إقـــــــرار

أنا الموقع أدناه مقدم الرسالة التي تحمل العنوان:

كشف برمجيات اندرويد الخبيثة باستخدام الـ(تانت)

Android Malware Detection Tool Using Dynamic Taint Analysis

أقر بأن ما اشتملت عليه هذه الرسالة إنما هو نتاج جهدي الخاص، باستثناء ما تمت الإشارة إليه حيثما ورد، وإن هذه الرسالة ككل أو أي جزء منها لم يقدم من قبل لنيل درجة أو لقب علمي أو بحثي لدى أي مؤسسة تعليمية أو بحثية أخرى.

DECLARATION

The work provided in this thesis, unless otherwise referenced, is the researcher's own work, and has not been submitted elsewhere for any other degree or qualification

Student's name: Mohammed A. H. Lubbad اسم الطالب: محمد عبد المنعم حسين لبد

Signature: التوقيع:

Date: التاريخ: 2015/07/20

Islamic University of Gaza

Deanery of Graduate Studies

Faculty of Engineering

Computer Engineering Department

كشف برمجيات الأندرويد الخبيثة باستخدام الـ(تانت)

Android Malware Detection Tool Using Dynamic Taint Analysis

Submitted by

Mohammed A. H. Lubbad

Supervised by

Dr. Hasan Qunoo

A Thesis Submitted to the Computer Engineering Department - Faculty of Engineering – in partial Fulfillment of the Requirements for Master Degree in Engineering

1436 هـ – 2015 م

بسم الله الرحمن الرحيم

الجامعة الإسلامية – غزة
The Islamic University - Gaza

مكتب نائب الرئيس للبحث العلمي والدراسات العليا   هاتف داخلي 1150

الرقم ج س غ/35/ Ref

التاريخ 2015/07/13 Date

نتيجة الحكم على أطروحة ماجستير

بناءً على موافقة شئون البحث العلمي والدراسات العليا بالجامعة الإسلامية بغزة على تشكيل لجنة الحكم على أطروحة الباحث/ **محمد عبدالمنعم حسين لبد** لنيل درجة الماجستير في **كلية *الهندسة*** قسم **<u>هندسة الحاسوب</u>** وموضوعها:

كشف برمجيات الأندرويد الخبيثة باستخدام الـ (تانت)

Android Malware Detection Tool Using Dynamic Taint Analysis

وبعد المناقشة التي تمت اليوم الثلاثاء 26 رمضان 1436هـ، الموافق 2015/07/13م الساعة الثانية مساءً، اجتمعت لجنة الحكم على الأطروحة والمكونة من:

د. حسن نجيب قنوع   مشرفاً ورئيساً   ....

د. أيمن أحمد أبو سمرة   مناقشاً داخلياً   .... ابو

د. عبد الحميد بشير زغبر   مناقشاً خارجيًا   ....

وبعد المداولة أوصت اللجنة بمنح الباحث درجة الماجستير في **كلية *الهندسة*** / قسم **<u>هندسة الحاسوب</u>**.

واللجنة إذ تمنحه هذه الدرجة فإنها توصيه بتقوى الله ولزوم طاعته وأن يسخر علمه في خدمة دينه ووطنه.

والله ولي التوفيق ،،،

مساعد نائب الرئيس للبحث العلمي وللدراسات العليا

أ.د. فؤاد علي العاجز

شئون البحث العلمي والدراسات العليا
الجامعة الإسلامية غزة
The Islamic University
Research & Graduate Affairs

الملخص

تحليل أكواد تطبيقات أنظمة الأندرويد وكشف الخبيث منها يعتبر محور بحث هام جدا ، هناك الكثير من الأبحاث والطرق الجديدة والمتنوعة في هذا المجال لكنها في مجملها تركز على الكشف عن البرمجيات الخبيثة في وقت الفحص (ScanTime) ما قبل التنفيذ وذلك عن طريق مقارنة التشابه بين الجديد الغير مكتشف منها بالقديم المكتشف والمفهرس في مواقع قواعد بيانات البرمجيات الخبيثة والفحص الشهيرة مثل موقع (VirusTotal) بطرق مختلفة ومتعددة ، عدد هذه الأبحاث ينمو بسرعة كبيرة وذلك لتزايد عدد تطبيقات انظمة الأندرويد بشكل ضخم وكبير .

هذه الأطروحة تحاول أن تساهم في حل هذه المشكلة عن طريق استخدام طريقة مبتكرة وحديثة لتميز بين البرمجيات السليمة والخبيثة حيث تقوم بتحليل الأكواد أثناء التنفيذ (RunTime) ومن ثم تقوم بتتبع وتسجيل تسرب البيانات ذات الطابع الخاص (Privacy Data) والسرية عبر الانترنت او كرت الشبكة اللاسلكي أو اي منفذ آخر يقوم بالاتصال بجهة خارجية كالبلوتوث وذلك عن طريق استخدام التحليل الديناميكي المسمى (Dynamic Taint Analysis) وتقنيات الذكاء الاصطناعي وتعليم الآلة.

قامت هذه الأطروحة بتطبيق هذه المفاهيم كجزء من نظام الحماية ضد البرمجيات الخبيثة كما وتم التأكد من صلاحية الفكرة وتم مقارنة النتائج لتحليل البرمجيات والمفاضلة بين عدة تقنيات ذكاء اصطناعي للحصول على أفضل وأدق النتائج، حيث تم استخدام عينة من البرمجيات السليمة والخبيثة تتكون من 50 تطبيق وتنفيذهم على منظومة فحص تم اعدادها بعناية فائقة تقوم مراقبة وتسجيل تسرب البيانات ومن ثم تحليل البيانات المسربة باستخدام الذكاء الاصطناعي وقد تم انتقاء التقنية المصنفة (Random Forests) حيث قدمت أفضل النتائج في عملية التصنيف وكانت النتيجة أن الطريقة المقترحة في الرسالة حققت نسبة اكتشاق وتحقق 74.7% وتعتبر هذه النتيجة مرضية واكثر من جيدة تبعا لتنوع البرمجيات الخبيثة وتعدد تصنيفاتها من برمجيات تجسس وفيروسات وأحصنة طروادة، وتعتبر هذه الأطروحة من الدراسات والأبحاث القليلة في هذا المجال وذلك لصعوبة تتبع وفحص البرمجيات في وقت التنفذ أو أثناء تشغيلها (RunTime) وذلك من خلال تجهيز واعداد بيئة الفحص حيث تم تنزيل نظام تشغيل الأندرويد مفتوح المصدر والتعديل عليه بشكل كبير لتتبع حركة البيانات ومن ثم بناء نسخة جديدة تعمل على هاتف ذكي خاص لهذه العملية حيث تم تنفيذ البرمجيات عليه واستخراج النتائج الأولية لعملية التصنيف والكشف.

