Otherwise, the message is parsed and decomposed in printable entities. These entities are drawn as nodes between the communicating nodes and interconnected with dashed line. For example, com.keji.danti is communicating with com.android.phone through a message that encapsulates the com.android.internal.telephony service. Nodes, highlighted with red are part of a path that contributes in malware classification.

Figure 3.2: Partial graphlet of RZStudio as generated by BinderProfiler. Notice, that the fundamental actions that have been already documented [4] can be easily identified just by looking at the exported graphlet.
Figure 3.3: IPC activity of each analyzed application, excluding events related to GUI. Observe, that all applications generate tens of IPC events in the running period. Also, legitimate applications are more active.

in red color are nodes with high weights in the classification algorithm. We discuss this in more detail in Chapter 4 and 5.

We can make many important observations simply by inspecting this example application graphlet. First, the graphlet is of high complexity. Recall that we only run the malware for three minutes. The size of the graph suggests that IPC traffic generated by an application may be sufficient for profiling the application. Second, many actions performed by the application can be easily identified semantically. For example, notice how accessing the telephone functionality of the device is visualized through the interconnected nodes `com.keji.danti` and `com.android.internal.telephony`. In this work, we argue that the application graphlet expresses the higher-level semantic functionality of an application, which is hard to obfuscate or hide. For example, observe in Figure 3.2 a part of the application graphlet exported by RZStudio, a malware that has been analyzed by security experts. The results of the analysis [4] can be instantly identified in its application graphlet. Certainly, a malware can be stealthier, i.e., perform all actions
slowly, but this reduces the aggressiveness of the malware. We discuss this in Chapter 7.

To get a clearer picture for IPC activity, we plot all IPC events recorded for each analyzed application, excluding events related to GUI, in Figure 3.3. Observe, that all applications generate tens of IPC events in the running period. Running period is three minutes. Also, legitimate applications are more active. Recall, that all applications run in an emulator, which incorporates Android Monkey [12]. We speculate that some of the malicious applications are repackaged applications [45] with some of the originally functionality turned off. For selecting an adequate running period, we do the following: We randomly select three malicious applications and analyze them for a running period of 30 minutes. We depict IPC messages exchanged per minute in Figure 3.4. Observe, that most of the IPC activity takes places during the first few minutes. Thus, we select a three-minute analysis period for our further experiments.
3.5 Blind and Deep Mode

All communication through Binder involves the exchange of custom messages. Some of them are hard to parse. Even if we are currently able to parse some of them, applications can be modified to exchange encrypted messages, and thus, evade BinderProfiler. We were tempted to simply use the end-points of the communication, and not to take into account the payload of messages. We refer to this as blind mode. As expected, in blind mode a large training phase needs to be used. This has several disadvantages. First, it is difficult to collect a large amount of applications, compared to other application domains such as passive network monitor. Second, many Android processes delegate other services for performing the actual actions. These delegated services act as proxies (see for example system_server in Figure 3.1). This delegation hides the actual end-points. It may be possible to perform time correlation analysis for matching which messages belong to a set of two communicating end-points; we plan to explore this in future work.

Despite, blind mode appearing inefficient for characterizing Android software in practice, it can be effectively used for identifying colluding applications, even if these applications communicate with encrypted messages [38, 26]. We further discuss colluding applications and permission re-delegation in Section 5.4.

For the rest of the manuscript, we use deep mode, i.e., we parse all IPC messages. We export the printable parts and we isolate entities, that contribute significantly to the application graphlet.
BinderProfiler can be effectively used for producing the profile of an Android application in terms of interprocess communication. From a security perspective, this profiling can be leveraged for distinguishing malware from legitimate software. In this chapter, we discuss how BinderProfiler can be used for malware classification.

