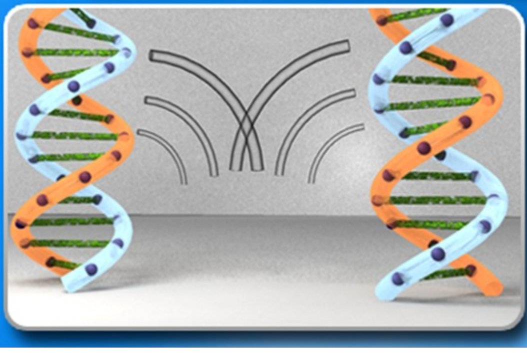
TR-nexGINRC-2012-65

Copyright © nexGIN RC, 2012. All rights reserved.

Reproduction or reuse, in any form, without the explicit written consent of nexGIN RC is strictly prohibited.

An Intelligent Secure Kernel Framework for Next Generation Mobile Computing Devices (ISKMCD)

NEXT GENERATION INTELLIGENT nexGIN RC NETWORKS RESEARCH CENTER



Technical Report

An Intelligent Secure Kernel Framework for Next Generation Mobile Computing Devices

(ISKMCD)

"A Survey on Recent Advances in Malicious Applications Analysis and Detection Techniques for Smartphones"

Dec 12, 2012

National University of Computer & Emerging Sciences, Islamabad, Pakistan

A Survey on Recent Advances in Malicious Applications Analysis and Detection Techniques for Smartphones

Farrukh Shahzad, M Ali Akbar and Muddassar Farooq

Smartphones are becoming the core delivery platform of ubiquitous "connected customer services" paradigm; as a consequence, they are attractive targets of malicious intruders (or imposters). Researchers have realized that classical signature-based anti-malware techniques are not capable of providing efficient and effective detection tools against novel, zero-day and polymorphic malware for resource constrained smartphones; therefore, in last couple of years unconventional (non-signature) intelligent solutions, based on behavioral analysis (static or dynamic) have been proposed. In this survey article, we provide an overview of the recently proposed static and dynamic malicious application detection techniques for smartphones. The survey provides relative merits (or demerits) of each technique that would enable security researchers and practitioners to propose next generation security solutions and tools for smartphones.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection; K.6.5 [**Management of Computing and Information Systems**]: Security and Protection

General Terms: Security

Additional Key Words and Phrases: Smartphone, Malware, Static analysis, Dynamic analysis, Privacy Leaks, Tools and techniques

1. INTRODUCTION

Recently, mobile hand held devices, including smartphones and tablets, are having the same computing power as that of the desktop computers of the last decade of the previous century. As a result, the smartphones have integrated as an essential enabler for accessing "connected services" in a ubiquitous manner. Smartphones are now enabling customers to access m-government services, to stay connected with the family and friends on the social media, to do electronic transactions and e-commerce, to participate in audio/video live streaming conferences, to remotely attend e-learning classes and to diagnose and monitor patients in an e-health (or m-health) environment. As a consequence, the number of users of smartphones have exponentially grown globally. In a recent survey of Gartner Research conducted in 2011 (4th quarter), the smartphone market grew by 47% as compared to 2010 (4th quarter) [1]. In the beginning of 2012, mobile phones market has reached

Authors' address: Next Generation Intelligent Networks Research Center (nexGIN RC), FAST National University of Computer and Emerging Sciences, A.K. Brohi Road, H-11/4, Islamabad, 44000, Pakistan.

Permission to make digital/hard copy of all or part of this material without fee for personal or classroom use provided that the copies are not made or distributed for profit or commercial advantage, the ACM copyright/server notice, the title of the publication, and its date appear, and notice is given that copying is by permission of the ACM, Inc. To copy otherwise, to republish, to post on servers, or to redistribute to lists requires prior specific permission and/or a fee.
© 2012 ACM 1529-3785/2012/0700-0001 $5.00

419.1 million units, out of which 144.3 million are smartphones [2]. Mobile phone market is anticipated to reach 645 million till the end of 2012 which will surpass the number of personal computers worldwide [2]. Google's Android based smartphones dominate the market with a maximum of 64.1% overall share, while Apple's iOS based smartphone have 18.8% share in the second quarter of 2012 [3]. In Table I, we have summarized the share of different smartphones that dominate the market. In another report by Kaspersky Lab [4], 16% of smartphone users store and transfer private documents from their smartphones, 53% use smartphones to send or receive emails, and 47% stay connected on the social networks (Facebook, Twitter etc.) using their phones. Typically, 62% of smartphone users browse the Internet using their smartphones. The top security officials at Symantec and Mcafee confirm [5] that the increased market share of smartphones have made them an ideal target for hackers and malicious software writers. They are now shifting their focus on smartphones instead of desktop computers.

The major security vendors have recently reported an alarming rise in the malware attacks on smartphones. These malicious applications[1] are a source of serious concern for two reasons: (1) they have the ability to take control of the phone and perform unauthorized activities in a stealthy (unnoticed) manner; and (2) they can access a user's private information and leak/sell it to his adversary or different advertising agencies. In the "mobile threat report 2011" by Juniper Networks Inc. [6], the overall growth for all mobile operating systems, in their malware sample database, during 2011 was 61%. In case of Android, the malware samples increased from 400 to 13302 (an increase of 3325%) during the last seven months of 2011. The distribution of malware samples for different operating systems in 2010 was: JavaMe:70.3%, Windows Mobile:1.4%, Symbian:27.4%, BlackBerry:0.4%, and Android:0.4%. In comparison, the unique malware samples collected in 2011 show a totally different distribution: JavaMe 41%, Windows Mobile 0.7%, Symbian 11.5%, BlackBerry 0.2% and Android 46.7%. To conclude, the number of threats on Android OS have significantly increased. The report also provides a taxonomy of the malware observed in 2011: 63.39% are Spyware, 36.43% are SMS Trojan, 0.09% are SMS-Flooder, and 0.09% are Worms. On Android platform, fake installers are the biggest infection vector (56% of the total threats). The Internet threat report of Symantec Corporation [7] also confirms a dramatic increase in smartphone malware. Their analysts have categorized the malware on the basis of functionality and potential risks: 28% malware are data collection applications, 24% send smartphone stolen contents to the remote hosts, 25% are used for location tracking, 7% change the device settings and 16% are traditional threats (virus and worms etc.). F-Secure Labs in their mobile threats report 2012 [8] report that only 10 new malware families (along with their variants) were known by the beginning of 2011. Only within a year, 37 new malware families and their variants have been discovered. They report that malware writers are exploring new infection vectors and are focusing on evading malware detection techniques. Take the example of already known malware families – *DroidKungFu*, *GinMaster*, and *Fakeinst umbrella* – that are using cryptography and randomization techniques for evasion. Malware writers have also used images to hide the malicious code in them e.g. *FakeRegSMS*. Fur-

[1]We will use the terms malware and malicious applications interchangeably in this paper.

thermore, they report that profit motivated threats (34%) are significantly higher as compared to non-profit threats (15%). A sixfold increase in malicious applications (175,000 in September compared with 30,000 in June) has been reported on Android platform in the third quarter of 2012 [9].

The exponential increase in malware on smartphones has started an anti-malware product race within security vendors worldwide. Most of them have enhanced their signature based anti-virus products [10] for smartphones. As a result, the phones are protected against malware whose signatures exist in the database. Malware forensic experts analyze malware and generate signatures that are inserted into the threats databases. The antivirus applications on an end user's phone use the (malware) signature databases to detect malware. The process of creating a malware signature on the basis of the domain knowledge of a forensic expert is time consuming, challenging and error-prone. The true challenge for them is: to create a generic signature to achieve 0% false alarm rate (it should be able to detect mutated variants of the same malware and moreover it should not detect legitimate programs as malware). Malware writers typically release their malware in families consisting of multiple variants of a single malware having minor changes in functionality, structure and/or code. Every week, the security vendors mostly receive a large number of malware variants out of them only few dozens are new malware families [11]; therefore, the capability to automate the process of malware analysis and detection is gaining momentum. Broadly speaking, malware analysis and detection techniques for smartphones can be categorized into two classes: (1) static detection techniques, and (2) dynamic detection techniques. Static analysis is performed on the disassembled binary files without executing the malware. On the other hand, dynamic analysis is performed by monitoring the malware and/or its effects on a system during or after execution of the malware.

The major contributions of this survey paper are:

(1) Focusing on niche and state-of-the-art malicious applications detection techniques for smartphone platforms.[2]
(2) Formulating a generic detection framework for malicious applications that will be used as an architecture guideline for mapping (or understanding) existing anti-malware security products.
(3) Discussing the emerging and new (recently proposed) detection techniques that

[2]The most of existing survey papers lack comprehensive treatment on mobile malware and instead focus broader smartphone security.

Table I. Market Share of Smartphone Operating Systems (thousands of units)

OS	2^{nd}Quarter 12		2^{nd}Quarter 11	
	Units	Market Share (%)	Units	Market Share (%)
Android	98,529.30	64.1	46,775.90	43.4
iOS	28,935.00	18.8	19,628.80	18.2
Symbian	9,071.50	5.9	23,853.20	22.1
RIM	7,991.20	5.2	12,652.30	11.7
Bada	4,208.80	2.7	2,055.80	1.9
Microsoft	4,087.00	2.7	1,723.80	1.6
Others	863.3	0.6	1,050.60	1
Total	153,686.10	100	107,740.40	100

are used by smartphone security and privacy analysts.

(4) Surveying (at length) recently proposed tools – using the above-mentioned techniques – with an aim to understand their relative merits and shortcomings.

The rest of the paper is organized as follows. In Section 2, we enumerate various types of malicious applications and the infection vectors they exploit on smartphones. In order to develop detection techniques, it is relevant to understand the important challenges; therefore, we enumerate them in Sections 3 and 4. In Section 5, we present a generic detection framework for mobile malware. The framework augments the system wide understanding of malware analysis and detection techniques. We also present, analyze and categorize the latest published techniques for static and dynamic malware detection on smartphones in Section 6 and Section 7 respectively. In Section 8, we describe and review the recently developed malware detection frameworks and tools that utilize these techniques. The related work is presented in Section 11. Finally, we conclude the paper with an outlook towards the future directions of detecting mobile malware.

2. MALICIOUS APPLICATIONS

In this section, we describe the types of malicious applications that have plagued the smartphone systems over the last few years. We also enumerate the commonly used methodologies and typical infection vectors utilized for spreading malware infections to new (uninfected) smartphones.

2.1 Types of Smartphone Malicious Applications

In order to make the paper self contained, we provide a brief overview of different types of malicious applications in this section. An interested reader can refer to [12] [13][14][15][16][17] for an in-depth discussion on smartphone malicious applications.

A malicious code that usually spreads by exploiting vulnerabilities in the network services is termed as a *worm*. The worms usually run as independent executables and make copies of themselves on the networked machines. The oldest known worm is the famous Morris worm [17]. On smartphones, the first known worm Caribe (Worm.SymbOS.Cabir) was released in June 2004 [18]. Other notable worms include iOS Ikee Worm [19] and Commwarrior [20].

Viruses infect normal processes on a system and use them to execute their malicious code. They usually spread through sharing of infected programs. Duts [21] is one of the well known viruses for smartphones.

Trojan Horses pose themselves as legitimate and productive applications and they execute their malicious codes in the background without user's knowledge. They often spread by infecting other programs on the new system, and are often used as a gateway for installing additional malware on the infected system. Most malware for smartphones are trojans. Some well known trojan horses include Trojan-SMS AndroidOS.FakePlayer [22], ZeuS Trojan [23], Trojan-SMS AndroidOS.Foncy [24] and Skull.D [25].

Spyware present themselves as useful programs but their primary objective is to steal sensitive information about a user such as passwords, emails, user surfing behavior, credit card information etc. Some notable spyware include GPSSMSSpy and Nickyspy [12]. It is important to realize that most of mobile malware writers focus on stealing a user's private information and use it to characterize her/his behavior in order to spam her/him with relevant advertisements. In 2011 alone, 63.39% of mobile malware were Spyware[6]. As a consequence, protecting a user's privacy and detecting malicious applications have become synonymous in the security jargon.

Sometimes the primary purpose of an attacker is to get a large number of computing resources at her/his disposal. This is usually achieved by a specific type of malware known as *Bot*. Bots (infected computers) dial home to a master bot that then controls all the bots and issues commands to them. The whole network of bots and master bot is known as botnet. Typical uses of botnets include denial of service attacks, spamming, fraudulent activities etc. Anserverbot, Nickybot and Beanbot [12] are representative malware of this category.

A *rootkit* hides a malicious application on a system by modifying a system's kernel such that the system API calls are instrumented (the logging and other similar operation binaries are replaced) [26]. TDL, ZeroAccess [27], ITFUNZ and Z4Mod [28] are examples of rootkits infecting mobile devices.

It is important for a hacker to ensure access to a hacked system even if the present vulnerability is patched in the future. *Backdoors* serve exactly this purpose. The presence of a backdoor allows unauthorized access to a system that enables a remote attacker to run commands on the system, usually by opening a port and waiting for an attacker to connect. BaseBridge-C, FAKE Angry and KMin on Android [28] and ZTE Score M and ZTE Skate mobile phones have been known to contain such backdoors [27].

Modern malicious application writers have established an underground malware mafia industry to quickly accumulate wealth. *URL injectors* replace actual search results and web links in a user's browser with alternate links, webpages and sales pages that transfer revenue to the malware author or her/his associates. *Adware* programs usually display ads and pop ups trying to force a user to click on them and buy the products being advertised using affiliate accounts. Affiliate advertising brings in money for the malware writer. SslCrypt on Symbian OS [29] and Toplank (Counterclank) [30], Plankton [28] are examples of adware. Some malware applications (*Money Stealers*) perform money sending actions such as sending premium SMS messages to a malware author or her/his associates account without user's consent. Foncy on Android [24] is one such example. A collection of latest malware applications for Andorid, iPhone and Symbian platforms is available on the website *www.contagiodump.blogspot.com*.

2.2 Infection Vectors & Methodologies

To protect malware infections, it is important to understand how malicious applications infect a smartphone and spread it to other systems. We now introduce the most important infection vectors for smartphones only.

The major source of automated infection spreading are exploitable vulnerabilities in applications that listen to network traffic or process incoming traffic. Malicious applications writers can automate the process of exploitation for such scenarios; as a result, infection spreads very quickly in a short time (specially) when the vulnerable services are common across many systems on the Internet. This infection methodology is utilized by worms.

When a user is browsing the Internet, malicious websites can exploit vulnerabilities in her/his browser to download and install malicious code on her/his machine without her/his consent [6]. A user doesn't necessarily need to visit malicious websites for such attacks. The techniques – cross site scripting (XSS), clickjacking, script injection and iframe injection – are typically used by attackers to spread malicious code to the visitors of innocuous websites. This methodology is usually coupled with social engineering exploits to trick a user to go to the malicious website and become infected.