Abstract

Code analysis and Malwares detection for Android applications are considered as an serious problem; there are many researches to apply new and creative techniques that can detect Malwares at scan time before run the application then compare the similarity between them and the old malwares that archived on some malwares databases and some scanning website like VirusTotal. These researches are being rapidly grown because of wide using and a huge number of new applications.

This thesis tries to take the lead of the way of detection malwares using dynamic analysis in specific dynamic taint analysis this method based on android application analysis at run time then monitoring and logging the information flow out of the device from any port like wireless card interface or Bluetooth, specially private data and secure info such as credit card info, SMS, contacts, IMEI . etc, Our malware dataset consist of **50** Android applications for this research **50%** of them benign and the rest malwares. Finally we feed the machine learning algorithm with data to classify it and we measure the accuracy and detection ratio it reach **74.7%** this result being satisfied and good enough because of variety of malwares in real life and difficulties on classifying them such like Trojans, spywares, exploits and viruses application.

Thesis is considered as one of little researches on malwares detection using dynamic analysis, this because of huge difficulties faced by the way of monitoring and logging the information flow, it also take from us a huge effort on prepare and initialize the testing environment, downloading android OS source code and making some modifications then build a customize version that been compatible with some special devices types.

Dedication

To my loving parents, family and to my wife

Acknowledgement

I would like to acknowledge my thesis supervisor Dr. Hasan Qunoo for their guidance and valuable help; I would also to thank Dr. Aiman Abu Samra for his advices and his valuable discussion around my work. Finally I appreciate Ms. Karl Hiramoto researcher at VirusTotal for giving me permission for Android malwares dataset.

Contents

Glossary

Astro	File Manager application.
VirusTotal	A website, originally developed by Hispasec, that provides free checking of files for viruses.
Benign	Clean, the opposite of malignant.
Malware	A short for malicious software.
TaintDroid	An Information Flow Tracking System for Real-Time Privacy Monitoring on Smartphones.
Weka	A collection of machine learning algorithms for data mining tasks.
Taint	
TaintCollector	An Android application that parse taints from android log and save it on database file.
True Positive Rate	Correctly classified ratio.
False Positive Rate	Incorrectly classified ratio.
Kappa Statistic	
Receiver Operating Characteristic	A trade off between TPR and FPR
Aptoide	An unofficial android market.
Mobogenie	An unofficial android market.
1Mobilemarket	An unofficial android market.

Trojan	Generally a non-self-replicating type of malware program containing malicious code
Virus	A small infectious agent that replicates only inside the living cells of other organisms.
Exploit	A sequence of commands that takes advantage of a bug or vulnerability in order to cause unintended or unanticipated behavior to occur on computer software
Spyware	A software that aims to gather information about a person or organization without their knowledge and that may send such information to another entity without the consumer's consent, or that asserts control over a computer without the consumer's knowledge.
Zero-day attack	A severe threat. The terms also describe warez-group releases of pirated software on or before the release of the software.
Contagio	A collection of the latest malware samples, threats, observations, and

	analyses.
Jellybean	A legacy version of Google's mobile OS.
Dex files	Android’s Java byte-code

Table Of Abbreviation

SIMPIL	A simple intermediate language
IP	Internet protocol
GPS	Geometric Position System
IF	information flow
AI	Artificial intelligent
ACG	Activity Call Graph
MDG	Minimal Direct Graph
AV	Antivirus
ROC	Receiver Operating Characteristic
AMDA	Automatic Malware Detection Algorithm
ARFF	Attribute Relation File-Format
APK	Android Application
TPR	True Positive Rate
FPR	False Positive Rate

List of figures

List of tables

Chapter 1

1 Introduction

1.1 Motivation

A key feature of modern smartphone platforms is a centralized service for downloading third-party applications. The convenience to users and developers of such “app stores” has made mobile devices more fun and useful, and has led to an explosion of development. Apple’s App Store alone served nearly 3 billion applications after only 18 months. Many of these applications combine data from remote cloud services with information from local sensors such as a GPS receiver, camera, microphone, and accelerometer. Applications often have legitimate reasons for accessing this privacy sensitive data, but users would also like assurances that their data is used properly. Incidents of developers relaying private information back to the cloud and the privacy risks posed by seemingly innocent sensors like accelerometers illustrate the danger. Resolving the tension between the fun and utility of running third-party mobile applications and the privacy risks they pose is a critical challenge for smartphone platforms. Mobile-phone operating systems currently provide only coarse-grained controls for regulating whether an application can access private information, but provide little insight into how private information is actually used. For example, if a user allows an application to access her location information, she has no way of knowing if the application will send her location to a location-based service, to advertisers, to the application developer, or to any other entity. As a result, users must blindly trust that applications will properly handle their private data, So the question is how to detect unknown malware that spies on private information by reliable and useful method with low cost of processing to save mobile resources.