4.1 Training

BinderProfiler generates application graphlets. A graphlet depicts the esoteric high-level functions, expressed in the form of IPC communication, of a running application. Our intuition suggests that Android malware is characterized by the accumulation of certain subsequent activities (e.g., touching the address book, sending SMS messages or calling premium numbers), which inevitably will trigger IPC traffic. This intuition mainly stems from recent studies about malware behavior in Android [46]. Thus, we seek of a malware classification algorithm based solely on application graphlets or information that can be exported from them.

An application graphlet is the IPC profile of an application. It is questionable whether this information is enough for distinguishing malicious from legitimate behavior. The classification algorithm we present in this chapter
CHAPTER 4. MALWARE CLASSIFICATION

### Table 4.1: Example payload entities used for creating taxonomies. Each taxonomy is characterized by its depth, \( d \), describing the maximum amount of intermediate nodes, i.e., payload entities, which we take into account for connecting two individual nodes. A taxonomy with great depth is more descriptive, but less general for further capturing unknown malware. A taxonomy with less depth is more general and it fails to distinguish malware from legitimate applications.

<table>
<thead>
<tr>
<th>Depth</th>
<th>Nodes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>system_server, android.app.IActivityManager</td>
</tr>
<tr>
<td>2</td>
<td>com.android.phone, com.android.internal.telephony.ISms, 1066156686</td>
</tr>
<tr>
<td>3</td>
<td>system_server, android.app.IActivityManager, com.zft android.provider.Telephony.SMS_RECEIVED, android.permission.BROADCAST_SMS</td>
</tr>
<tr>
<td>full</td>
<td>system_server, android.app.IActivityManager, android.intent.action.VIEW, vnd.android-dir/mms-sms, sms_body, *, vnd.android-dir/mms-sms</td>
</tr>
</tbody>
</table>

is not perfect, but recall that we are tolerant in false positives mainly for two reasons. First, our system can be used in combination or as part of an existing system [3, 8]. Second, BinderProfiler acts more as a warning system assisting the security analyst, or even the end-user (see Chapter 6), in characterizing an application’s behavior, rather than as a sensitive detector which blindly decides whether an application is malware or not.

Application graphlets are graphs with nodes representing processes or payload entities. These graphs are formed by connecting individual nodes with each other when there is IPC between them. Comparing two such graphlets, in the general case, is non-trivial. Moreover, we are interested in particular legitimate actions, which if we encounter in groups, we consider them as offensive. For example, getting the address book and sending an SMS produces a graph, which can be hardly considered offensive. However, getting the address book, accessing the network, and sending a series of SMSs in a short time window, produces a graph that is unlikely to be associated with legitimate behavior, or at the very least should raise concerns about the goals of the application. Thus, we need to use partial information extracted from graphlets for deciding about the probability of a given application encapsulating malicious features. Ideally we want to accomplish that with
minimum overhead. Although, our system operates in off-line mode, we envision a service that will test multiple Android applications in parallel, and thus incur low computation overhead. This also makes our system suitable for inline run-time operation, which we further discuss in Section 5.4.

To summarize, we leverage BinderProfiler for malware classification in the following way. First, we produce the application graphlets of a series of malicious applications and legitimate ones. We use the Android Malware Genome Project as a malware source [46]. Collecting legitimate applications is a harder task, since there is no official set composed by provably non-malicious applications. We collect all legitimate applications from Slideme market [11], since Google prevents crawling the official Android market. Slideme is a popular unofficial market for Android software and we expect it incorporates security auditing technologies. Although, it has been demonstrated that such technologies can be bypassed [5], we believe that currently, malware authors have not reached such a level of sophistication. Nevertheless, we select the most popular ones for reducing the probability in getting malware, that has not yet been identified and reported. Each graphlet is produced by running the application for three minutes on a modified Android emulator running BinderProfiler. We selected a three-minutes time for analysis period, since, as we discussed in Chapter 3, we observed that the most of IPC activity occurs during the first few minutes of a running process (see Figure 3.4). The specification of the platform is exactly the same as the one analyzed in Chapter 3.