Social Engineering techniques use (or misuse) a user's trust paradigm or naivety to entice him/her into actions that make it possible for a malicious application to do the malicious activity. For example, a link in an email from a coworker's email address could actually lead to a malicious website that installs malicious code on a user's system. In early days of Internet, emails and chat messengers were the primary technologies used for such attacks. With the evolution of social networks, expanded online social circles, interaction with strangers and user provided content publication, novel social engineering attacks are becoming straightforward. A tweet on a famous hashtag with a malicious link and a promise of never-seen-before video of a celebrity is all it takes nowadays to entice thousands (if not millions) of susceptible users to download and install a video plug-in that is actually a trojan.

Vulnerabilities in the operating system allow an un-privileged program to gain enough privileges to infect the entire system, including the operating system files, and the directories of other users; as a result, infection spreads to all users of the machine. Removable Media such as SSD cards may contain infected files which may execute when the removable media is connected to the system. The malware on the infected system tries to copy itself to the new removable media devices and thus spread further. On a network with shared resources such as shared file servers, email clients etc., the malware can spread by copying itself to writable locations (or infecting executable files existing there), and then infecting the systems of the users who access those resources.

Smartphone platforms usually allow access to various protected system resources using a permission model. Over-privileged smartphone applications can become an important infection vector that can allow an application to leak private information or execute tasks not authorized by the user. It is also possible that another malicious application can use a legitimate privileged application as a deputy for an attack if the legitimate application provides an open interface for invocation. This can lead to privilege escalation attacks as well.

An attacker can setup rogue public WiFi access points, luring users to enjoy free Internet access [6]. By pointing to a rogue DNS server controlled by the attacker, the users can be redirected to malicious websites which can infect the system through drive-by download methodology described earlier. Bluetooth and MMS functionalities are ubiquitous. Bluetooth [31] connections and MMS have the capability to help a malware leverage the system vulnerabilities to install itself on the system. Commwarrior [20] is an example of malware that spreads through MMS messages.

Some mobile platforms (e.g. Android) support open application distribution architecture. A user can download applications from virtually any place on the Internet without the need of getting them signed from a competent authority. The systems with open architecture are highly susceptible to rogue application distribution sites that trick users to download rogue applications and install them [6].

3. CHALLENGES FOR MALICIOUS APPLICATIONS ANALYSIS & DETECTION

After discussing the common types of malicious applications and relevant infection vectors that help in propagation of the malware, we now list a number of challenges that are faced by security experts working on designing detection frameworks for malicious applications.

3.1 Generic Challenges for Malicious Applications Detection

We first present the challenges for malicious applications detection that are common to both desktop and mobile operating systems.

Packers. Malware try to evade static detection using packers [12][32]. Packing involves either compression or encryption (or both) of binary executables. This compressed and encrypted image is loaded at runtime to perform malicious operations. Static malware detection schemes – like n-gram analysis – usually fail to correlate the packed malicious binary as malware because their content patterns are significantly altered during the packing process.

Polymorphic malware. Polymorphic malware [32] are similar to packed malware because it contains an encrypted malicious payload and a decryption routine. However, unlike packers, polymorphic malware try to evade detection by re-encrypting its contents, using a different key, every time it executes.

Metamorphic malware. Metamorphic malware modify their code by rewriting themselves on each infection [32] that makes their detection a challenge. The common techniques used to transform the code are: (1) register renaming (using different CPU registers in instructions); (2) code permutation (intelligently ordering instructions to preserve the semantic outcome/result); (3) code expansion (using more instructions to do the same thing); and (4) code compression and garbage code insertion (such as adding NOPs). Dynamic detection of such malware, using advanced virtualization techniques, yield better detection results.

System Performance Degradation. Processing overheads are more important during dynamic detection because they directly contribute towards deteriorating completion time of processes and this may have severe consequences especially for time-critical realtime processes.

Detection delay. A detection scheme should have low detection delay for a better user experience. Similarly, the memory cost of maintaining detection data structures should be within acceptable limits on resource constrained smartphones. Early detection of an executing malware is a challenge because delay might lead to an infected system that might end up in an unrecoverable state. To mitigate this, security solutions keep a log of runtime changes done on the memory and disk, and used rollback mechanisms to undo these changes (disinfection) after detection.

High False Alarm Rate. A malware detection system should ideally have a zero false alarm rate; otherwise, frequent false prompts that ask users to quarantine benign programs will add to the frustration of normal users and they might be tempted to turn off the detection system completely.

Low Detection Rate. In order to ensure a nearly zero false alarm rate, dynamic detection techniques typically increase the threshold of confidence on the basis of which a program is declared as malware; as a result, detection rate is degraded that might leave a user vulnerable to attacks.

Robustness to Evasion. Crafty malware writers can attempt to evade the malicious application detection systems. Evasion is possible when malicious applications can reproduce a feature set that is similar to the feature set produced by legitimate applications. This is a significant challenge for security researchers. Most of dynamic analysis techniques tend to detect a process's behavior by analyzing its behavior in the kernel space of an operating system; therefore, malware try to hide themselves through mimicry of behavior of benign processes. For example, if a dynamic detection scheme monitors pattern of API calls made by the processes, a malware process can try to intersperse its own API calls within sets of benign API calls to evade detection.

3.2 Additional Challenges for Smartphone Platforms

After introducing the detection challenges for malware detection techniques, we now focus our attention to challenges that are relevant to smartphones only.

Battery Constraints. Battery is one of the most important resources on smartphones; therefore, any malware detection solution should use it carefully to provide users with enhanced connected times without the need to frequently recharge.

Limited processing power. Detection algorithms should be simple, having relatively small processing complexity, so that they can execute (without degrading a user's experience) on mobile processors. It is advisable to compromise accuracy for complexity.

Limited Memory. Memory is an expensive resource on a smartphone and detection algorithms should use it in an optimized manner.

Processors architecture Issues. Smartphones typically use processors with low power footprint (like ARM) to conserve battery. It is, therefore, imperative that the detection techniques should ensure that their framework is not tightly coupled with processor specific features.

4. IMPLEMENTATION DECISIONS

In this section, we introduce the design spectrum for malicious applications detection frameworks. A malware detection scheme can plug and play different design options to create a customized detection engine.

4.1 Host-based vs Decoupled Security

The malware analysis and detection process can take place at three points: (1) entirely on a smartphone; (2) partially on a smartphone and partially on a remote server; or (3) entirely on a remote server. The process of delegating security away from a smartphone is known as decoupled security.

A designer needs to make an informed choice by critically reviewing merits and demerits of each option. For example, a family of smartphones might be having limited memory, processing and battery resources; therefore, it is prudent to move compute-intensive behavioral analysis and pattern matching computations to a remote server. But sending a program to a remote server can take more time and Internet bandwidth (might be expensive in 3G networks). This can eventually load a central server that results in a degraded user experience. To conclude, a logical compromise is: to do simple analysis on a smartphone and to delegate compute-intensive calculations to a remote server that might be located in a cloud. Jupiter [33] provides an environment that augments smartphone environments with the cloud computing.

4.2 User space vs Kernel space

The malware detection system can run either in the user space or in the kernel space. For user space implementation, it is necessary that a smartphone operating system provides necessary APIs for collecting the required features (information) about the processes that are running on the system. In comparison, an implementation in the kernel space provides more flexibility because the behavioral information is collected by hooking the kernel functions and monitoring the kernel structures. In terms of man machine hours, a user space implementation takes less effort compared with the kernel space implementation albeit it is less sophisticated and accurate.

One should remain cognizant of the fact that malware can easily detect a user space program that attaches hook to it; as a result, it refrains from doing malicious activity to avoid detection.

4.3 Actual System vs Sandboxing/Emulation

The program (under analysis) can either be run on an actual smartphone with normal privileges and restrictions as that of a trusted program, or it can be run in a sandboxed or emulated environment for analysis. Sandboxing provides a confined operating environment with a security policy that restricts the actions that an application can perform. Emulation means imitating the actual operating environment for an application. The purpose of sandboxing/emulation is to analyze the behavior of a program without endangering the actual operating system.

Sandboxing is sometimes implemented by the operating system of a smartphone. The second option is to install third party sandboxing tools for smartphones. The third option is to use the concept of decoupled security and execute the program (under analysis) on a remote server within an emulator. Sandboxing can be done either in user space or in kernel space. Both of them suffer from the same merits and demerits as already discussed for user and kernel space malware detection systems.

The sandbox is configured to intercept and log the changes made to a system by an executing program. The objective is to log a number of features such as system and API calls, changes to a filesystem, resource utilization, network activity, battery drainage and other parameters of a system that define the system's health and responsiveness.

A program is installed in a sandbox environment. The system maintains a log of changing features during its execution. The programs on smartphones are mostly interactive; therefore, a simulated user input is required for automatic analysis of the sandboxed programs. The programs such as Android's *Monkey*[3] tool allows simulating random user events at different intervals. By simulating user events, a program's behavior in response to the user events is logged. The log of features can be used for dynamic malware detection (during or after execution) by employing techniques like function call monitoring, power utilization, behavioral analysis etc.

[3]Monkey tool can be executed on an Android OS by the command: *$adb shell monkey*

4.4 Static Detection vs Dynamic Detection vs Hybrid Detection

Analysis to detect malicious applications can be performed in two different ways [34]. Static detection is performed before a file is executed, using the features set extracted from an executable file. Dynamic detection, on the other hand, is performed during or after the execution of a program (usually in an isolated environment). Static detection is relatively quicker and inexpensive, and it doesn't require execution of the application; but, it is easier to evade. Dynamic detection systems degrades the performance of a system because of the associated processing overheads. A detection system for malicious applications can be either static or dynamic or a hybrid of both approaches. A hybrid system combines the benefit of early detection of static analysis with the robustness benefit of dynamic systems against evasion attempts.

4.5 One Class (Anomaly Detection) vs Two Class Classification

The basic purpose of a detection system is to classify a given program as benign or malicious; therefore, detection can be treated as a two class problem. But, using anomaly detection systems, it can be treated as one class – the system learns the normal behavior of a program and any deviation, measured with information-theoretic distance measures, from normal is classified as malicious (also known as outlier detection). The anomaly detection can be based on statistical measures (e.g. mean, variance), distance measures (e.g. nearest neighbor, clustering) or models/profiles.

The anomaly/outlier detection is more suitable when it is difficult to get adequate number of unique samples of one class (e.g. malicious). However, for doing anomaly detection, the complete picture of one class behavior is needed to train a classifier (a daunting task in itself). If large samples of both classes are readily available (e.g. Android, Symbian OS), two class classification makes more sense. On the other hand, anomaly detection is a more logical approach for operating systems with limited number of malicious applications (e.g. iOS, Windows Mobile).

To summarize, we have introduced malicious applications types, common infection vectors, design challenges and implementation decisions to be taken in building intelligent malicious applications detection frameworks. Armed with this knowledge, we now focus on a generic framework to design and develop detection systems for malicious applications on smartphones.

5. GENERIC MALICIOUS APPLICATIONS DETECTION FRAMEWORK FOR MOBILE COMPUTING DEVICES

In this section, we introduce a hybrid security framework for detection of malicious applications (causing security threats and privacy leaks) on smartphones. This framework provides a cohesive view of two paradigms (i.e. static and dynamic detection). Moreover, it serves as a common abstract framework through which different instances (by selecting suitable modules) can be instantiated. As a result, it becomes a blueprint for designing future security solutions for detecting malicious applications. Last but not least, it acts as a reference framework for comparing implementations of different solutions, their merits, demerits and performance.

Static and dynamic analysis techniques usually share some common archetypes.

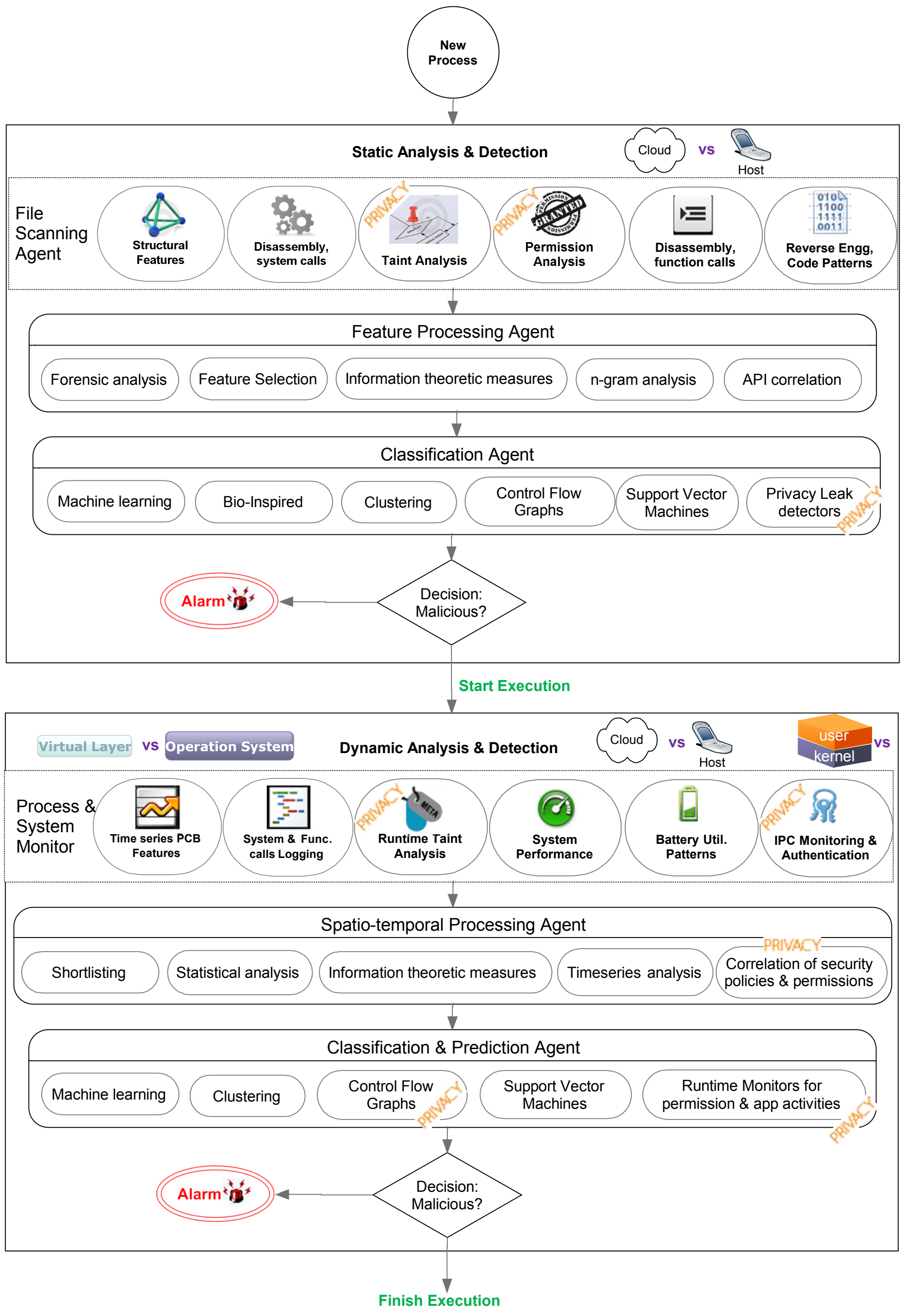


Fig. 1. Proposed generic malicious applications (security threats & privacy leaks) detection framework for smartphones

For example, the first step is to extract the features that are relevant to a program or process. Then, statistical and information-theoretic measure analysis is performed to select features having the ability to discriminate between two different types of processes. The final step is to make a decision by using classification and prediction algorithms. Despite of having these similarities, significant differences do exist between both analysis paradigms. The static techniques are usually employed before execution of a process (i.e. *first-line-of-defense*) while dynamic techniques are employed either at run-time or after the process has executed (i.e. *second-line-of-defense*); therefore, combining both of them in a framework can help provide a two-layer defense against emerging zero-day malware.