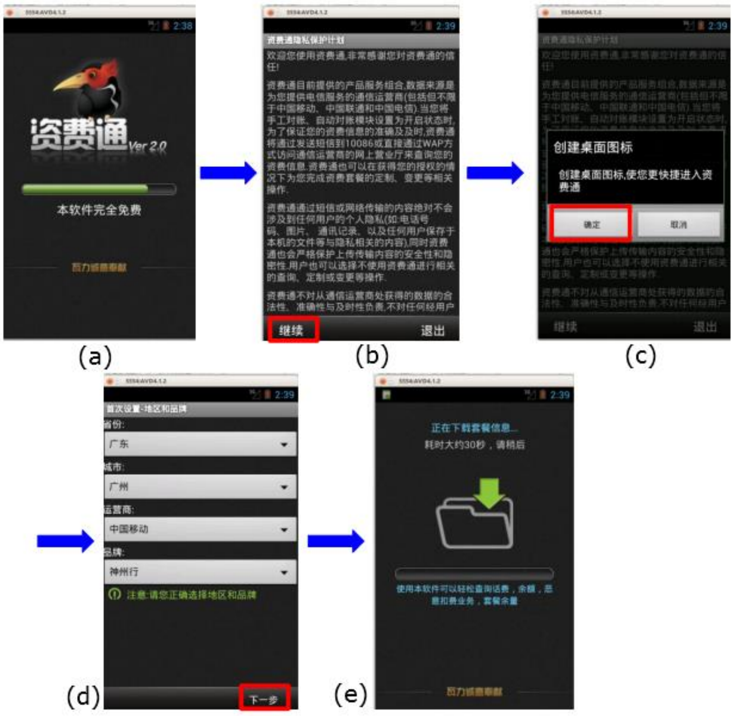
1.2 The problem

As technology becomes more and more advanced we are becoming less and less reliable on stationary methods of computation which we have spent a great deal of time learning to secure. In nowadays, day and age the aspect of mobility is now almost synonymous with the business and social world. Smartphones are used daily to transfer data via a wide variety of both native and third party applications. However, for despite their usage they have yet to reach the same level of security as desktop computing. For such a large amount of data transferred via mobile devices there is not nearly enough security, especially for the type of private data many people store on their phones such as GPS location, banking information, contacts, emails, etc. In order to help address this, the makers of TaintDroid have developed a system which can identify the pieces of private data being sent from your mobile device and the IP where they are being sent.

1.3 The problem statement

Many Applications on Android Market not verified by google (which is the case in AppStore) Developers can only request coarse-grained permissions, users rarely reads or understands the meaning of the permissions, so there is a leak of security which guide attackers to do what they want with sensitive data without any restriction, TaintDroid developed for Android with the purpose of analyzing Android applications with aspect to information flow (IF), it's an example of a dynamic analysis system of IF, it's only detect sensitive data gone out of the smartphone, but not prevent it from going out, leak of sensitive user information, but still not enough to detect and classify the variety of mobile malwares all over the world.

In figure 1, How to detect new unknown Malwares that steal you private data and violate privacy.



资费通隐私保护计划

欢迎您使用资费通,非常感谢您对资费通的信任!

资费通目前提供的产品服务组合,数据来源是为您提供电信服务的通信运营商(包括但不限于中国移动、中国联通和中国电信).当您将手工对账、自动对账模块设置为开启状态时,为了保证您的资费信息的准确及及时,资费通将通过发送短信到10086或直接通过WAP方式访问通信运营商的网上营业厅来查询您的资费信息.资费通也可以在获得您的授权的情况下为您完成资费套餐的定制、变更等相关操作.

资费通通过短信或网络传输的内容绝对不会涉及到任何用户的个人隐私(如:电话号码、图片、通讯记录、以及任何用户保存于本机的文件等与隐私相关的内容),同时资费通也会严格保护上传传输内容的安全性和隐密性,用户也可以选择不使用资费通进行相关的查询、定制或变更等操作.

资费通不对从通信运营商处获得的数据的合法性、准确性与及时性负责,不对任何经用户

继续 退出

创建桌面图标

创建桌面图标,使您更快捷进入资费通

确定 取消

首次设置-地区和品牌

省份: 广东

城市: 广州

运营商: 中国移动

品牌: 神州行

注意:请您正确选择地区和品牌

下一步

正在下载套餐信息...

耗时大约30秒，请稍后

使用本软件可以轻松查询话费，余额，是否扣费业务，套餐余量

Figure 1 Information leakage example

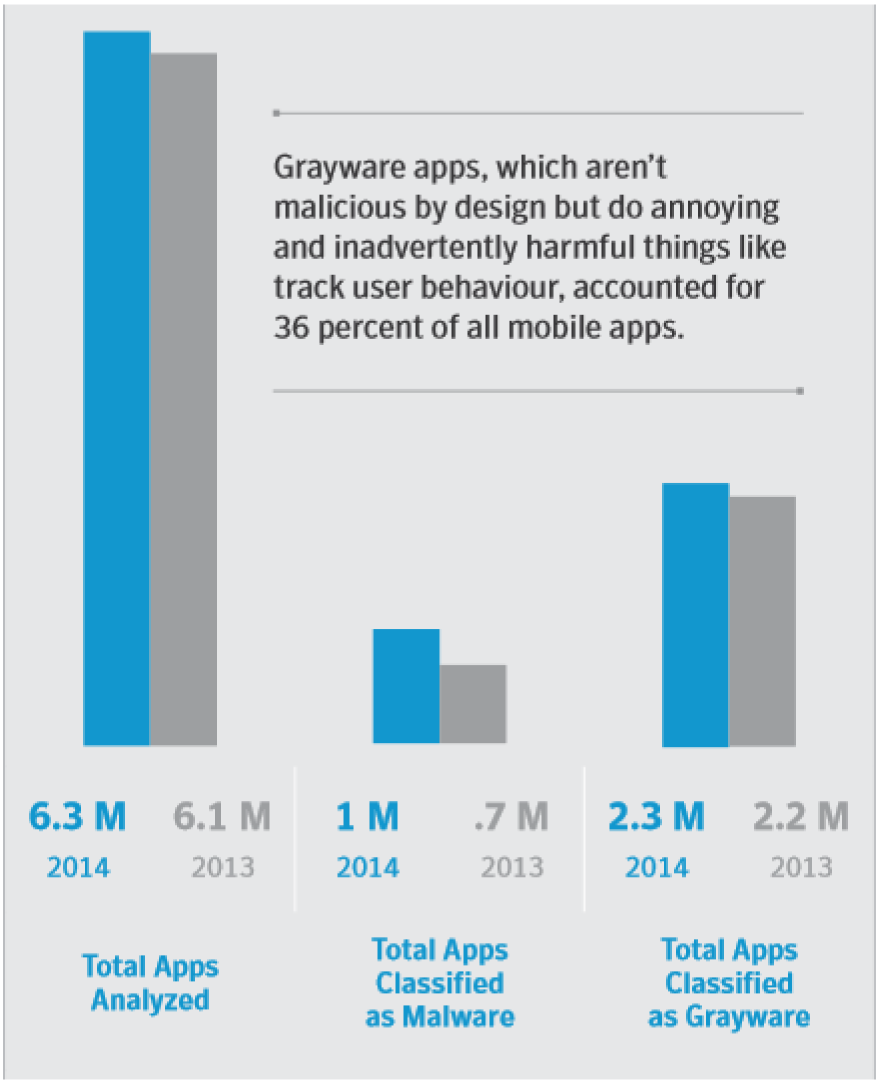


Figure 2 App Analysis by Symantec's Norton Mobile Insight [1]

Figure 2 shows massive increasing of Apps that classified as malware on one year only from 0.7 million to 1 million.