Second, we group all graphlets as benign or malicious. Now, recall how the graphlet is composed. Two connected nodes represent two processes that exchange an IPC message. All IPC messages encapsulate a payload. We parse the payload and export all entities that reflect an API call. Parsing an IPC message is not trivial as it has been already serialized and it possibly includes Java objects whose semantics we are not aware of. Thus, we extract only printable strings, because class names and other sensitive information, such as contacts or telephone numbers, are expressed in text. These entities are further nodes that interconnect the two initial nodes (see Figure 3.1). We use these entities for producing taxonomies based on the popularity of paths that occur more frequently in the malware set than the legitimate one. As seen in Table 4.1, each taxonomy is characterized by its depth, $d$, describing the maximum amount of intermediate nodes, i.e. payload
CHAPTER 4. MALWARE CLASSIFICATION

Figure 4.1: Taxonomies expressing paths of different level. We begin with the original graphlet (top and left). From left to right, the second graphlet depicts a taxonomy with depth 1. Only processes that receive a message are taken into account. The third graphlet depicts a taxonomy with depth 2, where processes that receive a message, as well as the first payload entity (usually the class name, which receives the message), are taken into account. In the same fashion, graphlets in the second row, from left to right, represent paths of depth 3 and 4, respectively.

An example of how taxonomies are extracted based on different path depth, \( d \), is shown in Figure 4.1, where we depict 4 different taxonomies, each one having depth from 1 to 4. We begin with the original graphlet (top and left). From left to right, the second graphlet depicts a taxonomy with depth 1. Only processes that receive a message are taken into account. The third graphlet depicts a taxonomy with depth 2, where processes that receive a message, as well as the first payload entity (usually the class name, which receives the message), are taken into account. In the same fashion, we depict two cases, where taxonomies represent paths of depth 3 and 4, respectively.

We use taxonomies for extracting weights. More precisely, we observe which graph paths are dominating in the malware set and which are rare in the legitimate set, and we assign each path with a positive weight. We also take into account the reverse behavior, \( i.e. \), paths that are dominating in the legitimate set and are rare in the malicious one. We assign each such
4.2 Classification Algorithm

We use two scoring functions for classifying unknown applications, one application frequency aware, denoted as $F_{af}(n_i) \leftarrow (w_i, f_i, d)$, and one non application frequency aware, denoted as $F_{naf}(n_i) \leftarrow (w_i, d)$. Notice, that each function depends on the collected weights from the training phase and from the taxonomy’s depth, $d$. For a particular depth, the analytic expressions of these scoring functions are the following:

$$F_{af}(n_i) = \sum_i \frac{w_i n_i}{1 - f_i},$$

$$F_{naf}(n_i) = \sum_i w_i n_i.$$
All weights are normalized to 1. The sum is over all paths composing a non-classified graphlet. If a path has no weight then we remove it from the sum (we implicitly assume a weight of zero). We use these functions to evaluate graphlets of applications that are not pre-classified. For each graphlet, the final score is an indicator of how similar it is with the graphlet with either a malware one or a legitimate app. The higher the score, the greater the probability of an application containing malicious functionality. In Chapter 5 we evaluate our classification algorithm in terms of accuracy.
In this chapter we evaluate the malware classification algorithm we presented in Chapter 4. We first give an overview of the experimental setup and then we present the accuracy of the classifier. We finally discuss various overheads introduced in Android due to our modifications.

5.1 Experimental Setup

We have a set of 1,000 malware from the Android Malware Genome Project [46] and a few hundred legitimate applications, which we have manually collected. We randomly select 400 malware and 400 legitimate applications for training the system. Training is done as described in Section 4.1.

In both phases, training and classification, some applications do not produce any Binder activity. This is mainly because they crash or they do not run as expected. There were no applications that run normally in our testbed and produced zero Binder activity in the three minutes testing period. The diversity of different Android versions is the main reason for applications crashes or abnormal runs. All these problematic applications are not taken into account. We plan to port BinderProfiler on older Android versions for having a multi-version environment for covering a broader range of software in the future.
Table 5.1: Summary of the experimental setup.