The proposed hybrid framework consists of two layers: (1) Static Analysis and Detection Layer (SADL); and (2) Dynamic Analysis and Detection Layer (DADL). The prominent modules of SADL are:(i) file scanning agent, (ii) features processing agent, and (iii) classification engine. On the other hand, the top level modules of Dynamic Analysis and Detection Layer (DADL) are:(i) process and system monitor, (ii) spatio-temporal processing agent, and (iii) classification engine.

The implementation of the framework operations can be done either in the kernel space or the user space or both. (Generally speaking, it is preferred to implement the dynamic layer in the kernel space.) Figure 1 summarizes the flow, interaction and characteristics of different modules. The prominent features, their characteristics and functions of layers along with their submodules – provided in the following subsections – are based on common solutions proposed for both static and dynamic analysis of smartphone applications.

5.1 Static Analysis and Detection Layer (SADL)

Before the program is executed, a binary executable is processed by SADL which performs static analysis to detect security threats or leak of private information. This layer further consists of the following submodules.

5.1.1 *File Scanning Agent.* The file scanning engine analyzes the executable and extracts basic features from it. Static analysis based security solutions on smartphones use features that are extracted from source code and binary executables. Some examples of such features are executable's structural features [35] [36], system calls and library calls in binaries [37] [38], n-gram of assembly instructions [39], malicious patterns analysis in disassembled code [40] and static taint based features [41] etc. Different solutions use different nomenclature for such features and components. Therefore, we have given the generic name 'file scanning agent' to this module in the static analysis and detection layer of our proposed hybrid framework.

5.1.2 *Feature Processing Agent.* The feature processing agent is the second major module within SADL. It uses pre-processing filters to remove the features that are not useful during classification i.e. the features with zero or very low classification potential. It makes sense to eliminate the features of least predictive significance or combine them with other similar types of attributes; as a result, the dimensionality of input attributes space is reduced. Consequently, the detection accuracy is not only increased but processing overheads are also reduced. The common practice is to perform dimensionality reduction by utilizing information-theoretic measures such as information gain or gain ratio measurements, Fisher Score, Chi-squared distribution [42], principal components selection and discrete Haar/Wavelet transform [35] etc. The dimensionality reduction also helps in reducing training and testing time requirements of classifiers.

5.1.3 *Classification Agent.* The primary goal of classification engine is to accurately classify a given feature set as belonging to either benign or malicious class of executables. Most of static detection systems on smartphones use machine learning classifiers – decision trees, inductive rule learners [35], support vector machines [16], bayesian nets [43], neural networks [44] and clustering algorithms [45] etc.

5.2 Dynamic Analysis and Detection Layer (DADL)

To classify the program during (or after) execution by monitoring its runtime behavior, the Dynamic analysis and detection layer (DADL) is utilized. This layer consists of the following modules.

5.2.1 *Process and System Monitor.* Security solutions, employing dynamic techniques, extract features based on the state of an operating system, runtime behavior of processes and their performance logs. The Process and System Monitor submodule is responsible for extracting these features at runtime during specific time windows. The common feature sets are: process control blocks of executing processes in the kernel [35] [46] [47], system call and library call logs of processes [48], in-execution taint features of programs [45], battery utilization based anomalies [49], operating system event logs (free RAM, user inactivity, sent SMS count etc.) [50][51].

5.2.2 *Spatio-temporal Processing Agent (STPA).* In dynamic detection, it is important to estimate and predict the behavior of parameters which change with time and that can be used for malicious application detection. Spatio-temporal Processing Agent is responsible for time series analysis to select relevant features online (while execution) and offline (after the execution of processes is finished) on smartphones. The time series analysis employs statistical techniques such as the moving mean, variance, divergence, time series model building, estimation, and forecasting to compute temporal derivatives of raw features [35]. Dynamic taint analysis [45] is another methodology that is used for realtime or in-execution malware detection. Spatio-temporal Processing Agent also uses the pre-processing filters to eliminate the features or attributes that don't play a vital role in the classification process.

5.2.3 *Classification and Prediction Agent.* The basic aim of Classification and Prediction Engine is to accurately classify a given feature set as belonging to either benign or malicious class of executables on smartphones. Most of dynamic and behavioral analysis based malware detection techniques use learning classifiers for one or two class classification e.g. decision trees, inductive rule learners [35], support vector machines [16], Bayesian nets, neural networks and clustering algorithms [50][45] etc. Moreover, outlier or anomaly detection is also employed on smartphones [52][51].

After introducing the generic framework, we now describe the common malicious applications detection techniques for smartphone platforms. This will help to build a basic understanding of common methodologies, employed by security researchers, to predict and classify smartphone applications as benign or malicious. We have categorized the techniques into static and dynamic analysis paradigms to maintain the flow of discussion.

6. STATIC ANALYSIS TECHNIQUES FOR DETECTION OF SECURITY THREATS AND PRIVACY LEAKS ON SMARTPHONES

Analyzing structure, source code, functions and system calls in a binary executable (or in the source code if available) – residing on a disk or preparing to be launched (without executing it) – is called static analysis. Static analysis techniques are mostly applied on different forms and representations of a program code or executable. In case of source code availability, the static analysis techniques help analysts in finding memory leaks and corruption faults. They also play an important role in quality assurance and correctness of models for different systems. If the source code of an executable is not available, static analysis tools use its binary instructions, disassembled assembly code and the information in structural headers for classification. This section deals with various techniques and approaches that have been applied to perform such static analysis.

6.1 Malicious Code Pattern Detection

A simple and fast way to detect a malware (especially the ones derived from previously known malware) is similar to the signature detection schemes. The mobile application's source code can be searched for existence of malicious patterns of code (or other resources). It is performed at the time of installing applications; therefore, it doesn't result in degrading a user's experience. Obviously, this methodology can be evaded using code obfuscation and polymorphic techniques to create malware.

In most cases, the source code of the mobile applications is not available; therefore, the general steps performed in this scheme are: (1) the application installer packager is decompressed to extract the files (on some mobile platforms (e.g. iOS), the binaries are not only signed but are also encrypted, so decryption needs to take place.); (2) after decryption, binary might need to be unpacked to obtain plain executable; (3) the executable is then decompiled/disassembled to get the assembly listing of the binary (In case of applications compiled as intermediate byte code (e.g. JAVA classes), inexpensive decompile techniques exist to get the original java source.); (4) the observed malicious patterns are stored in a database; and (5) the final step is to search for malicious patterns – a code block, a specific API call or pattern of calls, a combination of permissions, call to the native runtime environment, attempts to bypass the permissions, or attempts to use services or provisions that would quickly deplete the battery – in the executable. A match triggers a malware alarm.

6.2 Static Function Call Analysis

Some researchers use static analysis of function calls in a program to differentiate between benign and malicious processes on smartphones (functions are reusable blocks of code that perform a specific task.). In this methodology, a programme that uses certain function calls is classified as malware. Any good disassembler has a built-in capability to list function calls, existing in the code of a program, and the resulting assembly code (obtained from disassembly of a binary program) to build a function call flow trace. In some binary formats (like ELF) relocation and symbol tables contain the information about function calls. The static function call analysis is done on the pattern of function class flows of a program. It uses statistical distance measures to match the pattern with malware or benign models.

6.3 Static Permissions Leak Detection

Most mobile operating systems allow/disallow use of their resources by defining and applying a permission or capability model. Each application can only access the permissions that it has specifically requested at the time of its installation. These permissions are also confirmed by a user at installation time (or first time use) depending on the policy of an operating system.

Two interacting applications can by pass the permissions model by invoking indirectly the services of another application. Assume that an application A has the permission to access Internet. If another application B can invoke A, it is possible for B to access Internet indirectly without explicitly getting the permission for it. This phenomenon is known as Permission/Capability Leak. If a privileged application – having permissions to perform a privileged action – exposes an interface for invoking privileged actions, it can help un-privileged applications to invoke privileged actions indirectly.

Interfaces can be defined in a number of ways. For example, Android uses the mechanism of Intents, and the iOS allows applications to register as URI handlers. Different applications from the same developer might use the same identifier. As a consequence, applications from the same developers can use a union of the permissions of individual applications. This permits applications to do actions for which permission is not sought and this leads to implicit permission/capability leak.

Possible control flows of a program (as mentioned before) are created using a control flow graph. Indirect control flows (like threads) also need to be catered. Moreover, a control flow for each entry point (in case of multiple entry points) needs to be created and they need to be merged in a single graph. A leak can be detected by finding a feasible path between an interface and the privileged action. If such a path exists, a permission/capability leak has occurred.

7. DYNAMIC ANALYSIS TECHNIQUES FOR DETECTION OF SECURITY THREATS AND PRIVACY LEAKS ON SMARTPHONES

Dynamic analysis techniques monitor behavior patterns in runtime execution traces to detect security threats and privacy leaks. For this purpose, they analyze the information available in process control blocks of processes, function and system calls – their functionality, call sequences and parameters, taint analysis of executing instructions of programs, performance benchmark counters and parameters of OS. Dynamic techniques are relatively more resilient to evasion techniques – code obfuscation, polymorphism and metamorphism – that are successful against static analysis techniques. Dynamic techniques come in two flavors: (1) post-execution – off-line analysis performed on dynamically produced datasets (system calls, taint data); and (2) In-execution – online analysis in realtime to detect malicious programs during their execution to protect the OS and other user programs from their malicious activities. This section summarizes various techniques and approaches that have been used to perform post-execution and in-execution dynamic analysis.

7.1 Information Flow Tracking

A smartphone user would like to protect her/his privacy sensitive information or data that includes user's contacts, messages, device ID/phone number information etc. Moreover, modern smartphones come with several local monitors – GPS, accelerometer, camera and microphone – that can provide useful information about the location of a user to advertisers. But sharing such information with advertisers without the explicit consent of a user is an obvious breach of her/his privacy but also a violation of her/his trust. This type of sensitive data is tracked by using the concept of *tainting*: mark/label the sensitive data and then track its usage by different applications.

The taint labels helps in tracking different transformations that are applied to the original taint data. The entry point of taint data into an application is termed as a taint source. Similarly, a taint sink is the point where taint data leaves the application. APIs for accessing contact lists and GPS information are examples of taint sources and network interface APIs – used to transmit taint data over the network – are examples of taint sinks. A taint sink can be programmed to filter tainted data to ensure privacy of a user. Using taint labels, a sink can detect transformed sensitive data provided taint labels are properly propagated from the source to the sink. In case of "direct data propagation" – direct assignments and string and arithmetic operations – taint labels need to be assigned to track the information flow. In "indirect data propagation", data can be transformed using the address mapping of a known table. An application might map each character in the tainted data to an index of a table (containing only unique values) and then use the index as an address; as a consequence, this indirect memory address of the tainted information needs to be propagated as well.

In case of control flow (if/else), keeping a track of taint labels becomes a daunting task because the information transformation may span over multiple instructions. For example taint data could be tested against different values and its new copy be created without resorting to "direct data" or "address" mapping. In implicit flow, the values are indirectly assigned with the help of branches that are not executed; as a result, tracking becomes difficult.

7.2 Dynamic Function Call Tracing

Dynamically monitoring function calls of a program has been used by the majority of researchers for malware detection on smartphones (and desktop as well). A security driven analysis of invoked function calls of a process provides useful information about the intent of its writer. The main difference from static function analysis is that these techniques analyze the order in which the function calls are made at runtime and generate related function "call flow graph". Classification is generally performed by comparing the call flow graph of a process with that of benign and malware. They employ statistical distance measures to label the executing process as benign or malware.

To enable this, a program needs to be hooked to log a limited number of invoked function calls – using interfaces defined by the operating system – that are useful (like Software Development Kit (SDK) APIs and system calls). For example, Android applications use Java SDK APIs that allow for easy interfacing with the device.

7.3 Runtime Permissions Leak Detection

As mentioned before, modern smartphone operating systems use permission models to allow applications to do operations. Kindly recall permissions can be leaked when an unwitting (or possibly colluding) application that has a permission, exposes an interface that allows another application (having no permission) to request a privileged action on its behalf leading to privilege escalation attacks. Moreover, it is also possible for two colluding applications, with different sets of permissions, to share data with each other through covert channels – extending their permission set to the union of both applications' permission set. These covert channels can be established by applications through shared resources such as contacts database, or observable (and mutable) properties of the system resources (such as volume level, brightness level, etc.).

To detect a permission leak attack at runtime, the interprocess communication needs to be monitored. Specifically, the standard mechanisms provided by smartphone operating system (such as Intents on Android) need to be restricted based on a security policy. An application A that has access to some private data of a user but has no permission to access the network, should not be allowed to communicate to an application B that has permission to access the network.

Even when the standard interprocess communication channels are monitored and restricted, it may be possible for an application A (without network access) to communicate with application B (with network access) through covert channels that can be restricted based on a security policy. For example, if an application A has made a new entry in the address book, an application B should not be allowed to see that specific change.

The biggest challenge in this approach is to create a comprehensive security policy and to reduce the number of false positives.

7.4 Misbehavior analysis using power utilization patterns

Another misbehavior detection approach is to monitor and analyze the power consumption of applications running on battery powered smartphones. Typical applications have a specific battery depletion pattern and the basic assumption of this approach is that malicious applications would use a non-standard battery utilization model. The challenge is to model battery utilization behavior of benign applications accurately. The challenges and emerging issues of power based malware detection (presented in [53] and [49]) are: (1) accurately modeling the battery usage patterns according to a user's behavioral patterns on a smartphone; (2) monitoring the battery power usage in realtime is a difficult task because the precision of power measurement APIs varies on different smartphones; (3) frequent queries about battery's remaining capacity also load the CPU and discharge the battery; and (4) polymorphic variants of malware need to be detected to keep the false positive and false negative rates at an acceptably low level.

Different methodologies have been proposed to accurately measure and model power consumption. The well known are two: (1) *external measurement*, and (2) *on-device battery status APIs*. The external or physical measurement of power consumption requires external sensors and probes. As power is a product of voltage and current drawn over a unit time; therefore, both of them need to be monitored. Mostly battery maintains the voltage within an acceptable error range; as a result, only the current drawn needs to be sampled over time. One way is to do the measurement directly by intercepting a smartphone's battery circuit. An indirect way is to measure the magnetic field, produced by the current, using the well known phenomenon of Hall effect. In order to generate the power profile of an application, sampling by the power monitor needs to be synchronized with the real on-device activity. Most mobile operating systems also provide APIs for reading the current battery status. Using a timer, the battery status can be sampled to measure the power consumption of an activity or application. The problem, however, is a significant loss of accuracy. The results are reported as battery segments or remaining battery in percentage, with a granularity between 5 or 6 segments to 100 segments. Latest smartphones provide 1% resolution for battery status and it is only marginally acceptable.