1.4 Thesis contribution

The main contribution of this research is to propose a new malware detection model for android application based on TaintDroid, the new model will detect all Android apps tainted communication, then sent them to a remote server for heuristic analysis by one of AI Classifier algorithms that help to detect malicious code in the app repackaged or injected with malicious code malwares in general, Finally the model will be able to detect a new malware applications who have features similar to classified group.

1.5 Thesis Structure

Here the contents of each chapter are summarized and placed in the overall research process.

In *Chapter 2* we gather all previous studies and related works that was relevant to our topic and summarizing it with brief description of the related work, then we classify them into four topics android security, dynamic analysis of malwares, TaintDroid implementation, machine learning algorithms and classifiers.

In the beginning of *Chapter 3* presents a review relevant literature. Dynamic analysis, dynamic taint analysis are described and critiqued. The concept of dynamic analysis, its definition and importance. This review focus also on dynamic taint analysis challenges, opportunities and also limitations and drawbacks.

Then in *Chapter 4* Describes the implementation of dynamic taint analysis on android (TaintDroid), the concept of TaintDroid, its definition and importance, architecture, challenges and opportunities and also limitations and drawbacks.

Next in *Chapter 5* Describes the method we used of how making the detection for mobile application (APKs) by doing deep analysis of types of private data that the application violate the rules and send them via internet or wireless network card and then classify them into two classes benign or malware.

Later in *Chapter 6* is focused on the experiments, results and evaluation that we done in this study, firstly we make a single case study to describe how we test the android application at runtime, analysis its behavior, gathering information and building the taints dataset, then we discussed a real test comparison of some machine learning classifiers and select the best of them, finally we get the results and explain it with charts.

Describes and summarizes the core findings of this thesis. Then brief discussion of results we found that lead us to conclusion of our study and some advices and points for researchers to complete after this study in the same field in future work **in *Chapter 7*.**

1.6 Summary

In this chapter, we define the problem that we focused on during this study. Section 1.1 describes briefly the study problem, then we discussed it on more details on section 1.2, after that we make a contribution for the issue and explain it at section 1.3, finally we build the study structure at section 1.4.

Chapter 2

2 Related Work

In this chapter we summarized some of previous works that have been relevant to our study and divide them into four sections. Section 2.1 Android security researches from past to present, section 2.2 Dynamic taint analysis, the analysis based on runtime and the behavior of the application, at section 2.3 we see the TaintDroid system implementation on android source code and its limitations and restrictions, then we take some look at machine learning classifier and its accuracy in section 2.4.

2.1 Android security

To save power of scan processing some strategies are proposed to detect malware in malware in Smartphones as in [2] ,A preliminary prototype based on this strategy has been built for the android dev phone. solution will be outside the Smartphones. The key observation is that users often connect their Smartphones to a PC or desktop for information synchronization. Based on this observation, strategies that run detection/prevention software will be PC/desktop.

In [3], automatic malware detection mechanism for the Android platform based on the results from sandbox is proposed. They extracted network spatial features of Android apps and used independent component analysis (ICA) to determine the intrinsic domain name resolution behavior of Android malware. The proposed mechanism can identify Android malware automatically. A public Android malware app dataset and popular benign apps collected from the Android Market are used for evaluating the effectiveness of the proposed approach in terms of its grouping ability and effectiveness in identifying Android malware.

2.2 Dynamic analysis and prevention

For dynamic prevention [4] kernel-based behavior analysis for android malware inspection is presented. The system consists of a log collector in the Linux layer and a log analysis application. The log collector records all system calls and filters events with the target application. The log analyzer matches activities with signatures described by regular expressions to detect a malicious activity. Signatures of information leakage are automatically generated using the Smartphone IDs, e.g., phone number, SIM serial number, and Gmail accounts. They implement a prototype system and evaluate 230 applications in total. The result shows that our system can effectively detect malicious behaviors of the unknown applications.

Some studies investigate the behavior of malicious Android applications; they [5] present a simple and effective way to safely execute and analyze them. As part of this analysis, they use the Android application sandbox Droidbox to generate behavioral graphs for each sample and these provide the basis of the development of patterns to aid in identifying it. As a result, they are able to determine if family names have been correctly assigned by current anti-virus vendors. Results indicate that the traditional anti-virus mechanisms are not able to correctly identify malicious Android applications. The work in this paper is the first to examine behavior in malicious applications using DroidBox. dataset comprises samples that were collected from publicly available sources. Each malicious application is executed for 60 seconds in a sandboxed environment and the generated log files are collected at the end of execution. Using Droidbox, they also generate two types of graphs (behavior graphs and treemap graphs) for each sample. Both graphs help us to analyze the activities performed during run-time and also to establish patterns between variants from the same malware family. These graphs also illustrate how some benign applications might leak data connected to short message service (SMS) texting and other features of the applications.

A feature-based mechanism to provide a static analyst paradigm for detecting the Android malware proposed in [6] . The mechanism considers the static information including permissions, deployment of components, Intent messages passing and API calls for characterizing the Android applications behavior. In order to recognize different intentions of Android malware, different kinds of clustering algorithms can be applied to enhance the malware modeling capability. Furthermore, They leverage the proposed mechanism and develop a system, called DroidMat˙It extracts the information (e.g., requested permissions, Intent messages passing, etc) from each application’s manifest file. In addition, it traces API calls for each component since API calls in different components may imply different intentions. Then, it applies K-means algorithm that enhances the malware modeling capability. Finally, it uses kNN algorithm to classify the applications benign or malicious. This will help in providing a static feature-based mechanism to extract represent active configuration and trace API calls for identifying the Android malware. So there are no need for dynamic simulation that can save the cost in environment deployment and manual efforts in investigation.

Detecting malicious behavior in Android apps by automatically collecting Android applications, analyzing relevant malicious behavior and informing of the results. The proposed framework [7] [6] features automatic collection of apps in the Android market and black markets; static analysis extracting risky APIs and strings from Android apps; and dynamic analysis based on the patterns of malicious behavior

New algorithms for static analysis of Android applications was proposed [8] in order to address some problems that users and developers are encountering. All of the algorithms are based on the similarity distance using real world compressors [9]. To determine if someone pirated an application or parts of an application. They extend this problem to extract automatically malware that has been injected into an application. In addition, they find similarities between two applications, the algorithm can be applied to evaluate the efficiency of an obfuscator on the application. They have used the same algorithms to identify small dissimilarities in methods and in basic blocks (without using graphs) in order to use the longest common subsequence algorithm to extract exact differences.