<table>
<thead>
<tr>
<th>Malware set size</th>
<th>1000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Malware excluded</td>
<td>175</td>
</tr>
<tr>
<td>Legitimate applicaitons set size</td>
<td>825</td>
</tr>
<tr>
<td>Malware used for training</td>
<td>400</td>
</tr>
<tr>
<td>Legitimate applications used for training</td>
<td>400</td>
</tr>
<tr>
<td>Unknown malware used for classification</td>
<td>425</td>
</tr>
<tr>
<td>Unknown legitimate applications used for classification</td>
<td>425</td>
</tr>
</tbody>
</table>

The weights are calculated multiple times, since the application taxonomies are computed for multiple depths. Unknown applications that are classified are not added to the training set. We could have our system dynamically adapt by extending the training set each time a correct classification happens. Adding the application needs recomputing the taxonomies and weights. The results presented in this chapter do not include this dynamic behavior. However, in Chapter 6 we discuss a prototype service based on BinderProfiler, which periodically updates all weights based on manually confirmed classifications. We present a summary of the experimental setup in Table 5.1.

5.2 Accuracy

We plot the Receiver Operating Characteristic (ROC) curve in Figure 5.1 for the two scoring functions, namely the application frequency aware and the non application frequency aware one, for various depths in each case. Notice, that our system achieves the best performance for two setups. If the non application frequency aware scoring function is used, then for depth 6 we receive 8.72% false positives and 9.88% false negatives. On the other hand, if the application frequency aware scoring function is used, we receive false positives 14.53% and false negatives 6.58% for depth 4.

There are a number of observations. First, using the application frequency increases false positives, but reduces false negatives. Recall that application frequency expresses the percentage of malicious applications sharing a particular offensive path (i.e., a path, which was mainly recorded in the malware set), or the percentage of legitimate applications sharing a particular non offensive path (i.e., a path, which was mainly recorded in the legitimate applications set). Generally, the malicious activity is usually a
small part of the whole application graphlet. As a result, it’s common to notice possible malicious paths at low frequency. The application frequency aware scoring function increases the score for those paths, hence there is an increased chance to identify malware reducing false negatives. As a trade-off, the legitimate applications that are identified as malware increase.

Second, remember that our technique is based only on IPC activity. By looking only at IPC communication we can classify the majority of unknown malware with approximately 9 to 15% of false positives. A two-digit percentage of false positives might seem high, but recall that BinderProfiler aims primarily at assisting the security analyst and not operating as detector. Moreover, we strongly believe that if we apply our technique to larger datasets, while being capable of supporting more Android versions, then our results will improve further.

**Figure 5.1:** Receiver Operating Characteristic (ROC) curve for the two scoring functions, namely the application frequency aware and the non application frequency aware one, and for various depths in each case. Notice, that our system receives the best performance for two setups. If the non application frequency aware scoring function is used, then for depth 6 we receive 8.72% FPs and 9.88% FNs. On the other hand, if the application frequency aware scoring function is used, we receive FPs 14.53% and FNs 6.58% for depth 4.
We believe that our technique is effective for two reasons mainly. First, the malware set is rich in applications performing IPC communication for accessing the telephony functionality. Second, as it has already pointed out by similar works [46, 45], many of the malicious applications are repackaged variants of a single application, i.e., there are malware families, which share common IPC patterns. Thus, currently deployed malware can be easily exposed in terms of IPC. In Chapter 7, we discuss how malware can be evolved to evade detection in the IPC domain.

5.3 Overhead

We measure the overhead imposed by the emulator when BinderProfiler is used for capturing all IPC traffic. We use two custom applications. One application is sending an intent to the other and is receiving the reply. We measure the time needed for the message to be delivered from one application to the other, and the time needed for the reply. We perform thousands of such transactions in a modified and non-modified emulator. We plot the CDF
5.3. OVERHEAD

Figure 5.3: CDF of the average throughputs needed to transmit a compressed IPC trace of each application from our pool of Android software, both legitimate and malware. On average we need tens of KBits/sec for transmitting the IPC traffic created by a 5-minutes analysis.

of the times needed for each message, and each reply, to be delivered with BinderProfiler running and not in Figure 5.2. Observe that BinderProfiler introduces less than 10 ms per message.