A typical classification paradigm is: to compare the power consumption profile of a phone with that of the one logged during normal operations. If no malware is running, the power consumption profile should be similar to that of the logged one; otherwise, in case of a discrepancy an alarm about a malicious activity is generated.

7.5 System Performance/Behavior based Anomaly Detection

Another dynamic approach is to monitor a system's performance or behavior, and identify the anomaly in this behavior to detect malicious applications. The underlying assumption is: a system's performance parameters differ in a significant manner because of presence or absence of a malware activity.

In this technique, the goal is accomplished through periodic monitoring of different activities (including but not limited to): message activity, telephony activity, filesystem activity, CPU utilization, memory consumption, network activity, battery drainage, processes and threads, and other parameters relating to a system's health and responsiveness. These metrics are measured for normal user activities or in the presence of malware activity or both. The profile of a system behavior for normal and malicious activities is termed as benign or malicious profile respectively.

The common mechanism to present such profiles is through mapping points in a multi-dimensional space, a set of boolean or comparison rules, and probabilities etc. The profiles can be updated when a new malware is detected. The classification is done by applying data mining techniques on the profile. Typical machine learning algorithms used by researchers for malicious behavior detection on smartphones are: Bayesian Networks, K-Nearest Neighbors, Random Forest, Artificial Immune System, Radial Basis Function and Self-Organizing Maps.

The true merit of this technique is the ability to detect packed/encrypted malware. No unpacking/decryption is needed because the impact of an executing process on a system's behavior is observed. Moreover, it is possible to detect zero-day (unknown) malware that do the same malicious (as that of known malware) activity but by using different instructions, steps and processes.

The major challenge in this technique is: selecting thresholds for different performance parameters that discriminate normal and malicious behavior. Moreover, for anomaly detection, the complete picture of normal or anomalous behavior needs to be defined and then used in training. A benign process that is different from the processes used in training will be definitely classified as malware (a false positive); therefore, achieving low false positives is not an easy task.

After discussing the common static and dynamic malware detection techniques for smartphones, we now focus our attention to the existing tools and frameworks that utilize them. We restrict our discussion to the major tools and frameworks only, proposed in the recent years, with an aim to choose the representative tools for the above-mentioned techniques.

8. SECURITY AND PRIVACY ANALYSIS & DETECTION TOOLS FOR SMARTPHONES

In this section, a critical review of tools and frameworks for analysis and detection of malicious executables and programs on different smartphone operating systems is presented. A short description of different tools along with their analysis and detection methodology is discussed. Specifically we focus on classification efficiency, processing overheads, scalability issues, robustness and resilience against evasion techniques. Finally, we provide relative merits and demerits of a technique compared with others.

8.1 Woodpecker

Woodpecker [54] is a security tool that exposes privilege escalation attacks in Android based smartphone applications. It is capable of detecting both explicit and implicit permission leaks. The authors have defined explicit permission leaks as "use of the public interface of a privileged application by a non-privileged application to circumvent the operating system's permission model". On the other hand, implicit permission leaks happen when applications share permissions by using shared developer keys.

Woodpecker detects possible permission leaks in the installed (or to be installed) third party applications as follows. First, the Dalvik code is extracted from an application's executable and it is disassembled. Then, Woodpecker generates control flow graphs from the byte code. Since the application may have several entry points, control flow graphs are generated from each entry point. Woodpecker also takes care of indirect flows (such as thread *runnables* in Java) to maintain control flow connections.

Using control flow graphs, the capability leak is detected through identification of feasible paths that contain dangerous calls (permission dependent functions). For each entry point, the control flow graph is traversed and the feasibility of each path is computed. If a feasible path passes through any dangerous call, it is flagged as a capability leak. For implicit leak detection, Woodpecker crawls the manifest files of all applications and computes union of permissions for applications with shared identifiers (the applications developed by the same developer). Then, woodpecker uses the control flow graphs to detect if such a permission leak is being exploited by any application. In the case of implicit leaks, Woodpecker generates control flow graphs from entry points and looks at their initialization routines as well.

The authors tested Woodpecker tool on 8 different phones from 4 different vendors. The phones were selected to ensure diversity of tested applications on the Android platform. The authors selected a set of 13 privileged permissions which include: getting a user's location, making calls or sending messages, deleting user's data and recording user's conversations. Their analysis shows that 11 of these 13 permissions were explicitly leaked on the chosen phones. Some phones leaked nearly 8 of these permissions. This means that a third party application can perform a privileged operation by using a leaked permission without the need to ask for a permission from the user or the system. The authors claim that although a large number of possible paths with capability leaks are returned by the Woodpecker, the path feasibility calculation in the Woodpecker tool removes the false trails. The manual verification of the results confirm zero false alarms. The false negatives (in this case) are not reported, which is understandable as the authors don't have knowledge of all the permission leaks. However, the authors could have generated a test suite with different permission leaks, and then tested the tool on it to report any false negatives. Since the tool is designed to be used offline; therefore, the processing time of around 1 hour for each system image (with typically 100 to 150 applications) seems reasonable. If this tool is to be used online at the time of installing applications, the processing overhead needs to be significantly improved.

8.2 Static Function Call Analysis for Collaborative Malware Detection on Android OS

The authors in [37] have presented a framework for detection of malware on Android OS using static function call analysis of ELF files by using supervised data mining algorithms for classification. A framework is presented for collaborative detection of malware in ad hoc mobile networks. In a simulation environment, the collaborative framework produces significantly better results.

The framework has three components: On-device analysis, Collaboration and Remote analysis. The features are extracted through on device analysis. The data extraction is done at the OS level because the Java framework doesn't provide required APIs. A custom written tool *Interconnect Daemon* monitors the filesystem and operating system events. This daemon is responsible for identifying ELF executables on the system. It parses the executables, using the *readelf* tool, and creates a list of reference functions for each running executable on the system.

The list of functions used by an ELF executable are divided into six different attribute sets based on the type of a function (dynamic, relocation and a set of both) and its presence in malware and benign executables (mutually present, set of all functions). The machine learning classifiers – Prism, PART and Nearest Neighbor Algorithms (KNN) – are trained and tested on the dataset of features extracted from both benign and malware executables.

The features are analyzed using *WEKA* tool [55]. Approximately 100 benign ELF executables present in Android `/bin` directory, and 240 ELF malware executables found on Internet have been used for training and testing of classifiers. Using 10 fold cross validation, the classifiers are trained and tested on the features extracted from benign and malware executables in a dataset. The results show that Prism achieves zero false positives but has a low detection rate (approximately 70%) and also uses a large set of classification rules. In comparison, PART produces minimal rules set and has a detection rate of over 99%. However, it produces 12-16% false positives. The learning and classification can be performed on a remote server to reduce the processing load on a smartphone. However, the authors have not reported any memory and processing overheads on the system.

To reduce the false negatives, the authors have created a collaborative environment (an ad hoc network) where a node can request its neighboring nodes to help it in classifying malware. Using a threshold of uncertainty, the results returned from the neighboring nodes can be used for properly classifying malware. They have simulated an ad hoc mobile network to prove the claim that collaborative nodes help in reducing false positives. The authors also demonstrate that frequently collaborating nodes reduce the number of infected nodes in an ad hoc network; however, such collaborations lead to a fast depletion rate of the battery of a smartphone.

8.3 Static Function Call Analysis using Centroid - Symbian

A tool for malware detection through static function call analysis on smartphones has been presented in [38]. The framework has been implemented and tested on the Symbian OS. The tool classifies the applications into benign and malware on the basis of a clustering algorithm called *Centroid machine*.

The framework consists of two parts: (1) feature extractor and (2) centroid machine. The list of functions in an executable defines the features' set. The list of functions is statically extracted using IDA Pro[4]. In some cases, an executable might have to be unpacked before extracting features from it. Unpacking is done by using the UnSIS[5] tool on the Symbian platform. After unpacking and feature extraction, feature selection is performed by using statistical techniques. This helps in reducing the number of attributes and increasing the classification accuracy. A small set of attributes help in reducing the memory and processing overhead of the detection technique. The labeled training dataset is divided into the clusters of benign and malware datasets by using the centroid machine. The classification is performed by measuring the ratio of distance of a given application's attributes from the centroid of benign and malware clusters respectively. The application is classified as benign if its distance ratio – within a threshold that is adaptively determined – is smaller from the benign cluster compared with the malware cluster.

To test the framework, the authors collected 33 malware programs available on the Symbian OS and 49 benign popular applications available on the Internet. The experiments highlight the impact of features reduction on the the detection accuracy. Moreover, the authors have also compared the performance of centroid machine with Naive Bayes and Binary Support Vector Machine (SVM) algorithms. For statistical significance, the results have been averaged on 1000 runs, and 10-fold cross validation is used for training and testing.

The results show that the detection accuracy is only marginally affected by reducing the number of attributes from 3620 attributes (accuracy: 98.7%) to only 14 attributes (accuracy: 96.5%). For 14 attributes, the Naive Bayes and Binary SVM achieve accuracy of 90.2% and 91.9% respectively. The authors do claim that the centroid algorithm is efficient and light-weight (making it suitable for running on resource constrained smartphones) but they have not provided its empirical evidence by reporting memory and processing overheads.

[4]http://www.hex-rays.com/idapro/

[5]http://developer.symbian.com/main/tools_and_sdks/developer_tools/critical/unsis/index.jsp

8.4 PiOS

Another static analysis based tool PiOS is proposed by [56] on iOS platform to detect information leaks. It tracks information flow through static analysis of Mach-o executables. In this approach, the tool generates control flow graphs (CFG) from binary executables. In Objective-C, the function calls are bound to the function instances through the Objective-C runtime library; as a result, the function calls are replaced by messages calls (*msgSend*). To detect a correct function call, the authors have built a hierarchical structure to identify the derived and relevant base classes. Afterwards *backward slicing* is used to track the input parameters and their type to a dispatch function. In this way, the authentic targets of function calls are determined as a pre-requisite of CFG and it leads to successfully constructing CFG.

In the next phase, the framework performs the reachable analysis to determine the existence of paths (hierarchical function calls), which provide connectivity of the source of the sensitive data to their sinks – modules that provide connectivity and communication. This analysis on CFG determines the information leakage from iPhone devices to the third-parties or hosts. In the final phase, the data flow analysis is performed to validate it. Finally, PiOS generates the list of source-sink pairs for the analyzed information flows (the pairs represent different private information leakage scenarios). The tool allows a user to manually inspect the list of pairs that are linked even though no information flow is seen between them.

The authors have analyzed 1400 iPhone applications using PiOS. They report that other than few 'bad *apples*', most of the test applications didn't leak sensitive information. It is interesting to note that more than 50% applications secretly leaked the identity of the mobile device. This leak, coupled with profiling a user's smartphone usage pattern, has the potential of a privacy leak.

8.5 SmartDroid

Smartphone applications are typically highly interactive, which means that a malware can hide its activity from automatic malware detection tools by hiding the trigger for the malicious activity in complex user interactions. *SmartDroid* [57] is a framework that attempts to solve this problem by finding such user-interface (UI) interactions through a hybrid (static and dynamic) analysis of the application and simulating the triggering interactions. The static analysis module i.e. *static path selector (SPS)* determines the possible activity switches and function calls paths. It performs this analysis by disassembling the application and constructing function call graphs (FCG) and activity call graphs (ACG). All indirect and event-driven APIs are also included in the construction of CFG. The ACG is constructed through analysis of explicit and implicit Intent constructor calls. Dynamic analysis is performed to match the UI-elements with their related UI-event functions. A dynamic UI Trigger is created by modifying the Android framework and building a modified emulator. The UI-Interaction Simulator helps in performing dynamic analysis in an automated manner through traversal of the UI tree and simulating interactions with the UI-elements in activities.

The prototype of *SmartDroid* framework is evaluated using 19 malicious applications, belonging to seven different malware families. The authors report that SmartDroid is able to efficiently unveil the simple indirect UI-based trigger conditions. However, some of the complex indirect conditions (data based UI elements creation) for trigger are missed. The static analysis mean time is reported in the range of 5-16 seconds, while the dynamic analysis mean time is typically about half a minute per path. This framework is suitable only for offline analysis or decoupled analysis of the potentially malicious applications because it requires changes in the Android framework and relatively large analysis overhead. Moreover, the framework has only been tested on a very small and selected malware dataset.

8.6 Multi-Level Anomaly Detector for Android Malware (MADAM)

The authors of [50] have proposed a multi-level anomaly detection framework that detects malicious application through dynamic analysis in realtime using machine learning classifiers. As the framework uses anomaly detection instead of two class classification, it is supposed to be capable of detecting zero-day (previously unseen) malicious applications. The framework monitors the smartphone on two different levels: the kernel space and the user space. The system calls made by the smartphone applications are intercepted and logged in the kernel space using a kernel module. The list of running processes, the memory usage and the CPU utilization are also monitored in the kernel space. Moreover, a user's state (active or idle), key-strokes, called numbers, SMS, Bluetooth and WLAN activity is analyzed and logged in the user space. This multilevel view of the system events helps in a wider range of features (used for monitoring) and provides a correlated view of the events occurring at different levels.

The authors have employed k-nearest neighbors (KNN) (with k=1) algorithm for classification of the collected feature set. A dataset of 10 malicious and 50 legitimate applications on Android is used for evaluation purpose. The authors have divided classification process into three phases: (1) training phase, (2) learning phase, and the (3) operative phase. The framework is claimed to be doing anomaly detection but its classifier is trained using feature vectors of a normal user behavior and synthetically created malicious behaviors. In the learning phase, the classifier is trained for a user-specific behavior to estimate the false alarm rate (a decrease in FAR is reported from 26% to 0.1% as the analysis time increases from 10 to 240 mins respectively). The authors show that the framework can adapt by gradually adding new elements in the training set at run-time. Finally, in the operative phase, a classifier does anomaly detection on short-term (1 sec) and long-term (60 sec) timing windows. An average detection rate of 93% along with an average false positive rate of 5% is reported. The performance overheads are: 3% memory utilization, 7% CPU overhead and 5% battery depletion. The major shortcomings of the framework are: (1) high false alarm rate; and (2) evaluation on limited number of real malicious applications. The authors claim that the framework is capable of detecting zero-day malware.

8.7 Virus Meter - Battery Utilization patterns - Symbian

A misbehavior analysis and detection tool *Virus Meter*, which uses battery utilization patterns of applications, is presented and demonstrated for Symbian OS on Nokia 5500 smartphone [49]. Its core principle is to detect energy hungry applications [53]. The authors have profiled energy utilization of applications based on a user's activity (the duration of voice calls, frequency of sending/receiving text messages, usage & processing of documents, a system's idle state, entertainment and other activity benchmarks), and a system's performance/benchmark parameters – signal's strength & weakness and the network's activity and its state & conditions. It is argued that these conditions significantly effect the battery utilization behavior: for example a long voice call, frequently sending & receiving SMS and low signal strength results in more battery depletion. Three power calculation functions are approximated by using machine learning algorithms (linear regression, neural networks and decision trees) to compute the power consumption between two subsequent power measurements.