In the other hand, new dynamic analysis techniques and algorithms proposed in [10], Dynamic taint analysis and forward symbolic execution are quickly becoming staple techniques in security analyses. Example applications of dynamic taint analysis and forward symbolic execution include malware analysis, input filter generation, test case generation, and vulnerability discovery. Despite the widespread usage of these two techniques, there has been little effort to formally define the algorithms and summarize the critical issues that arise when these techniques are used in typical security contexts.

2.3 Dynamic Taint Analysis

Dynamic analysis is the capability monitoring code execution, has become major tool in computer security. Dynamic analysis is catchy, it allows us to reason about actual executions, and so can precise security analysis based upon run-time information. On the other hand, dynamic analysis is modest, we need only consider information about a single execution at a time.

2.3.1 Concept of dynamic taint analysis

One of the most commonly used dynamic taint analysis. Dynamic taint analysis runs a program and notices which computations are affected by predefined taint sources such as user input.

The number of security applications utilizing this technique is enormous. Example security research areas employing dynamic taint analysis is:

- Unknown Vulnerability Detection. Dynamic taint analysis can observe for misuses of user input while an execution. For example, dynamic taint analysis can be used to prevent code injection attacks by monitoring whether user input is executed.

- Data-Flow Tracking System. Dynamic taint analysis can used for tracking data-flow from user input to output and while propagation. For example, dynamic taint analysis can be used to prevent lack in privacy data while d

```
program       ::= stmt*
ptmt s        ::= var := exp | store(exp, exp)
                        | goto exp | assert exp
                        | if exp then goto exp
                          else goto exp
exp e         ::= load(exp) | exp $\Diamond_b$ exp | $\Diamond_u$ exp
                   | var | get_input(src)  | v
$\Diamond_b$            ::= typical binary operators
$\Diamond_u$            ::= typical binary operators
Value v       ::= 32-bit unsigned integer
```

Figure 3 A simple intermediate language (SIMPIL). [10]

- $\Diamond_b$ to represent typical binary operators, e.g., addition, subtraction, etc. Similarly.
- $\Diamond_u$ represents unary operators such as logical negation.
- The statement *get_input(src)* returns input from source *src*. We use a *dot (.)* to denote an argument that is ignored, e.g., we will write *get _input(.)* when the exact input source is not relevant.
- For simplicity, we consider only expressions (constants, variables, etc.) that evaluate to 32-bit integer values, extending the language and rules to additional types is straightforward.

Dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input. A precise definition of dynamic taint analysis must target a specific language. It used SIMPIL: a Simple Intermediate Language. The grammar of SIMPIL is presented in Figure 3. Although the language is simple, it is powerful enough to express typical languages as varied as Java and assembly code. Indeed, the language is representative of internal representations used by compilers for a variety of programming languages.

The execution context is described by five parameters:

- The list of program statements (Σ).
- The current memory state (μ).
- The current value for variables (Δ).
- The program counter (pc).
- The current statement (i).

The Σ, μ and Δ contexts are maps, e.g., Δ[x] denotes the current value of variable x. They denote updating a context variable x with value v as x   v, e.g., Δ[x   10] denotes setting the value of variable x to the value 10 in context Δ.

A summary of the five meta-syntactic variables is shown in Figure 3.

$$\frac{v\ is\ input\ from\ src}{\mu, \Delta \vdash get\_input(src) \Downarrow v} INPUT \quad \frac{\mu, \Delta \vdash e \Downarrow v_1\ v = \mu[v_1]}{\mu, \Delta \vdash load\ e\ \Downarrow\ v} LOAD \quad \frac{}{\mu, \Delta \vdash var \Downarrow \Delta[var]} VAR$$

$$\frac{\mu, \Delta \vdash e\ \Downarrow\ v\, v' = \lozenge_u\, v}{\mu, \Delta \vdash \lozenge_u\, e\ \Downarrow\ v'} UNOP \quad \frac{\mu, \Delta \vdash e_1 \Downarrow v_1 \quad \mu, \Delta \vdash e_2 \Downarrow v_2 \quad v' = v_1 \lozenge_b v_2}{\mu, \Delta \vdash e_1 \lozenge_b e_2 \Downarrow v'} BINOP \quad \frac{}{\mu, \Delta \vdash v \Downarrow v} CONST$$

Figure 4 Summary of the five meta-syntactic variables [10]

The evaluation rules for expressions use a similar notation. They denote by *μ*, *Δ*, ⊦, *e*, ⇓ , *v* evaluating an expression e to a value v in the current state given by *μ* and *Δ*. The expression *e* is evaluated by matching *e* to an expression evaluation rule and performing the attached computation.

$$\frac{computaion}{\langle current\ state\rangle, stmt\ \rightsquigarrow\ \langle end\ state\rangle, stmt'}$$

Figure 5 Evaluation rules for expressions [10]

Rules are read bottom to top, left to right. Given a statement, we pattern-match the statement to find the applicable rule. We then apply the computation given in the top of the rule, and if successful, transition to the end state. If no rule matches (or the computation in the premise fails), then the machine halts abnormally.

Consider evaluating the following program:

*x := 2 * get_input (.)   [Take input = 20]*

$$\cfrac{\cfrac{\cfrac{\square}{\mu,\Delta \vdash 2 \Downarrow 2}\,CONST \quad \cfrac{20\ is\ input}{\mu,\Delta \vdash get\_input(.) \Downarrow 20}\,INPUT\ v' = 2 * 20}{\mu,\Delta \vdash 2 * get\_input(.) \Downarrow 40}\,BINOP\ \Delta' = \Delta[x \leftarrow 40]\ i = \sum[pc + 1]}{\Sigma,\mu,\Delta,pc,x := 2 * get_{input(.)} \rightsquigarrow \Sigma,\mu,\Delta',pc + 1,i}\,ASSI$$

Figure 6 TaintDroid Rule [10]

The purpose of dynamic taint analysis is to track information flow between sources and sinks. Any program value whose computation depends on data derived from a taint source is considered tainted (denoted T). Any other value is considered untainted (denoted F). A taint policy P determines exactly how taint flows as a program executes, what sorts of operations introduce new taint, and what checks are performed on tainted values.

There are two types of errors

- **Overtainted:** Dynamic taint analysis can mark a value as tainted when it is not derived from a taint source. For example, in an attack detection application overtainting will typically result in reporting an attack when no attack occurred.

- **Undertainted:** Dynamic taint analysis can miss the information flow from a source to a sink. In the attack detection scenario, undertainting means the system missed a real attack. A dynamic taint analysis system is precise if no undertainting or overtainting occurs.