We further explore the volume of IPC traffic generated by each application during an analysis of 5 minutes, to see whether it is realistic to outsource all information collected by BinderProfiler to a cloud infrastructure. Notice, that this is the raw IPC traffic, i.e., all captured information, which has received zero analysis. We plot the CDF of the average throughputs needed to transmit a compressed IPC trace of each application in Figure 5.3. Observe that on average we need tens of Kbits/sec. This is of the same order as similar infrastructures [33]. Also notice that IPC traffic is triggered artificially by Android Monkey [12] and not by normal human usage.

BinderProfiler is primarily designed for off-line usage. In Section 5.4 we explore some ideas for real-time operation. There is space for further improvement of the currently deployed prototype. We plan to perform various optimizations in our future work for making BinderProfiler more lightweight
for real-time usage.

5.4 Further Applications

BinderProfiler is a generic system that can be applied in various security and privacy applications. So far, we have explored an algorithm for classifying malware. We now explore how the collected IPC information can be used for other applications.

5.4.1 Private Information Exfiltration

Many applications, intentionally or not, leak information, which may be considered private and sensitive. This may include location coordinates, address book information, etc. The State-of-the-Art in detecting such information leakage is by using tainting [22]. However, there are two main problems with tainting. First, system-wide tainting imposes significant overhead. Second, tainting detects information, before leaving the device, and thus cannot capture implicit control flows. Consider, for example an application receiving the geographical location of the user, and transmitting them using a side-channel. Instead of sending the actual location in the network, it transmits a number of packets towards a particular colluding host, which can eventually decode them, by inspecting various packet headers and not the actual contents of them, in values that reveal the location of the device. Using side-channels in mobile devices has been extensively explored in Soundcomber [38].

BinderProfiler can not detect side-channels, but can easily identify applications that request private information, since sensitive information will be requested through IPC communication. Most importantly, BinderProfiler can do that with low overhead. BinderProfiler can have a list of services offering sensitive information and alert the user, whenever an application communicates with the particular service. The user, then, will be able to add the running application in a white or black list.

5.4.2 Real-time Detection

We have demonstrated how BinderProfiler can assist in malware identification in off-line mode. The system can be potentially leveraged for real-time detection. We can achieve this by using the following two models of operation. First, we can capture all activities towards malicious resources. For example, calling or texting to premium numbers can be detected in real-
5.4. FURTHER APPLICATIONS

We expect that this information needs limited storage resources. Second, we can apply the malware identification algorithm we presented in Chapter 4 in real-time for all running applications. There are two challenges here. First, the classification algorithm is developed for capturing the IPC activity of isolated applications running in an emulator. A real system is expected to have multiple applications running at the same time, something which can potentially increase unwanted noise in IPC. Second, the library of known graphlets needs further storage requirements. One possible workaround for mitigating these issues is to outsource all computation needed for classification in the cloud, where all graphlets are securely stored, in the spirit proposed in Paranoid Android [33].

In Chapter 5 we have explored overheads imposed in Android due to our modifications (see Figure 5.2), as well as the throughput required for outsourcing all information in a cloud infrastructure for further processing (see Figure 5.3). Notice, that the overhead imposed is not significant (less than 10 ms per message exchanged) and the throughput is of the same scale with proposed infrastructures that collect information related to system calls activity [33].

5.4.3 Colluding Applications

In contrast to iOS, where Apple monitors all software available for the platform, Android applies security constraints in applications using a permission-based model. Each installed application declares all permissions it requires at install time, and the user is free to decide whether the software’s requirements are compatible with their needs or not. This permission model has been explored by the research community [25], and many systems have been proposed for making this model more secure and more accurate [24, 32, 47]. Some researchers have also identified permission violations from stock applications shipping with smartphones [27].