A state machine is defined to model a user's behavior that is previously defined. This state machine consists of a sequence of steps: (1) the power consumption monitoring application is installed on a clean (no malware) smartphone OS; (2) a known process, whose power consumption is to be measured, is launched; (3) the relevant events along with their characteristics are identified and recorded; (4) an association and correlation of events with the launched process is computed and relevant features of the events are recorded; and (5) finally, steps 1 to 4 are iterated to identify the sequence of common events.

Using the power models and state machine events, the difference between predicted power and the measured power is calculated. If abnormal patterns are observed between the two, the application is declared as malicious. Virus Meter uses linear regression model to detect misbehaving applications in realtime. The linear regression model reduces the processing overheads but realtime prediction might not be accurate due to oscillatory behavior of electro-chemical batteries. This might increase the false positive rate in misbehavior detection. The machine learning algorithms – decision tree and neural networks – provide better results and reduce the probability of false positives because they give a decision over time series collected data instead of individual instances. The machine learning algorithms consume relatively more processing and battery power; therefore, they are only used in the charging mode.

The framework is evaluated using known Symbian malware FlexiSpy and different variants of Cabir. FlexiSpy is basically a spyware, designed to do silent calls, interception of calls and SMS forwarding etc. Cabir uses bluetooth functionality to spread itself. The misbehavior (min-max) detection rate of the framework on detecting silent calls, calls interception and SMS forwarding is 85-93%, 66-90% and 89-98% respectively in both realtime and charging modes. On the other hand, Cabir's variants detection rate is 89-93%. Moreover, the false positive rate of the framework is 4-22% using the above mentioned predictor and classifiers. Furthermore, the processing overhead of the framework is 1.5% in terms of power consumption using linear regression. To conclude, the virus meter is not suitable for realtime deployment because of its high false alarm rate and inconsistent behavior of electro-chemical batteries (due to power fluctuations).

8.8 Energy-Greedy Anomalies & Malware detection - Windows Mobile

This framework uses signatures, derived from the power utilization history, to detect energy greedy malicious applications [53]. It is developed for HP iPAQ smartphone on a Windows Mobile operating system. The framework consists of two major components: (1) a power monitoring module and (2) a data analyzer module. The power monitoring module collects, analyzes and maintains power consumption – calculated by the product of instantaneous current and voltage over a fixed time – of different applications running on the phone. The power is precisely calculated, by using an oscilloscope with a hall effect probe, which measures the current drawn by the phone (since the voltage is constant; therefore, the power is directly proportional to the current drawn).

The malware can learn the sampling pattern and accordingly change their execution behavior; therefore, the authors use two different power calculation methods. In the first method, power samples are taken after a fixed period of time but the starting and ending points are randomly chosen – making the measurement interval random. In the second method, even the frequency of collecting samples is made random as well. The monitoring module also computes and stores a mean value of different power levels for each state of the smartphone e.g. on, backlight off, screen off and on, backlight on and screen on etc. Once the power consumption approaches near to the threshold level, the power monitor raises an alert and begins to store the energy utilization record. To achieve better accuracy, a higher sampling rate is preferred but that might lead to more power loss.

The data analyzer component extract patters from the collected samples and generates signatures. It uses a moving average filter to remove high frequency outliers in the sampled data and it uses a customized compression algorithm to reduce processing overhead of matching the generated signatures with the ones in the signature database. Newly installed or signature-less applications are generally misclassified.

Using the above-mentioned scheme, battery depletion attacks are detected with 100% accuracy – different programs (e.g. WiFi faker and dummy programs etc.) and their combinations are evaluated. The framework is tested using four mobile worms i.e. Cabir, Lasco, Commwarrior and Mabir (they belong to the same malware family). The authors have shown that the power signature of one malware can be used to detect other unknown (zero-day) malware of the same family. The accuracy to detect one worm varies between 93% and 100%. On the other hand, the detection accuracy of a family for worms – detected with the same signature – varies from 80% to 93% (and 100% only in the case of Mabir worm family). The false positive rate is approximately 2%. The processing overhead of the framework, in terms of battery utilization, is not reported and the robustness of battery based signatures is not analyzed. A crafty attacker can design a malicious application that consumes the same power utilization pattern as that of a benign application; as a consequence, it can successfully masquerade as a legitimate application.

8.9 Cloud-based Paranoid - Android

A dynamic and decoupled security solution *Paranoid* is presented in [58]. To avoid the resource extensive implementation of a security framework on a mobile host, the authors have suggested a cloud-based security framework for Android. They record and replicate the minimal instruction traces of Android's executing processes to a remote server in the cloud. On the server, the collected instruction set of processes' are replayed in the Android's emulated environment and a multi-layer forensic analysis is performed to detect malicious processes.

The flow of information among different components in the Paranoid framework is summarized. An instruction tracer records (using *ptrace* system call) program instructions and stores them into a secure storage, available on a mobile host. To protect the blocks of instruction traces from being tampered by any intruder or malicious software, a message authentication code (in conjunction with the hash key) is attached with every instruction block. To protect the previous blocks in the storage, key rolling approach (hashed MAC code of the previous key) is used to calculate the hash of the new (to be) used key. To avoid any failures (e.g. battery power problems), instruction traces are only synchronized with the cloud server when the smartphone is in the charging mode. To record and replay the complete execution process of a program, the recorded instructions and the input data – provided by the user through hardware – is required. Paranoid doesn't store the input data with the instruction blocks; therefore, a separate proxy server is implemented on the server side to fetch the data on demand. The instruction blocks and data are transferred from a mobile host to the cloud in a compressed form. An Android emulator is installed on the cloud server to perform the security checks during the replay of a program's instruction trace. The checks are: (1) code injection and buffer overflow attacks; (2) a signature-based on access scanning of files by the open source antivirus; (3) an emulator's memory scan for malicious codes; and (4) anomaly detection using the system call traces of the programs.

In the emulation environment of a cloud server, the malware detection accuracy and false positive rates are not reported explicitly. In general, the authors have emphasized the processing, storage, power consumption and replication overheads. They have collected data and process traces from real smartphone users' (more than 100 smartphones can connect to the cloud and replicate data.). It is reported that, most of the time, a smartphone remains in an idle state or is used for voice calls. The data rates in the idle and busy states are approximately 64-120 B/s and 2 KB/s respectively. The volume of collected data of voice call traces exceed 20 MB. Replicating data from a smartphone to the cloud server can have significant costs in terms of bandwidth hogging and the price charged by 3G operators. A better approach would be to store them in a local memory and use WiFi connections instead. The framework increases the processing load of a smartphone by more than 15% and battery depletion rate by 30%. Both overheads are quite significant.

8.10 Crowdroid - System calls based decoupled security

This framework employs dynamic monitoring of system calls of Linux kernel to detect malicious applications on Android. It uses decoupled malware detection approach to reduce processing and power overheads [59]. The authors have developed a smartphone client application – *crowdroid* – that uses crowd souring. It consolidates the log of system calls of different applications, running on multiple smartphones, and uploads the data on a remote server. This framework employs the *strace* tool for logging systems calls and ftp to upload the data to the remote server. The users share only behavioral data related to the used applications and not their confidential or personal information.

On the server side, the dataset is parsed by a data processing module that uses perl scripts to do behavioral analysis and generates different behavior vectors based on the system calls for each application. These vectors contain the information of accessed files, execution duration of processes, and a count of system calls used by the application programs. Afterwards, the k-mean clustering algorithm is applied for classification. The classification results for every individual application are stored in the database.

The authors have tested the framework using two real malware i.e. *PJApps* embedded in a steamy window application and *HongTouTou* trojan embedded in a monkey jump application. They obtained benign and infected android applications from known online repositories. They collected data from 20 users. The data contained 60 system call traces from benign and self synthesized malicious applications (50 benign and 10 malware). A 100% detection accuracy is reported for self-created malware. For realworld malware, authors have used 20 features' vector of benign and malicious applications. They have reported 100% detection accuracy for *PJApps* malware and 85% accuracy for *HongToutou*. The false positive rate is 20% in case of *benign-HongToutou* clusters. Such a high false alarm rate is not suitable for a real world deployable application. Experiments are needed to show the robustness of their features set against evasion. Moreover, the overhead of the proposed algorithm is not evaluated.

8.11 Knowledge-based temporal abstraction - Android

A lightweight, host based intrusion and malware detection security solution to detect unknown (zero-day) malware by monitoring time series measured data and events on Android phones is presented in [60]. This framework is adapted form of a knowledge-based temporal abstraction (KBTA) [61] [62] ontology for smartphones. KBTA is based on five different types of parameters: (1) primitive features (i.e. CPU usage, sent or received data packets over network interfaces etc.); (2) abstract features – they are derived from the primitives (i.e. percentage of CPU's busy state etc.); (3) the events are a form of raw data based on a user's (or system) behavior (the number of screen touch events, the number of applications launched and terminatied etc.); (4) the context is used to assign meaning to different type of parameters: the CPU state in a user's busy or idle state etc., and the classification function of parameters changes with respect to their context (they are further classified into different types: state (high or low CPU activity), trend (escalating or decreasing rate of camera activity) and rate (quantified rate of change of a feature value) etc.); and (5) the patterns are derived from parameters, their context and associated events by defining local and global timing constraints.

The framework consists of the following components: (1) feature manager, (2) agent service, (3) processors and (4) a graphical user interface. After a periodic interval, the feature manager module extracts features from different layers of OS. The processor unit is mainly an analysis and detection framework that consists of a machine learning classifier for misbehavior detection. It receives dataset from the feature manager and classifies it after doing relevant processing. Finally, it forwards the results to threat weighting units (TWU) that apply selection or summation algorithms – majority voting or distributed summation etc. The final results are forwarded to an alert service that applies a smoothing filter and min/max thresholds. The agent services module provides the organization and communication services to all components. The graphical user interface is used for configuring agents, warning alerts and visual exploration of the extracted data. The framework uses a fuzzy algorithm that works on a collection of constraints (instead of the classical signature based approach) to detect malware.

Different types of sample malware applications are used to test the framework: a game for information stealing (camera pictures), a tip calculator for denial of service, a malicious application to block outgoing calls, an information leakage (from SD card) application and a contact stealing and transmitting application. The frequency of data logging is selected from four intervals that vary from 2 to 14 seconds. The detection rate of applications (in the best case scenario of 2 sec sampling intervals) is 98-100% and detection time varies in between 5-32 seconds. If the sampling intervals are increased beyond 2 sec, the detection rate and time are only marginally effected. The authors have not reported false alarm rate even though they have intuitively presented scenarios that could lead to false alarms. Moreover, they have also not discussed robustness of their fuzzy algorithm against evasion attempts.

8.12 Andromaly

It is a host based anomaly detection security system for Android smartphones [42]. Its architecture is inspired from the knowledge base temporal abstraction framework [60]. The authors have used two Android HTC G1 smartphones that have 20 installed benign games. To incorporate behavioral changes, the phones were given to two different users. The authors used 3 features reduction methods – Chi-Square, Fisher Score and Information Gain – to rank the features (they created three datasets with 10, 20 and 30 top ranked features). Later they classified the datasets with K-Means, Logistic regression, Histograms, Decision tree, Bayesian networks and Naive Bayes. The authors preferred light weight classifiers to optimize power consumption.

They have created four different training/testing scenarios: (1) In the first scenario, training set consists of 80% benign and malware samples and the remaining 20% is used for testing; (2) In the second case, they trained on the dataset of 3 malware and 3 benign applications and tested on one malware and one benign application; (3) In the third scenario, the dataset of the first phone is used for training and the system is tested on the dataset of second phone; (4) In the last scenario, the dataset of three benign and malicious applications of one phone are used for training and the system is tested on one malware and one benign application of the second phone. The obtained results for four scenarios are: (1) In scenario 1, the decision tree outperformed other classifiers and provided 99.9-100% accuracy with 0% false positive rate (FPR); (2) In the second scenario, logistic regression provided the best results (86-90% accuracy with 12-11% FPR); (3) For the third and fourth scenario, Naive Bayes achieved 82-88% accuracy with 23-14% FPR and 75-85% accuracy with 29-17% FPR respectively. Andromaly uses 8.5% RAM of a smartphone and its processing puts 3.5-7.5% additional load on the CPU and its detection time is 5 sec. Overall, it degrades performance of smartphone by 10%. A very small set of malicious applications has been used for experiments and a more comprehensive evaluation is needed.

8.13 Behavioral Misuse Detection - iPhone

A misuse detection system based on a users' data logs on iPhone is proposed in [63]. The classical machine learning algorithms – random forest, bayesian networks, radial basis function and k-nearest neighbors – are applied to a dataset that contains a user's voice calls, text messages and Internet browsing history. The data is logged individually (for each individual application) as well as in a multi-modal fashion (combined data from all services).

To collect the dataset from 35 iPhone users, the authors designed a client-server that logs and stores the dataset. The voice calls records consist of the number, flags indicating incoming/outgoing calls, the time stamp and call duration. The features' set from text messages is derived in a similar fashion. Moreover, the browsing history is maintained by logging the web-link and time stamp parameters. The outcome data analysis is: (1) 66% users used their iPhones for Internet browsing and they hardly visit previously unknown URLs; (2) only 2% text messages were sent and received from new mobile numbers while other 98% were from known family and friend numbers (almost a similar trend is observed for voice calls).

The authors have used two different validation schemes: 10-fold cross validation and 66% data split. Using random forest classifier, the system achieved 99.8% true positive rate (labeled as sensitivity) and 0.3-0.4% false positive rate to detect misuse of calls, text message and browsing services. They claim that their detection rate is 1.2% superior compared with the previous solution [49]. The error rate and false negative rates are 1.6% and 0.7% respectively. The aggregated detection time varies in between 1.5 and 6 seconds for both validation schemes. The robustness of the used features set is not discussed by the authors.

8.14 TaintDroid

TaintDroid [45] is an information flow tracking tool for Android smartphones. It attempts to improve the visibility of sensitive data as it flows through the third-party applications and empowers a user to control its use. The tool is capable of tracking multiple sources of privacy sensitive data and provides dynamic taint tracking capability to the system. TaintDroid modifies the virtual execution environment of Android operating system. It applies taints (labels) to the sensitive data, tracks its flow and propagates associated taints. Finally, an alarm is raised if the labeled data leaves the system through an non-trusted third party application.

To efficiently track the information flow, TaintDroid applies labels at four granularity levels in the system: variable-level, method-level, file-level and message-level. The variable level labeling tracks intra application (running in a virtual environment) data flows. To track data flow in API calls, method-level tainting is performed. File-level taints are used to monitor storage and network I/O data flows. Interprocess communication data is tracked through message-level taints.