2.3.2 Dynamic Taint Policies

A taint policy specifies three properties:

- **Taint Introduction:** Taint introduction rules specify how taint is introduced into a system. The typical convention is to initialize all variables, memory cells, etc. as untainted.
- **Taint Propagation:** Taint propagation rules specify the taint status for data derived from tainted or untainted operands.
- **Taint Checking:** Taint status values are often used to determine the runtime behavior of a program, e.g., an attack detector may halt execution if a jump target address is tainted.

Tainted Jump Policy is essential policy in dynamic taint policies, however the goal of the tainted jump policy is to protect a potentially vulnerable program from control flow hijacking attacks. In SIMPIL, we perform checking by adding the policy to the premise of the operational semantics. For instance, the T-GOTO rule uses the Pgotocheck(t) policy. Pgotocheck(t) returns T if it is safe to perform a jump operation and returns F otherwise. If F is returned, the premise for the rule is not met and the machine terminates abnormally (signifying an exception). The policy introduces taint into the system by marking all values returned by get*input(.) as tainted. Taint is then propagated through the program in a straightforward manner, e.g., the result of a binary operation is tainted if either operand is tainted, an assigned variable is tainted if the right-hand side value is tainted, and so on.

Table 1 Tainted Jump Policy [10]

Component	Policy Check
P_{input} (.), $P_{bincheck}$ (.), $P_{memcheck}$ (.)	T
P_{const} ( )	F
P_{unop} (t), P_{assign} (t)	t
P_{binop} (t_1, t_2)	$t_1 \vee t_2$
$P_{memcheck}$ (t_a, t_v)	t_v
$P_{condcheck}$ (t_e, t_a)	$\neg t_a$
$P_{gotocheck}$ (t)	$\neg t_a$

```
1.  x := 2 * get_input(.)
2.  y := 5 + x
3.  goto  y
```

Figure 7 Taintdroid executing program [10]

In figure 7, on line 1, the executing program receives input, assumed to be 20, and multiplies by 2. Since all input is marked as tainted, *2 * get_input(.)* is also tainted. On line 2, x (tainted) is added to y (untainted). Since one operand is tainted, y is marked as tainted. On line 3, the program jumps to y. Since y is tainted, the T-GOTO premise for P is not satisfied, and the machine halts abnormally.

Table 2: Taint calculations for example program [10]

Line #	Statement	Δ	TΔ	Rule	pc
	start	{}	{}		1
1	x:=2*get_input(.)	{x←40}	{x→T}	T-ASSIGN	2
2	y:=5+x	{x←40, y←45}	{x→T, y→T }	T-ASSIGN	3
3	goto y	{x←40, y←45}	{x→T, y→T }	T-GOTO	error

2.3.2.1 Different Policies for Different Applications

Different applications of taint analysis can use different policy decisions. As we will see in the next section, the typical taint policy described in Jump Table 2 is not appropriate for all application domains, since it does not consider whether memory addresses are tainted. Thus, it may miss some attacks. We discuss alternatives to this policy in the next section.

2.3.3 Dynamic Taint Analysis Challenges and Opportunities

There are several challenges to using dynamic taint analysis correctly, including:

- **Tainted Addresses.** Distinguishing between memory addresses and cells is not always appropriate.
- **Undertainting.** Dynamic taint analysis does not properly handle some types of information flow.

- **Overtainting.** Deciding when to introduce taint is often easier than deciding when to remove taint.
- **Time of Detection vs. Time of Attack.** When used for attack detection, dynamic taint analysis may raise an alert too late

```
1) x := get_input(.)
2) y := load(z+x)
3) goto y
```

Figure 8 TaintDroid index table as input [10]

In figure 8 user provides input to the program that is used as a table index. The result of the table lookup is then used as the target address for a jump. Assuming addresses are of some fixed-width (say 32-bits), the attacker can pick an appropriate value of x to address any memory cell He/she wishes.

As a result, the attacker can jump to any value in memory that is untainted. In many programs this would allow the user to violate the intended control flow of the program, thus creating a security violation. The tainted jump policy applied to the above program still allows an attacker to jump to untainted, yet attacker determined locations. This is an example of under-taint by the policy. This means that the tainted jump policy may miss an attack. One possible fix is to use the tainted addresses policy. Using this policy, a memory cell is tainted if either the memory cell value or the memory address is tainted.

2.3.3.1 Sanitization

One of the dynamic analysis problem, dynamic taint analysis as described only adds taint, it never removes it, leads to the problem of taint spread, as the program executes, more and more values become tainted, often with less and less taint precision. A significant challenge in taint analysis is to identify when taint can be removed from a value. We call this the taint sanitization problem. One common example where we wish to sanitize is when the program computes constant functions. A typical example in x86 code is *b* = *a X-OR a*. Since *b* will always equal zero, the value of b does not depend upon *a*. A default taint analysis policy, however, will identify b as tainted whenever *a* is tainted.

2.3.3.2 Time of Detection vs Time of Attack

Dynamic taint analysis can be used to flag an alert when tainted values are used in an unsafe way. However, there is no guarantee that the program integrity has not been violated before this point. Note, however, that the tainted jump policy does not raise an error when the return address is first overwritten only when it is later used as a jump target. Thus, the exploit will not be reported until the function returns. Arbitrary effects could happen between the time when the return address is first overwritten and when the attack is detected, e.g., any calls made by the vulnerable function will still be made before an alarm is raised.

2.4 TaintDroid – Application of dynamic taint analysis on Android

To identify possible information leakage, LeakMiner [7] applies a static taint analysis to apps within Android market. The approach introduces three steps in identifying possible leakages: first, apk files of Android apps are transformed to Java bytecode so that the following analysis can directly work on Java bytecode. Besides, application metadata are extracted from the manifest file of Android app. Then, LeakMiner identifies sensitive information according to the extracted metadata. Finally, taint information is propagated through call graph to identify possible leakage paths. By introducing multiple entry point call graph, they can cover all the code of Android app. They choose a set of 1750 apps to evaluate the accuracy of LeakMiner. LeakMiner can identify 145 real leakages in this app set.

However, there is an implementation of dynamic analysis technique TaintDroid in [11], Today's smartphone operating systems frequently fail to provide users with adequate control over and visibility into how third-party applications use their private data. We address these shortcomings with TaintDroid, an efficient, system-wide dynamic taint tracking and analysis system capable of simultaneously tracking multiple sources of sensitive data. TaintDroid provides real time analysis by leveraging Android's virtualized execution environment. TaintDroid incurs only 14% performance overhead on a CPU-bound micro-benchmark and imposes negligible overhead on interactive third-party applications. Using TaintDroid to monitor the behavior of 30 popular third-party Android applications, we found 68 instances of potential misuse of users' private information across 20 applications. Monitoring sensitive data with TaintDroid provides informed use of third-party applications for phone users and valuable input for smartphone security service firms seeking to identify misbehaving applications.