Detecting and enforcing the right permissions at installation time can mitigate or reveal a number of over-privileged applications. Unfortunately, we expect that malware authors will soon be motivated to distribute many less-privileged applications, which, if combined, can be considered as an application composition powerful enough as the union of the permissions each one of the individual applications has. This tactic, known also as permission re-delegation [26], was demonstrated in Soundcomber [38], where
Multiple applications collude to steal data from the victim’s device. One may argue that forcing the user to install many applications is considered hard. However, since many malicious applications are re-packaged forms of popular legitimate ones [45], and taking into account that Android markets are overflowed with popular software, the probability of having multiple malicious applications installed can be considered significant. Especially, if we consider that an application may lure a user to download another one, which pretends to enhance the overall user experience by adding new functionality.

BinderProfiler can be efficiently used to detect such application communities that communicate with each other. We developed one such setup with two malicious applications that collude in order to exfiltrate the Address Book from the device and transmit it to a server. Notice, that normally you need one application for carrying out this task. However, this application requires access to both android.permission.READ_CONTACTS and android.permission.INTERNET, which, at installation time, may raise privacy concerns. In our scenario, which is heavily inspired by Soundcomber [38], we develop one application asking permission for android.permission.READ_CONTACTS and one for android.permission.INTERNET. The second application, denoted as com.example.malware2 asks from the first

Figure 5.4: An example scenario of two applications that collude for exporting the address book of a device. com.example.malware1 has permissions for accessing the address book, but has no permissions for using the network. com.example.malware2 has permissions to used the network, but has no permissions for accessing the address book. com.example.malware1 requests the address book at time \( t_1 \) through system_server and finally the address book is delivered to com.example.malware2 at time \( t_4 \). com.example.malware2 can exfiltrate the address book, since it has network access. By just inspecting IPC traffic is trivial to identify such behavior.
one, denoted as \texttt{com.example.malware1}, for the address book. After receiving the information, \texttt{com.example.malware2} can transmit the address book, which was never permitted to acquire, to an external server.

We depict the traffic as was captured by BinderProfiler in Figure 5.4. We have labeled each arrow with a timestamp $t_i$. The following convention is used: $t_1 < t_2 < t_3 < t_4$. Thus, each communication is ordered in the time domain. By applying time correlation for this series of events we can speculate that two application communicate. Identifying application cliques through IPC activity can be potentially be effective, especially, in combination with tools that investigate the effect of Android permissions [47].

We consider that colluding applications and communication over side-channels in Android malware is a new field of research. We expect this work to assist in developing event-based algorithms for identifying malware synergies.
6

Deployment

6.1 Overview of operation

G BinderProfiler was originally designed for assisting the security analyst. However, while conducting the experiments for this work, we discovered that BinderProfiler could be potentially implemented as a service and assist the end-user. The nature of the system is such that permits any user, no matter their technical background, to utilize its results. Any user can reconsider installing an innocent looking application after quickly inspecting its application graphlet. If, for example, the graphlet is similar to Figure 3.2, then it is evident that this particular application does more than it advertises. Notice, that our system is not powerful enough to convince someone to install an application by providing guarantees that the application is legitimate, but it is able to raise concerns about a possibly malicious application.

We implemented a first on-line prototype based on BinderProfiler, which runs over the web. The service is designed similarly to other related services, such as on-line antiviruses or malware analyzers [3, 8]. The user can anonymously upload an Android archive and receive a dynamically generated URL, which hosts the results of the analysis. On the background we have implemented scripts that automatically install the Android application.
The service is designed similarly to other related services, such as on-line antiviruses or malware analyzers [3, 8]. The user can anonymously upload an Android archive and receive a dynamically generated URL, which hosts the results of the analysis.

in an emulator which runs BinderProfiler. The under-analysis application runs for three minutes in the modified emulator, its application graphlet is exported, as well as its scores as they are calculated based on the classifier we have built using our existing malware database [46].