TaintDroid exploits spatial locality, to efficiently store and retrieve data labels, by storing data and labels adjacent to each other. The storage overhead is minimized by storing only one label for arrays. The authors have differentiated privacy sensitive data sources (taint sources) into four categories: high bandwidth sensors, low bandwidth sensors, information databases and device identifiers. The high bandwidth sensors include camera and microphone while low bandwidth sensors include accelerometer and GPS (location). The address book and SMS messages are stored in files or databases (termed as informational databases sources) and file level tracking is done for such sources. The information that identifies a user or a device is termed as device identifiers that include the phone number, IMSI, and device IMEI etc. TaintDroid considers the network interface as a taint sink. It looks for transmission of tainted data through the network interface.

TaintDroid has been tested on a set of thirty different applications. These applications were randomly chosen among the 50 most popular applications lists of 12 different categories. The test bed used was Nexus One phones running Android 2.1. During execution of thirty applications, information flow from sensitive data sources was tracked and logged. The experiment lasted for more than 100 minutes and over 22000 network packets were generated in the process. The packet trace on WiFi interface was logged using *tcpdump* that helped in validating the results.

TaintDroid correctly identified 20 different applications that were performing potential privacy violations. These violations included: sending device identifiers, phone information and location information to remote servers. The authors manually labeled the flagged TCP connections and packets. They reported zero false positives (although the possibility of missing false negative exists due to difficulty of identifying them). The authors report a 3% overhead in an application's load time. The overhead of propagating file taints is significant: address book modification (5.5% to create and 18% to edit). The call set up time increased by 10%. Taking a photo with the phone camera took 29% additional time. Overall, the authors report an average of 14% processing overhead in executing Java instructions and an average of 4.4% memory overhead. Interprocess communication was slowed by 27% with a memory overhead of 3.5%. The major limitation of TaintDroid is that it only looks for explicit information flow (through data); therefore, it is still possible to circumvent taint propagation through implicit flow of information.

8.15 AppInspector

Information flow tracking tools such as TaintDroid [45] can result in a large number of false positives as the outward flow of information may not necessarily indicate a privacy violation. The authors of TaintDroid [45] have recognized this problem and proposed a solution called AppInspector [64]. AppInspector is a tool that automates privacy testing and validation for smartphone applications. The proposed framework consists of four components: input generator, execution explorer, information flow tracking and privacy analysis tools. An application is installed in a virtual environment of the system. The automated inputs are supplied using input generator to provide maximum path coverage on the application. The authors propose a mixed of symbolic and concrete execution approach (termed as *concolic execution*) to provide better code coverage than random testing and execution. The execution explorer keeps track of the execution path; while information flow tracking keeps track of the flow of sensitive data and creates associated logs. The logging of the complete execution path – leading to a leak of sensitive data – might pinpoint the root cause of a privacy violation. The privacy analysis tools classify log activities as normal or violating privacy. The authors have proposed correlation of the logs with the privacy policy (EULA) of applications to automatically identify if flow of sensitive information violates a policy. The authors present the tool as a work-in-progress; therefore, its performance results are not available.

8.16 Android Application Sandbox (AASandbox)

AASandbox [40] is a tool for hybrid detection tool for Android smartphones. It detects malware by combining static and dynamic malware analysis techniques. Static detection is done by using malicious pattern searching technique. Afterwards, the application is installed and run on an emulator within a sandboxed environment and heuristics are used to classify the collected features' set.

In static detection, the application package (before it is installed) is decompressed and the binary executable and Android Manifest file are extracted. The manifest file contains important information such as permissions granted to an application at launch time. The executable (containing Dalvik code) is decompiled to get human readable code. Finally, the disassembled code is searched for malicious patterns from previously known malware. AASandbox searches for malicious patterns: malicious code blocks, malicious API calls, a combination of permissions, call to the native runtime environment, attempts to bypass the permissions, or attempts to use services or provisions that can quickly deplete the battery.

AASandbox performs dynamic malware detection through sandboxing within an emulator on a remote server. The Application Sandbox is installed on an Android emulator. It runs in the kernel space and hooks the system calls. When the application makes a system call, AASandbox intercepts the call, executes the original call and logs the call and results. Finally, it returns the results of the original call to the calling application in a user space. The Sandbox has been created as loadable kernel module (LKM) and can be installed on Android device at runtime.

AASandbox has been tested on a dataset of 150 most popular applications downloaded from the Android Market in Oct 2009. The authors created a custom malware (ForkBomb[6]) for testing the sandbox. The authors show that the system call logs of ForkBomb are significantly different from those of normal applications. AASandbox has not been tested on a large dataset and it does not use machine learning classifiers. The system can be implemented in a cloud environment that helps in providing decoupled security. The authors have not done any analysis of the time and memory overhead imposed by the detection system.

8.17 XManDroid: Framework for Mitigation of Privilege Escalation Attacks

XManDroid [65] is a security framework for Android that detects and prevents privilege escalation attacks at an application level. The framework detects transitive permission usage of applications at runtime and monitors interprocess communication through standard and covert channels. The detection and prevention of permission leakage is governed through a system-centric security policy.

Android operating system implements a middleware on top of a customized Linux kernel. This middleware manages the installation, configuration, service provision, permission management and interprocess communication for the Android applications. XManDroid framework consists of an application installer, a policy installer and a runtime monitor. It maintains a system-wide view that correlates information from the three components and represents the complete picture of a system in the form of a graph. The application installer updates the system-wide view

[6]ForkBomb replicates itself indefinitely to create sub-processes in an infinite loop that results in a denial of service attack.

with the permissions information of new applications. The system policy installers keep the system view updated by adopting the latest security policy. The authors have designed a security policy which consists of rules based on the permission contexts of the communicating applications. The runtime monitor verifies each interprocess communication call against the security policy. A reference monitor implements mandatory access control for interprocess communication between the applications. XManDroid extends the Android's reference monitor to detect and prevent interprocess communication calls that violate the defined security policy. The reference monitor applies the security policy to direct Intent calls as well as pending and broadcast Intents. A decision maker component helps the runtime monitor in the verification process.

The XManDroid framework also detects and prevents the use of covert channels for permission leaks. It performs this task by monitoring the content providers and the system services that can be used as covert channels. It maintains a track of reads and writes to the content providers and disallows any read from a content provider that had a previous write operation such that the $< read, write >$ pair could result in violating the security policy. Similarly, a consecutive $< write, read >$ pair also constitutes a violation of the security policy.

The authors tested XManDroid on a Nexus One phone running Android 2.2.1 kernel. A set of 50 benign applications has been collected from Android store. The authors tested their framework on a custom developed set of malicious applications. These applications employ seven different attack scenarios for privilege escalation. The authors have also developed a security policy – consisting of 7 different rules – for these scenarios. All attack scenarios were successfully detected by the framework. This is not surprising because the custom malware, developed by the authors themselves, might have resulted in an overfit. Other than automatic testing, the authors have also arranged a manual testing by a group of 20 students. The authors report an average of 3% false positives (incorrectly denied interprocess communication requests). This rate is relatively high for smartphone users and the system policy needs to be optimized to reduce the false positives. The authors report an average latency of 13.13 ms for interprocess communication requests that are not found in the cache and 0.11 ms for the ones found in the cache. The latency appears to be high if we consider the fact that typical runtime for interprocess communication is on the average 0.184 ms. The authors performed a usability test with a group of 20 students and report that users noticed degraded performance of the system but it was nevertheless usable. XManDroid has small memory overhead (maximum memory usage of 4 MB) and it is possible to trade more memory for achieving reduced latency.

8.18 Quire

On a smartphone, different applications can communicate with one another and depute tasks through publicly defined interfaces (e.g. Intents on Android). This allows an application to launch a confused deputy attack by improperly calling another application's interface and forcing it to use its privileges. Moreover, any application (with network permissions) can create an outgoing connection to a remote service; as a result, the *id* of the source of a network connection can be obfuscated. Quire [66] has been implemented on the Android operating system and it solves two problems: (1) inappropriate use of an application's permissions through its interface; and (2) trusted communication between applications by using network RPCs (remote procedure calls).

Quire enables an application or a service, receiving an interprocess communication (IPC) message, to be able to see and verify the entire path of an IPC chain. Using this information, the application/service can protect sensitive information from being misused; as a result, an application without privileges cannot trick an application with the sensitive data) to disclose its data. Quire uses cryptographic message authentication codes (MACs) to protect the integrity of the data across IPC and RPC channels. The keys are shared using a trusted OS service (Authority Manager). This enables the operating system to verify the authenticity and integrity of the network RPCs on behalf of the application.

To demonstrate the effectiveness and usefulness of Quire, the authors have developed two applications. The first application uses Quire as a click fraud protection mechanism for an advertising system. Instead of bundling the advertising system as a part of the application code (allowing modification of advertisement library and click frauds), the authors create a child advertisement application and then extend the UI layer to deliver UI events (such as clicks) to the child application. The two applications are stacked such that the transparency in the parent application allows a user to see the advertisement in the child application. The user events are transmitted directly to the child application which can leverage Quire to verify the click event, its source and its freshness (using a time stamp). This allows an advertisement service to ensure that only legitimate clicks result in generating revenues for an application's publisher.

The second application demonstrates the use of Quire in a micro-payment application (PayBuddy) for verifying the involved parties. A remote service – using Quire's message authentication code mechanism for IPCs and RPCs, and secure network connection (https) – can verify: (1) the authenticity and integrity of the original application's order; (2) the fact that the PayBuddy application approved the order (so an explicit approval of a user needs to be processed); and (3) the request originated from a particular device that has been issued with a certificate. This allows distrusting parties (original application, PayBuddy application and the remote payment service) to communicate with each other and validate the sequence of events with the help of Quire; as a result, a user's consent and the authenticity of the placed order is verified.

The authors evaluated the performance overheads of Quire framework on Nexus One phone with 1 GHz processor and 512 MB RAM. The message signing overhead is $20\mu s$ plus $15\mu s$ per kilobyte. The message verification takes $556\mu s$ plus $96\mu s$ per kilobyte that is significantly high. The overhead of tracking call chain is around $100\mu s$ per hop which is insignificant for small number of hops, but might become significant for large number of hops (hops are usually few). The RPC calls have an overhead below 5 ms even for a chain of 8 distinct applications and this is reasonable. One drawback of Quire is that it doesn't track two malicious applications that are collaborating to circumvent permission restrictions. The tool is only able to detect that a benign application is being misused by a malicious application.

8.19 Stowaway - Android Permissions Demystified

Stowaway is a tool to detect over-privileged compiled applications on Android platform [67]. The authors have extracted the API calls, invoked by an application, and correlated them with the permissions used or acquired by the application. Each application must use a set of least permissions according to its invoked APIs. But applications become over-privileged due to careless unsafe code written by developers. *Stowaway* builds permission maps to determine content providers and their intents; as a consequence, it is able to determine the privileges that are mandatory for an application to execute successfully.

The *DEX* executables are decompiled using a known disassembling tool *dedexer* and are provided to *Stowaway* tool. The tool extracts permissions from both executables and the manifest. It also tracks and extracts standard API methods. Afterwards, it separates the user defined classes that inherit methods from Android classes. The authors have used different heuristics to handle the problems of Java reflection: (1) the return value of an object is mapped with input parameters; and (2) the type of variable is determined dynamically at the time of its usage. Moreover, the applications which try to access a web address must have Internet privileges. Furthermore, in the Android environment, SD card write permissions are implemented in the kernel; therefore, in *Stowaway* tool, SD card access is identified by searching *sdcard* string in an application's strings and xml files or by checking if the APIs return path to the *sdcard*. The authors have employed a third party tool – ComDroid [68] – to analyze the permissions of sending and receiving intents.

A dataset of 964 Android applications has been used for evaluating the system. Initially, 24 randomly chosen applications are used for training the *Stowaway* tool. 40 out of remaining 940 applications were analyzed using the tool. It labeled 18 i.e. 45% applications as over privileged with 42 extra permissions. Afterwards, these applications are manually inspected by the authors. It is identified that 17 applications with 39 permissions are over privileged and the tool's false positive rate stands at 7%. In the remaining 900 applications, a set of 323 applications (35.8%) are marked as over privileged by the tool.

8.20 Categorization of Android Applications using static features (CAASA)

This framework uses machine learning algorithms to classify different applications on Android smartphones [43]. The tool extracts the features from Android applications by decompressing them with *Android Asset Packaging Tool* and using the files contained in *.apk* file: (1) the existence frequency of printable strings (known methodology of x86 platform for malicious executables detection [69]); and (2) the permissions set owned by the application along with its obtained rating from the Android market.

The extraction component uses *AndroidManifest.xml* to extract permissions and features of the device invoked by the application. Afterwards, the *dedexer* tool is used to disassemble the executable. As a result, a directory structure is produced with recognized classes. Finally, the strings are extracted from the application. The *term frequency* and *inverse document frequency* are calculated for the extracted strings. The authors have obtained the market information by using an open source API *android-market-api* that provides features: the number of ratings

Table II. Correlation Matrix of Tools and Features

	Stowaway (8.19)	CAASA (8.20)	Woodpecker (8.1)	SFCMD (8.2)	PiOS (8.4)	Centroid (8.3)	Virus Meter (8.7)	EG Anomalies (8.8)	Paranoid (8.9)	Crowdroid (8.10)	KBTA (8.11)	Andromaly (8.12)	Misuse Detector (8.13)	TaintDroid (8.14)	AppInspector (8.15)	AASandbox (8.16)	XManDroid (8.17)	Quire (8.18)	MADAM (8.6)	SmartDroid (8.5)
Mobile Platform																				
Android	✓	✓	✓	✓					✓	✓	✓	✓		✓	✓	✓	✓	✓	✓	✓
iOS					✓								✓							
Symbian						✓	✓													
Windows Mobile								✓												
Environment																				
On-Host (4.1)	✓	✓	✓	✓	✓	✓	✓	✓			✓	✓			✓		✓	✓	✓	✓
Decoupled (4.1)				✓					✓	✓			✓			✓				
Kernel Mode (4.2)											✓			✓	✓	✓	✓	✓	✓	
User Mode (4.2)	✓	✓	✓	✓			✓	✓	✓	✓	✓	✓	✓			✓		✓	✓	
Sandboxing (4.3)															✓	✓				
Emulation (4.3)									✓							✓				✓
Analysis Type (4.4)																				
Static Analysis	✓	✓	✓	✓	✓	✓										✓				✓
Dynamic Analysis							✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
Hybrid Analysis																✓				✓
Classification																				
Anomaly Based (4.5)						✓	✓		✓	✓	✓	✓							✓	
Two Class Classification (4.5)		✓		✓		✓	✓						✓							
Signature verification								✓	✓											
Control flow Graphs			✓		✓															✓
Features for Analysis																				
Static Function Calls				✓	✓	✓														✓
System Calls	✓									✓						✓			✓	✓
File Features										✓										
Info Flow Track					✓									✓	✓					
Text Strings		✓																		
Applications Permission	✓	✓	✓													✓	✓	✓		
OS & File Sys Events							✓	✓			✓	✓	✓						✓	
Instruction Traces									✓											
Concolic Execution															✓					
Power Utilization							✓	✓												
IPC monitoring & authentication																	✓	✓		

of an application, the number of times an application is installed, and the user ratings of the application.