While increasing of android worldwide smart phone market share, it reached almost 50% at 2011 then IOS becomes second largest smart phone platform.

In spite of its popularity it has many security threats and vulnerability, because the operating systems fail providing users with sufficient control over and clarity into how third-party applications use their private data. TaintDroid is one example of dynamic analysis tools for Android applications. TaintDroid is a specially crafted DVM that supports taint analysis of Dalvik instructions and across API calls. The biggest advantage of using TaintDroid is that it runs on actual devices. All of the hardware, sensors, vendor software and unpredictable complexities that come with a real device are there. This can't be achieved in an emulated environment.

TaintDroid is a real-time dynamic taint tracker for Android with the explicit goal of protecting users from information leakage. However, TaintDroid has revealed that there are many apps that leak private information to the network.

2.4.1 TaintDroid Challenges & opportunities

Monitoring data-flow over network of privacy sensitive information on smartphones faces several challenges:

- Smartphones are resource constrained. The resource limitations of smartphones precludes the use of heavyweight information tracking systems such as Panorama.
- Third-party applications are entrusted with several types of privacy sensitive information. The monitoring system must distinguish multiple information types, which requires additional computation and storage.
- Context-based privacy sensitive information is dynamic and can be difficult to identify even when sent in the clear. For example, geographic locations are pairs of floating point numbers that frequently change and are hard to predict. Applications can share information. Limiting the monitoring system to a single application does not account for flows via files and IPC between applications, including core system applications designed to disseminate privacy sensitive information.

2.4.2 TaintDroid Architecture

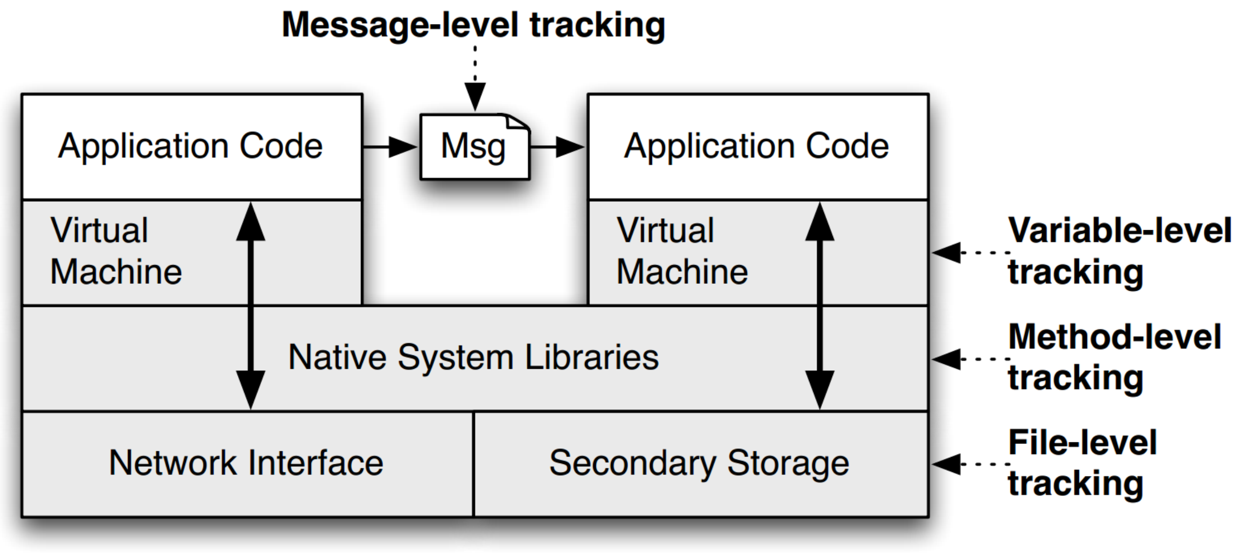


Figure 9 TaintDroid architecture [11]

Figure 9 show our approach to taint tracking on smartphones. We leverage architectural features of virtual machine-based smartphones (e.g., Android, BlackBerry, and J2ME-based phones) to enable efficient, system-wide taint tracking using fine-grained labels with clear semantics. First, we instrument the VM interpreter to provide variable-level tracking within untrusted application code. Using variable semantics provided by the interpreter provides valuable context for avoiding the taint explosion observed in the x86 instruction set.

Additionally, by tracking variables, we maintain taint markings only for data and not code. Second, we use message-level tracking between applications. Tracking taint on messages instead of data within messages minimizes IPC overhead while extending the analysis system wide. Third, for system-provided native libraries, we use method-level tracking. Here, we run native code without instrumentation and patch the taint propagation on return. These methods accompany the system and have known information flow semantics. Finally, we use file level tracking to ensure persistent information conservatively retains its taint markings.

2.4.3 Privacy Hook Placement

Taint sources can only add taint tags to memory for which TaintDroid provides tag storage. Currently, taint source and sink placement is limited to variables in interpreted code, IPC messages, and files. This section discusses how valuable taint sources and sinks can be implemented within these restrictions. We generalize such taint sources based on information characteristics.

2.4.3.1 Low-bandwidth Sensors

A variety of privacy sensitive information types are acquired through low-bandwidth sensors, e.g., location and accelerometer. Such information often changes frequently and is simultaneously used by multiple applications. Therefore, it is common for a smartphone OS to multiplex access to low-bandwidth sensors using a manager. This sensor manager represents an ideal point for taint source hook placement. For our analysis, we placed hooks in Android's LocationManager and SensorManager applications.

2.4.3.2 High-bandwidth Sensors

Privacy sensitive information sources such as the microphone and camera are high-bandwidth. Each request from the sensor frequently returns a large amount of data that is only used by one application. Therefore, the smartphone OS may share sensor information via large data buffers, files, or both.

When sensor information is shared via files, the file must be tainted with the appropriate tag. Due to flexible APIs, we placed hooks for both data buffer and file tainting for tracking microphone and camera information.

2.4.3.3 Information Databases

Shared information such as address books and SMS messages are often stored in file based databases. This organization provides a useful unambiguous taint source similar to hardware sensors. By adding a taint tag to such database files, all information read from the file will be automatically tainted. We used this technique for tracking address book information. Note that while TaintDroid's file-level granularity was appropriate for these valuable information sources, others may exist for which files are too coarse grained.