The user is able to visit the web site at a later time and inspect the application graphlet and the various scores exported by the system. Together with the scores, there is text, which advices the user about the level of maliciousness of the application, according to the system. Each application is processed for a few minutes, so we have implemented a virtual queue that hosts user requests. So far, our system is based on a few emulator instances, but we plan to enhance this with actual devices in the near future.

Finally, the user has the ability to give feedback to the system. They can confirm that the application is indeed malware or legitimate based on the application graph. The system then puts the application in a group of classified applications which are used to retrain the existing database of application graphlets. We depict screenshots of the home and results page of the BinderProfiler prototype, in Figure 6.1 and Figure 6.2, respectively.

In addition, we implemented an Android application for submitting application package files to BinderProfiler for analysis. It is implemented and tested against Android 4.0, API level 14. The application lists all available
Figure 6.2: The page hosting the analysis results of our experimental BinderProfiler prototype. The under analysis application runs for three minutes in the modified emulator, its application graphlet is exported, as well as its scores as they are calculated based on the classifier we have built using our existing malware database [46].

packages on device and the user is able to select an installed application for analysis.

If BinderProfiler is available on device, the selected application could be invoked and run for a specified time while BinderProfiler logs all IPC traffic data generated at this period. The user is informed about the submission of the data to the service. This approach is not optimal for gathering the IPC data log, since it raises privacy concerns about submitting user data which might contain sensitive information. Also, the requirement of the user to wait while the BinderProfiler gather results is user unfriendly. Finally
it requires BinderProfiler kernel modifications in user’s device. Figure 6.3 shows the dialog of data gathering process.

Hence, another approach is that the application is able to submit the whole installation package archive file, as it is stored in user’s device upon installation. The package will be installed and executed in BinderProfiler service, without requiring any other data from user’s side. Furthermore, there is the ability to submit a URI pointing to an Android package file. The user is informed about the result after successful execution in the service. This procedure is followed also if BinderProfiler is not available on device. This approach is recommended since it doesn’t provide the drawbacks described in previous one. We used Samsung Galaxy S3 I9300, in addition to Android Emulator, for evaluating both approaches.

6.2 Future Work

There are many challenges for deploying this model in practice. First, a broad range of devices, using different Android versions, should be employed. Notice, that installing applications directly from an Android market in the emulator is not feasible, yet. Thus, having many emulators running with different Android versions is not suitable in this case. Also, we expect that
many malicious applications in the near future will try to resolve if they are running inside an emulated environment or not. There are various tricks for making the emulated environment look like a real system. For example, in Android emulator many hardware identifiers, such as the IMEI is zero by default. A malicious application could inspect these and easily find out if it is running under an emulated environment. An obvious solution is to change these identifiers to reflect the appearance of an actual system. Detecting emulated environments is a field of research by its own [35]. We believe that it is not always trivial to hide that an application is emulated.

Moreover, even in actual smartphones, a malicious application could try to resolve if the system supports a particular feature, before actually exploiting it. For example, in the case of an SMS exploiting malware, the smartphone should have an actual plan or contract for sending SMSs. Finally, certain malware could compromise the device if they are capable of rooting the system. In that case, the smartphone must be reformatted for removing all changes performed by the offensive application. We plan to further investigate all these issues in follow-up work.
7

Limitations and Considerations

7.1 Evading Detection

Our system associates particular IPC communication patterns with suspicious behavior. An obvious strategy to evade detection is enhancing the malware with dummy IPC communication, which conceals all patterns that can be considered malicious. This can be combined with non-aggressive, stealthy malware. The design of BinderProfiler was mainly driven by the observation that much malware performs similar actions, and there are often variants of a main malware that defines a particular family. Applying the aforementioned strategies raises the bar for malware authors. It may be easy to introduce dummy IPC activity, but it is really hard to remove the fundamental actions that imply malicious behavior. We believe that all malware classification techniques experience an arms race between the algorithm’s accuracy and the techniques malware employs to evade detection. Finally, we believe that BinderProfiler can provide complementary information to the many other tools that perform malware analysis.
7.2 Performance Overhead