In the classification phase, the authors have employed four well known machine learning classifiers: bayesian networks, C4.5 decision tree, k-nearest neighbor and support vector machine. A ten-fold cross validation is used for training and testing. The proposed scheme is empirically evaluated using 820 Android applications belonging to 7 distinct categories. The authors report the accuracy of the system by using area under the ROC curve (AUC) measure. The Bayesian network classifier provides the best accuracy with an AUC of 0.93. The authors have claimed in the paper that the proposed scheme is suitable for malware detection but they haven't performed any experiments on malware samples. The authors have not reported processing and memory overheads.

To conclude, in this section, we have analyzed the existing tools for detecting malicious applications. In Table II, we present a summary of malicious application detection techniques for a quick reference.[7]

Acknowledgments

The work presented in this paper is supported by the National ICT R&D Fund, Ministry of Information Technology, Government of Pakistan. The information, data, comments, and views detailed herein may not necessarily reflect the endorsements of views of the National ICT R&D Fund.

9. CONCLUSION

Smartphones are becoming the core delivery platform of ubiquitous "connected customer services" paradigm; as a consequence, they are attractive targets of intruders (or imposters). Researchers have realized that classical signature-based anti-malware techniques are not capable of providing efficient and effective detection tools against novel, zero-day and polymorphic malware for resource constrained smartphones; therefore, in last couple of years unconventional (non-signature) intelligent solutions, based on behavioral analysis (static or dynamic) have been proposed. In this survey, we have enumerated various types of malicious applications and the infection vectors that are a threat to security and privacy on smartphones. Creating an application for malicious applications detection requires some important challenges to be met and making crucial implementation decisions and we have enumerated these challenges and decisions. A generic framework has been presented for detection of malicious applications on smartphones that helps a reader understand the system wide architecture of malware analysis and detection techniques. We have also presented, analyzed and categorized latest published techniques utilizing static and dynamic detection techniques on smartphones. Finally, we have reviewed the recent malware detection tools and frameworks that utilize these techniques. We hope that the survey will help mobile malware researchers and practitioners to understand the existing state-of-the-art detection techniques and use them to propose a comprehensive zero-day malware detection framework.

[7]Brief analysis of some other relevant techniques is presented as a supplement to this paper in Appendix.

SUPPLEMENT: APPENDIX A

10. MISC. SECURITY & PRIVACY SOLUTIONS FOR SMARTPHONES

The authors of [70] have proposed a secure operating system framework that wraps primary smartphone OS in a virtual machine to establish a seclusive environment. The renowned micro-kernel L4 is used for this purpose. The framework runs applications – demanding high security – in parallel with JVM.

In [71], a tool – *DroidMoss* – is developed to detect packed applications that are launched at six unofficial android application markets. Some of these applications (5 to 13%) are simply repacked versions of the applications that are already available at the official Android market. The authors concluded that repacking is used to replace existing advertisements with new ones to earn revenues in an illegitimate fashion. *DroidMoss* employs a fuzzy hashing technique to compute similarity score of different applications and use it to flag packed applications.

RGBDroid [72] is a response based tool that can detect privilege escalation attempts by identifying processes that try to obtain illegal root privileges and restricts access to protected resources. The tool restricts illegal activities *after* a security breach that results in reducing the processing overheads which are typically caused by protective approaches.

In [73], the authors have proposed *DroidScope* which is a multilevel semantic analysis tool that performs dynamic profiling and information tracking to detect malicious behavior and privacy leaks in Android based smartphone applications. The tool runs in a virtualized environment and logs instruction traces, API calls (at OS level and Dalvik VM level) and uses taint analysis to discover leak of sensitive information. The tool has been tested on two real world malware samples.

A security solution *I-ARM-Droid* is proposed in [74] for protecting smartphones from malicious or non trusted applications. The framework monitors the APIs and associated security polices to identify the violations. Later, the Dalvik byte code is rewritten that interpolates the existing methods to enforce the least required permissions or security policies. In case a method in class C is to be interposed, the framework generates a class W (wedge class) that extends the class C. The new W class contains the stub and wedge methods that correspond to the target methods. The extended classes from class C are identified and modified to extend the W class. The authors have demonstrated the compatibility of their rewritten code with the Android platform. They have reported 110 ms average processing overhead in rewritten applications while the size is increased by 2%.

Jana [75] platform sandbox aims at addressing a users' privacy concerns. Its distributed architecture safely transfers a user's sensitive information with dependable privacy protection mechanisms; as a result, the responsibility of providing privacy is shifted from applications to the framework that creates a sandbox for every application run by each user – spanning from smartphones to the systems on a cloud. The sandbox provides dedicated communication and storage channels, with customized privacy preserving methods, to enable smartphones applications to do useful tasks. The authors have developed a prototype to demonstrate different features of Jana.

In [76], a security framework for Android OS is presented that takes care of *confused deputy processes* – the vulnerable interfaces of some privileged or over-privileged application processes that are misused by malicious processes to perform their covert activity – and *collusive attacks*: malicious processes embed them into benign applications to misuse them to perform malicious activities not allowed by their personal permissions. The authors have built a cross-layer, system-oriented and policy-based architecture that scrutinizes the communication channels in realtime among applications. At an intermediate layer, direct and indirect (using OS components) inter-process communication (IPC) is controlled and *QUIRE-like* links are setup between communicating processes to validate the call chain by using reference monitors. Moreover, the authors impose access control procedures on the file system and all types of sockets in the kernel. A feedback channel – setup between the kernel and an intermediate functional layer – implements the policy at a lower level. Finally, the effectiveness of the framework is determined with the help of an application that launches realtime attacks.

A static analysis solution to detect over-privileged applications (on Android platform) with the help of the permission gap – difference between granted and needed permissions – is presented in [77]. The proposed tool is tested on two datasets: it detects 13% of 742 (dataset 1) and 5% of 679 (dataset 2) Android applications that suffer from a privileges gap. A significant amount of research is focused on detecting over-privileged applications on smartphones (an interested reader can refer to [78], [54], [41] and [79]).

An interesting article [80] provides a survey regarding smartphone users' behavior and their awareness about the role of permission settings in Android during the process of installing applications. Unfortunately, the Android's permission system significantly depends on the risk-awareness of its users: they can install an application if they agree with the demanded permissions at the install time; otherwise, they have an option to abort the installation. The authors conducted a survey of 308 online users and 25 users in their laboratory asking them about their sensitivity, knowledge and conduct towards permission settings during the process of installing applications. It is amazing to see that only 17% users paid attention to the permission during installation and a meagre 3% could only correctly answer three basic questions involving comprehension of permissions. The conclusion is: the Android's permission mechanism is inappropriate for a vast majority of Android users because it fails in enforcing correct privileges. Consequently, they suggested improvements for a better usability experience.

A runtime verification system for Android is proposed in [81]. It maintains a profile of suspicious applications and if the profiles of running applications match with the suspicious one, an alarm is raised. Take the example of an application that gets executed at the boot time, it requests the location from GPS, and finally, it connects to the Internet to send the mobile's location to a remote host. In this case, the framework automatically activates a monitor to generate an alert, if the sequence of such events is followed by any user application.

An OS service *MoRePriv* [82] provides omnipresent personalization like the location service support and should be implemented in the kernel space instead of a user space. The service parses smartphone users' information streams over the Internet – a users' email, SMS, social networking database and usual network communication information flows etc. – and builds a user's profile to preserve his privacy by providing filter hooks to protect information leaks. *MoRePriv* empowers a user to organize his preferences in different user applications by exposing relevant personalization APIs. Moreover, the service also enforces privacy constraints for advertisements in the applications on the basis of a user's preferences profile. The authors have conducted experiments to demonstrate that *MoRePriv* helps in reducing over-permissions in 73% of tested user applications.

Advertisements are (mostly) an integral part of smartphone applications to generate revenue. To embed advertisement services, binary libraries are shipped with the smartphone applications. The binary libraries require execution permissions and demand host applications to share their sensitive information with them. In [83] and [84], the authors have reported that 49% Android applications contain one binary library (on the average) for advertising services and 46% applications are able to subscribe over-permissions because of these libraries. They also show that 56% applications (having embedded libraries) also request the location information without the consent of a user. The authors have developed *AdDroid* that filters and isolates advertisement libraries and their requested permissions; as a result, they can show advertisements without tricking a user to enable sensitive permissions or share privacy related information. The other approaches that also isolate advertisement libraries (and their requests for sensitive information) from normal user applications are presented in *AdSplit* [85], [86] and [87].

11. RELATED WORK

In this section, we provide a brief summary of recently published surveys.

11.1 Other surveys on Smartphone Security

A survey paper on security of mobile devices is published by [88]. In the paper, the authors have briefly discussed mobile wireless technologies: GSM, GPRS and EDGE along with networking technologies WLAN and bluetooth etc. The attack vectors – wireless interface, buffer over flow, network infrastructure, virus and worms, user behavior, privacy, denial of service, battery depletion and over billing attacks etc. – on mobile devices have been discussed. They have classified attack detection methodologies into five different categories on the basis of detection principles, architecture of security solutions, threat detection modes, data collection strategies and mobile operating systems. These models are briefly summarized here: (1) detection principles used anomaly based algorithms, machine learning classifiers, algorithms modeling energy consumption, conventional signature based models etc; (2) the architecture defines the point of deployment of the technique: host-based or distributed solution in the cloud; (3) the mode determines the countermeasures will be active or passive; (3) detecting intrusions by training the framework on different types of datasets: system calls dataset, system performance counter and keystrokes etc.; and (4) finally, the security solution is developed for a particular mobile operating system – Android, iOS, Symbian etc.

In [89], the author has presented different detection techniques for smartphones. The techniques are classified on the basis of their protection mechanism and application analysis. The techniques for system and applications' policies, platform security (e.g. visualization etc.), multiple user access and information faking are categorized as "protection mechanisms". The dynamic, static, permissions and cloud based analysis approaches have been summarized as "application analysis".

In another survey [90], the authors have categorized mobile malware detection techniques into three topologies: (1) device based detection; (2) infrastructure based detection; and (3) hybrid topologies. In the first category, they discussed behavioral and access control models on mobile hosts. In the second category, a mobile host tracks and logs the communication messages (or packets) and transfers them to a remote proxy server for detecting malware on a single or multiple mobile devices concurrently. In the hybrid topology, malware detection is done in a distributed manner by placing the security solution on host and infrastructure; as a result, the processing overheads on a smartphone are minimized.

The authors of [91] have provided a brief overview of security challenges on different smartphone interfaces (mobile network security problems: attack vectors and vulnerabilities using browsers and backup mirror servers, DOS attacks etc.). Some vulnerabilities are hardware dependent (like smart card) and some are generic and platform independent: link encryption, connectivity and handshakes, SMS and MMS vulnerabilities etc. The examples of software-based vulnerabilities are: malicious software, identity theft, browsers problems, mobile botnets, SMS and MMS vulnerabilities etc. In another survey of mobile malware [92], a behavioral analysis of available mobile malware and existing protection mechanisms and emerging trends are discussed. They have focused their survey on root exploits for Android. They have also described different exploits and how they could be used in combination with root exploits.

In survey of [93], the authors have summarized industrial and academic research about different paradigms for detecting mobile malware: (1) monitoring power consumption patterns for benign and malicious applications; (2) using a dual mode approach (different addresses for APIs in development and execution stages) for smartphone applications is proposed. Moreover, they have proposed the idea of hardware sand-boxing (whenever a user is busy making a voice call, the hardware modules needed for Internet access must be disabled.)

The survey [94] briefly summarizes different paradigms for smartphones. But, two methodologies have been discussed at depth:(1) signature-based malware detection; and (2) detection by de-obfuscating the obfuscated code. The other relevant surveys are [95] and [15].

Most of the published surveys have not considered the subject of security on smartphones in depth and breadth demanded by ever increasing complexity and sophistication of smartphone vulnerabilities. (Most of the surveys focus only on a specialized area – infrastructure base security – and ignore highly relevant security and privacy threats. In this survey, we have attempted to cover the complete spectrum of challenges in mobile security: mobile malware and their types, infection vector and vulnerabilities, challenges for smartphone security systems, comprehensive analysis of malware and privacy leaks detection, security tools and their features relevant to the OS platform, detection accuracy and the false alarm rate of an approach, and overheads in terms of processing.

The major contributions of this survey paper are: a focus on the niche of malicious applications detection related to security and privacy on smartphone platforms, generalizing a comprehensive malicious smartphone applications detection framework as a guideline for future development by vendors and security researchers, a comprehensive analysis of recently proposed techniques utilized by smartphone security and privacy analysts, and a survey of recently proposed tools using these techniques, along with their significance and shortcomings.

REFERENCES

1. Gartner, "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth," *http://www.gartner.com/it/page.jsp?id=1924314 [last-viewed-July-2-2012]*, 2012.

2. ——, "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth," *http://www.gartner.com/it/page.jsp?id=2017015 [last-viewed-July-2-2012]*, 2012.

3. A. Gupta, R. Cozza, C. Milanesi, and C. Lu, "Market share analysis: Mobile devices, worldwide, 2q12," *http://www.gartner.com/resId=2117915 [last-viewed-November-05-2012]*, 2012.

4. Product-News, "The new version of kaspersky mobile security provides maximum protection against malware and web-borne threats," *Caspersky Labs*, 2012.

5. NDTV-Gadgets, "Smartphones under malware attack: Symantec, mcafee," *[Online] http://gadgets.ndtv.com/mobiles/news/smartphones-under-malware-attack-symantec-mcafee-232792 [last-viewed-July-2-2012]*, 2012.

6. Juniper-Report, "Mobile threats report," *Juniper Networks Inc.*, 2012.

7. p. Paul Wood, G. Egan, K. Haley, T. Tran, O. Cox, H. Lau, C. Wueest, D. McKinney *et al.*, "Symantec internet security threat report trends for 2011," *Symantec Corporation*, vol. 17, 2012.

8. Mobile-Threat-Report, "Mobile threat report q1 2012," *F-Secure Labs*, 2012.

9. Trend-Micro, "3q 2012 security roundup: Android under siege: Popularity comes at a price," *Research and Analysis*, 2012.

10. Best-AntiVirus, "The 3 best antivirus apps to protect your android security," *http://www.makeuseof.com/tag/3-best-antivirus-apps-android-security/ [last-viewed-July-3-2012]*, 2012.

11. McAfee-Threat-Report, "Mcafee q1 threats report finds significant malware increase across all platforms," *http://www.marketwatch.com/story/mcafee-q1-threats-report-finds-significant-malware-increase-across-all-platforms-2012-05-23 [last-viewed-July-4-2012]*, 2012.

12. Y. Zhou and X. Jiang, "Dissecting android malware: Characterization and evolution," in *IEEE Symposium on Security and Privacy.* IEEE, 2012, pp. 95–109.