BinderProfiler was initially designed for off-line operation. However, it was also ported on an actual device for collecting all IPC traffic. Making BinderProfiler operate in real-time had many challenges. First, IPC traffic is significant in an Android system. This is why we believed that simply by looking at IPC traffic you are able to characterize applications in the first place. Logging and transmitting all this traffic incurs high overheads. As we showed in Section 5.4 a device must transmit Kbits/sec for profiling just one application. Much of this traffic is redundant as it is associated with GUI activity. Applying a pre-filtering at the device can reduce this overhead, but our current implementation applies filtering inside BinderProfiler core.

7.3 Message Parsing

Parsing an IPC message is not trivial, since it has been already serialized and it possibly includes Java objects, whose semantics we are not aware of. Thus, we extract only printable strings, because class names and other sensitive information, such as contacts or telephone numbers, are expressed in text. Moreover, Android processes delegate other services for performing the actual actions. These delegated services act as proxies (see for example system_server in Figure 3.1). In this work, we have not attempted to analyze some of the most popular services used for delegating the communication, and shipped with the Android operating system. Our original goal was to show that IPC activity can be an efficient descriptor for Android applications, even in cases where the exact payload of the communication is not known. However, some of the core Android services could be analyzed, in order to reveal the actual IPC activity (the exact payload of each communication). In that case, our classification algorithm presented in Chapter 4 could be further improved as well as evaluated with recent malware applications, possibly based on real life malware-legit rates.

7.4 Device Rooting

There are malicious applications, which attempt to compromise a device. This process is called rooting or jailbreak, and is essentially happening by using a core vulnerability of the software running at the device. After compromising the smartphone the malicious application is literally free to do anything. If BinderProfiler was enabled in real-time mode, the malicious
application could easily turn off its operation or replace the kernel with one of its own. Detecting such attempts is out of the scope of this work. There are many proposed techniques for detecting device compromising. Ensuring Control Flow Integrity (CFI) [14] is one of the methodologies that have enjoyed attention by the research community [30, 19].

Although, we do not account for this kind of malware, we believe that BinderProfiler is still valuable to the research community. Consider that the majority of Android malware does not aim at rooting the device, but exploiting it by calling or texting premium numbers, stealing the address book, etc. [46] All these actions can be effectively captured by inspecting all IPC communication.
In this work we developed BinderProfiler, a novel system for analyzing Android software based solely on IPC activity. We showed that Android malware can be effectively described by simply observing the IPC communication. We argued that, in contrast with source-analysis methodologies, which examine applications at the microscopic level, we can enjoy similar efficiency by examining software at the macroscopic level, i.e., by looking at high-level operations. Android is a suitable platform for our technique, since it is built as a service-oriented platform. Each process needs to request access from a service for acquiring the permission to use particular resources of the system, such as the telephony functionality of a smartphone. This involves a series of IPC transactions, which expose the functionality of the application.

We introduced application graphlets; graphs that depict all IPC communication performed by an Android process. Based on application graphlets, we developed a classification algorithm, which efficiently identifies unknown malware with 9 to 15% false positives. BinderProfiler is designed to be a tool for assisting the security analyst and the end-user, and thus, is more tolerant to false positives, than an isolated detector. We further, discussed various applications, where information encapsulated in application graphlets, such as privacy exfiltration or permission re-delegation, can be utilized. Finally,
we delivered a prototype implementation of BinderProfiler, which can be accessed through the web. We believe that the graphical results produced by BinderProfiler can be useful even to non-expert end-users.

BinderProfiler is the first attempt at macroscopic malware detection. We believe that the results outlined in this work are promising, and can be used in conjunction with other techniques in the fight against mobile malware.
Bibliography