13. C. Miller, D. Blazakis, D. DaiZovi, S. Esser, V. Iozzo, and R. Weinmann, *IOS Hacker's Handbook.* Wiley, 2012.

14. D. Damopoulos, G. Kambourakis, and S. Gritzalis, "isam: An iphone stealth airborne malware," *Future Challenges in Security and Privacy for Academia and Industry*, pp. 17–28, 2011.

15. A. Schmidt and S. Albayrak, "Malicious software for smartphones," *Technische Universität Berlin, DAI-Labor, Tech. Rep. TUB-DAI*, vol. 2, 2008.

16. A. Bose, X. Hu, K. Shin, and T. Park, "Behavioral detection of malware on mobile handsets," in *International conference on Mobile systems, applications, and services.* ACM, 2008, pp. 225–238.

17. P. Szor, *The art of computer virus research and defense.* Addison-Wesley Professional, 2005.

18. A. Gostev, "Mobile malware evolution: An overview, part 1," *www.viruslist.com*, 2006.

19. P. Porras, H. Saidi, and V. Yegneswaran, "An analysis of the ikeeb (duh) iphone botnet (worm)," `http://mtc.sri.com/iPhone/`, 2009, accessed: August 08, 2012.

20. M. Adeel and L. Tokarchuk, "Analysis of mobile p2p malware detection framework through cabir & commwarrior families," in *IEEE International Conference on Privacy, Security, Risk and Trust.* IEEE, 2011, pp. 1335–1343.

21. F-Secure, "Threat description: Virus:w32/duts.1520," `http://www.f-secure.com/v-descs/dtus.shtml`, 2012, accessed: August 08, 2012.

22. Symantec, "Androidos.fakeplayer," http://www.symantec.com/security_response/writeup.jsp?docid=2010-081100-1646-99, 2012, accessed: August 08, 2012.

23. B. Krebs, "Zeus trojan author ran with spam kingpinskrebs on security," *Krebs on Security*, 2012.

24. D. Maslennikov, "Sms trojans: all around the world," http://www.securelist.com/en/blog/208193261/SMS_Trojans_all_around_the_world, 2011, accessed: August 08, 2012.

25. F-Secure, "Threat description: Trojan:symbos/skulls.d," http://www.f-secure.com/v-descs/skulls_d.shtml, 2007, accessed: August 08, 2012.

26. J. Bickford, R. O'Hare, A. Baliga, V. Ganapathy, and L. Iftode, "Rootkits on smart phones: attacks, implications and opportunities," in *Workshop on Mobile Computing Systems & Applications*. ACM, 2010, pp. 49–54.

27. Symantec, "System infected: Zeroaccess rootkit activity," http://www.symantec.com/security_response/attacksignatures/detail.jsp?asid=24377, 2012, accessed: August 08, 2012.

28. MilaParkour, "Contagio mobile - mobile malware mini dump," http://contagiominidump.blogspot.com/, 2012, accessed: September 23, 2012.

29. Fortinet, "Adware/sslcrypt!symbos," http://www.fortiguard.com/av/VID2715273, 2011, last accessed: August 08, 2012.

30. Mcafee, "Virus profile: Android/apphnd.a," http://home.mcafee.com/VirusInfo/VirusProfile.aspx?key=790111, 2012, accessed: August 08, 2012.

31. K. Haataja, K. Hypponen, and P. Toivanen, "Ten years of bluetooth security attacks: Lessons learned," *Computer Science I Like*, p. 45, 2011.

32. A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Annual Computer Security Applications Conference*, 2007, pp. 421–430.

33. Y. Guo, L. Zhang, J. Kong, J. Sun, T. Feng, and X. Chen, "Jupiter: transparent augmentation of smartphone capabilities through cloud computing," in *ACM SOSP Workshop on Networking, Systems, and Applications on Mobile Handhelds*. ACM, 2011, p. 2.

34. M. Chandramohan and H. Tan, "Detection of mobile malware in the wild," *Computer*, 2012.

35. Farrukh Shahzad, M. Saleem, and M. Farooq, "A hybrid framework for malware detection on smartphones using elf structural & pcb runtime traces," *Tech. Report TR-58 FAST-National University, Pakistan*, 2012.

36. Farrukh Shahzad and M. Farooq, "Elf-miner: using structural knowledge and data mining methods to detect new (linux) malicious executables," *Knowledge and information systems*, vol. 30, no. 3, pp. 589–612, 2012.

37. A. Schmidt, R. Bye, H. Schmidt, J. Clausen, O. Kiraz, K. Yuksel, S. Camtepe, and S. Albayrak, "Static analysis of executables for collaborative malware detection on android," in *IEEE International Conference on Communications*. IEEE, 2009, pp. 1–5.

38. A. Schmidt, J. Clausen, A. Camtepe, and S. Albayrak, "Detecting symbian os malware through static function call analysis," in *International Conference on Malicious and Unwanted Software*. IEEE, 2009, pp. 15–22.

39. M. Masud, L. Khan, and B. Thuraisingham, "A scalable multi-level feature extraction technique to detect malicious executables," *Information Systems Frontiers*, vol. 10, no. 1, pp. 33–45, 2008.

40. T. Blasing, L. Batyuk, A. Schmidt, S. Camtepe, and S. Albayrak, "An android application sandbox system for suspicious software detection," in *International Conference on Malicious and Unwanted Software*. IEEE, 2010, pp. 55–62.

41. C. Gibler, J. Crussell, J. Erickson, and H. Chen, "Androidleaks: Automatically detecting potential privacy leaks in android applications on a large scale," *Trust and Trustworthy Computing*, pp. 291–307, 2012.

42. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, pp. 1–30, 2012.

43. B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, and P. Bringas, "On the automatic categorisation of android applications," *Consumer Communications and Networking Conference*, 2012.

44. D. Barrera, H. Kayacik, P. van Oorschot, and A. Somayaji, "A methodology for empirical analysis of permission-based security models and its application to android," in *ACM conference on Computer and communications security*. ACM, 2010, pp. 73–84.

45. W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *USENIX conference on Operating systems design and implementation*. USENIX Association, 2010, pp. 1–6.

46. Farrukh Shahzad, S. Bhatti, M. Shahzad, and M. Farooq, "In-execution malware detection using task structures of linux processes," in *IEEE International Conference on Communications*. IEEE, 2011, pp. 1–6.

47. Farrukh Shahzad, M. Shahzad, and M. Farooq, "In-execution dynamic malware analysis and detection by mining information in process control blocks of linux os," *Information Sciences*, 2011.

48. M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Technical report: Detecting capability leaks in android-based smartphones," *Computer Science NC State University, USA*, 2010.

49. L. Liu, G. Yan, X. Zhang, and S. Chen, "Virusmeter: Preventing your cellphone from spies," in *Recent Advances in Intrusion Detection*. Springer, 2009, pp. 244–264.

50. G. Dini, F. Martinelli, A. Saracino, and D. Sgandurra, "Madam: a multi-level anomaly detector for android malware," in *International Conference on Mathematical Methods, Models and Architectures for Computer Network Security*, 2012, pp. 240–253.

51. A. Schmidt, F. Peters, F. Lamour, C. Scheel, S. Çamtepe, and Ş. Albayrak, "Monitoring smartphones for anomaly detection," *Mobile Networks and Applications*, vol. 14, no. 1, pp. 92–106, 2009.

52. A. Shabtai, U. Kanonov, Y. Elovici, C. Glezer, and Y. Weiss, "Andromaly: a behavioral malware detection framework for android devices," *Journal of Intelligent Information Systems*, pp. 1–30, 2011.

53. H. Kim, J. Smith, and K. Shin, "Detecting energy-greedy anomalies and mobile malware variants," in *International Conference on Mobile Systems, Applications, and Services*. ACM, 2008, pp. 239–252.

54. M. Grace, Y. Zhou, Z. Wang, and X. Jiang, "Systematic detection of capability leaks in stock android smartphones," in *Annual Symposium on Network and Distributed System Security*, 2012.

55. I. Witten, U. of Waikato, and D. of Computer Science, *WEKA Practical Machine Learning Tools and Techniques with Java Implementations*. Dept. of Computer Science, University of Waikato, 1999.

56. M. Egele, C. Kruegel, E. Kirda, and G. Vigna, "Pios: Detecting privacy leaks in ios applications," in *Network and Distributed System Security Symposium*, 2011.

57. C. Zheng, S. Zhu, S. Dai, G. Gu, X. Gong, X. Han, and W. Zou, "Smartdroid: an automatic system for revealing ui-based trigger conditions in android applications," in *ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2012, pp. 93–104.

58. G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Annual Computer Security Applications Conference*. ACM, 2010, pp. 347–356.

59. I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

60. A. Shabtai, U. Kanonov, and Y. Elovici, "Intrusion detection for mobile devices using the knowledge-based, temporal abstraction method," *Journal of Systems and Software*, vol. 83, no. 8, pp. 1524–1537, 2010.

61. Y. Shahar, "A framework for knowledge-based temporal abstraction," *Artificial intelligence*, vol. 90, no. 1-2, pp. 79–133, 1997.

62. R. Moskovitch and Y. Shahar, "Medical temporal-knowledge discovery via temporal abstraction," in *AMIA Annual Symposium Proceedings*, vol. 2009. American Medical Informatics Association, 2009, p. 452.

63. D. Damopoulos, S. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, and S. Gritzalis, "Evaluation of anomaly-based ids for mobile devices using machine learning classifiers," *Security and Communication Networks*, 2012.

64. P. Gilbert, B. Chun, L. Cox, and J. Jung, "Automating privacy testing of smartphone applications," Technical Report CS-2011-02, Duke University, Tech. Rep., 2011.

65. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, and A. Sadeghi, "Xmandroid: A new android evolution to mitigate privilege escalation attacks," in *Technical Report TR-2011-04, Technische Universität Darmstadt*, 2011.

66. M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. Wallach, "Quire: lightweight provenance for smart phone operating systems," in *USENIX Security Symposium*, 2011.

67. A. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner, "Android permissions demystified," in *ACM conference on Computer and communications security*. ACM, 2011, pp. 627–638.

68. E. Chin, A. Felt, K. Greenwood, and D. Wagner, "Analyzing inter-application communication in android," in *International conference on Mobile systems, applications, and services*. ACM, 2011, pp. 239–252.

69. M. Schultz, E. Eskin, F. Zadok, and S. Stolfo, "Data mining methods for detection of new malicious executables," in *IEEE Symposium on Security and Privacy*. IEEE, 2001, pp. 38–49.

70. M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter, "L4android: a generic operating system framework for secure smartphones," in *ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 39–50.

71. W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *ACM Conference on Data and Application Security and Privacy*, 2012.

72. Y. Park, C. Lee, C. Lee, J. Lim, S. Han, M. Park, and S. Cho, "Rgbdroid: a novel response-based approach to android privilege escalation attacks," in *Proceedings of the 5th USENIX conference on Large-Scale Exploits and Emergent Threats*. USENIX Association, 2012, pp. 9–9.

73. L. Yan and H. Yin, "Droidscope: Seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX conference on Security symposium*. USENIX Association, 2012, pp. 29–29.

74. B. Davis, B. Sanders, A. Khodaverdian, and H. Chen, "I-arm-droid: A rewriting framework for in-app reference monitors for android applications," *IEEE Mobile Security Technologies, San Francisco, CA*, 2012.

75. S. Lee, E. Wong, D. Goel, M. Dahlin, and V. Shmatikov, "Jana: Platform protection for user privacy," 2012.

76. S. Bugiel, L. Davi, A. Dmitrienko, T. Fischer, A. Sadeghi, and B. Shastry, "Towards taming privilege-escalation attacks on android," in *Network and Distributed System Security Symposium, San Diego, CA*, 2012.

77. A. Bartel, J. Klein, M. Monperrus, Y. Le Traon *et al.*, "Automatically securing permission-based software by reducing the attack surface: An application to android," 2012.

78. P. Gilbert, B. Chun, L. Cox, and J. Jung, "Vision: automated security validation of mobile apps at app markets," in *International workshop on Mobile cloud computing and services*. ACM, 2011, pp. 21–26.

79. D. Wetherall, D. Choffnes, B. Greenstein, S. Han, P. Hornyack, J. Jung, S. Schechter, and X. Wang, "Privacy revelations for web and mobile apps," *Proc. HotOS XIII*, 2011.

80. A. Felt, E. Ha, S. Egelman, A. Haney, E. Chin, and D. Wagner, "Android permissions: User attention, comprehension, and behavior," in *Symposium on Usable Privacy and Security*. ACM, 2012, p. 3.

81. A. Bauer, J. Küster, and G. Vegliach, "Runtime verification meets android security," *NASA Formal Methods*, pp. 174–180, 2012.

82. D. Davidson and B. Livshits, “Morepriv: Mobile os support for application personalization and privacy,” 2012.

83. P. Pearce, A. Felt, G. Nunez, and D. Wagner, “Addroid: Privilege separation for applications and advertisers in android,” in *Proceedings of AsiaCCS*, 2012.

84. G. Nunez, “Party pooper: Third-party libraries in android,” 2011.

85. S. Shekhar, M. Dietz, and D. Wallach, “Adsplit: Separating smartphone advertising from applications,” *Arxiv preprint arXiv:1202.4030*, 2012.

86. I. Leontiadis, C. Efstratiou, M. Picone, and C. Mascolo, “Don’t kill my ads!: balancing privacy in an ad-supported mobile application market,” in *Workshop on Mobile Computing Systems & Applications*. ACM, 2012, p. 2.

87. M. Grace, W. Zhou, X. Jiang, and A. Sadeghi, “Unsafe exposure analysis of mobile in-app advertisements,” in *ACM conference on Security and Privacy in Wireless and Mobile Networks*. ACM, 2012, pp. 101–112.

88. M. La Polla, F. Martinelli, and D. Sgandurra, “A survey on security for mobile devices,” *IEEE Communications Surveys & Tutorials*, no. 99, pp. 1–26, 2012.

89. W. Enck, “Defending users against smartphone apps: Techniques and future directions,” *Information Systems Security*, pp. 49–70, 2011.

90. M. Elfattah, A. Youssif, and E. Ahmed, “Handsets malware threats and facing techniques,” *International Journal of Advanced Computer Science and Applications*, vol. 2, no. 12, 2012.

91. M. Becher, F. Freiling, J. Hoffmann, T. Holz, S. Uellenbeck, and C. Wolf, “Mobile security catching up? revealing the nuts and bolts of the security of mobile devices,” in *IEEE Symposium on Security and Privacy*. IEEE, 2011, pp. 96–111.

92. A. Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner, “A survey of mobile malware in the wild,” in *ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 3–14.

93. Q. Yan, Y. Li, T. Li, and R. Deng, “Insights into malware detection and prevention on mobile phones,” *Security Technology*, pp. 242–249, 2009.

94. P. Vinod, R. Jaipur, V. Laxmi, and M. Gaur, “Survey on malware detection methods,” in *Hackers’ Workshop on Computer and Internet Security*, 2009, pp. 74–79.

95. D. Damopoulos, S. Menesidou, G. Kambourakis, M. Papadaki, N. Clarke, and S. Gritzalis, “Evaluation of anomaly-based ids for mobile devices using machine learning classifiers,” *Security and Communication Networks*,2011.

View publication stats