However, we have not yet encountered such sources. Device Identifiers: Information that uniquely identifies the phone or the user is privacy sensitive. Not all personally identifiable information can be easily tainted. However, the phone contains several easily tainted identifiers: the phone number, SIM card identifiers (IMSI,ICC-ID), and device identifier (IMEI) are all accessed through well-defined APIs. We instrumented the APIs for the phone number, ICC-ID, and IMEI. An IMSI taint source has inherent limitations.

2.4.3.4 Network Taint Sink

Our privacy analysis identifies when tainted information transmits out the network interface. The VM interpreter-based approach requires the taint sink to be placed within interpreted code. Hence, we instrumented the Java framework libraries at the point the native socket library is invoked.

2.5 Machine learning

Machine Learning (ML) techniques on static features that are extracted from Android's application files for the classification of the files as in [12]. Features are extracted from Android's Java byte-code (i.e., .dex files). Evaluation focused on classifying two types of Android applications: tools and games, they performed an evaluation using a collection comprising 2,850 games and tools. The results show that the combination of Boosted Bayesian Networks and the top 800 features selected using Information Gain yield an accuracy level of 0.918 with a 0.172 FPR.

2.5.1 Naïve Bayes

Naïve Bayes is the simplest form of Bayesian Network where in given a class variable, all attributes are assumed to be independent. [13] The algorithm is able to classify by calculating the maximum likelihood of the attributes belonging to a certain class. Even with the interaction of certain attributes, the Naïve Bayes assumption does not lose predictive accuracy even if the actual probabilities are different. [14] An understanding of the Bayes classifier (1) is required to also understand the Naïve Bayes classifier. C is the class of an unobserved random variable to be learned. X denotes a feature vector variable while x denotes the value of the variable. Given the Bayes Classifier, see figure 10 below [14]

$$h^*(x) = \arg\max P(X = x|C = i)\, P(c = i) \;..\; (1)$$

Figure 10 Bayes Classifier

Which determines the maximum a posteriori probability (MAP) given example x, proves difficult in providing direct estimation when there is high-dimensionality in feature space. This is because the Bayes classifier considers a class conditional probability distribution (CPD) defined in which relies on the dependence of each feature vector to another. Figure 11 describes a simplified assumption of the independence of features given the class. [15]

$$f_i^{NB}(x) = \pi_{j=1}^{n} P\left(X_j = x_j \,\middle|C = i\right)P(C = i) \,..(2)$$

Figure 11 Naive Bayes Classifier

2.5.2 Decision Trees

Decision Trees base the classification of instances by sorting feature vectors. In a decision tree, a node represents a feature to be classified and a branch represents the next possible value of a node. Decision trees may be interpreted as a set of rules for each path from the root to each leaf of the tree. [16] The rules employed by decision trees define how a split is created and how cases are classified as to what leaf is reached [16], these rules may also be derived from training data to be used for actual testing. [17]

2.5.3 J48

J48 is an open source Java-based implementation of the C4.5 Decision Tree Algorithm. The algorithm splits the data set to build a certain node for a tree. The data with the highest information gain would be the one that most effectively splits the data set onto one class or another so this certain data is chosen. After choosing the data, a decision node is created to split based on the data chosen. The sub list obtained by splitting on the data with the highest information gain is the recused and then added as children of the decision node. [17]

2.5.4 Random Forest

Random Forest utilizes many classification trees to be able to classify an object based on the majority vote of classification generated by the trees. A tree is grown by first sampling a random number of N cases in the training set. For each input variable M, a number value m is used for each node to select randomly from the input variable to be used to split a node. After, the generated tree is fully grown as deep as possible. [16]

2.5.5 Multinomial Logistic Regression

Since the study classifies more than two types of an Android application, Multinomial Logistic Regression (MLR) has to be utilized to produce polychotomous results over Logistic Regression (LR) which only produces a dichotomous result. In this note, MLR is an extension of LR which provides regression models by comparison of an arbitrary reference category to categories of an unordered response variable. Simply put, MLR utilizes multiple logistic regressions on a multicategory response variable that is unordered. Figure 12 illustrates a general multinomial logistic regression model where is an identified variable and is the reference variable, an explanatory variable affects the resulting model. [18]

$$\log \frac{\Pr(Y = j)}{\Pr(Y = \ j')} = \ \alpha + \ \beta_1 X_1 + \ \beta_2 X_2 + \cdots + \ \beta_k X_k$$

Figure 12 General Equation for MLR

2.6 Summary

Dynamic program analyses have become increasingly popular in security. The one most common dynamic taint analysis is used in a variety of application domains. However, despite it widespread usage, it has been little effort to formally define this analyses and summarize the critical issues that arise when implementing them in a security context.

While some mobile phone operating systems allow users to control applications' access to sensitive information, such as location sensors, camera images, and contact lists, users lack visibility into how applications use their private data. To address this, we present TaintDroid, an efficient, system-wide information flow tracking tool that can simultaneously track multiple sources of sensitive data. A key design goal of TaintDroid is efficiency, and TaintDroid achieves this by integrating four granularities of taint propagation (variable-level, message-level, method-level, and file-level) to achieve a 14% performance overhead on a CPU-bound micro-benchmark. We also used our TaintDroid implementation to study the behavior of 30 popular third-party applications, chosen at random from the Android Marketplace. Our study revealed that two-thirds of the applications in our study exhibit suspicious handling of sensitive data, and that 15 of the 30 applications reported user's locations to remote advertising servers. Our findings demonstrate the effectiveness and value of enhancing smartphone platforms with monitoring tools such as TaintDroid.

Chapter 3

3 Methodology and Design

In this chapter we describes the method we used of how making the detection for mobile application (APKs) by doing deep analysis of types of private data that the application violate the rules and send them via internet or wireless network card and then classify them into two classes benign or malware. Section 3.1 describes briefly the TaintDroid environment that used detect and analysis android applications behavior, then in section 3.2 we illustrate how we collect and gather dataset of taints, at section 3.3 we make some manipulation on data as feature extraction and selection, section 3.4 behavior-based Analysis Module was discussed, then a briefly description of some machine learning classifiers which we will try on the dataset at section 3.5 and finally the proposed solution for the problem as algorithm and pseudo code in section 3.6.

3.1 Introduction

We gather a various collection of android APK samples (both clean & infected), then we install them on our system (modified Android operating system with TaintDroid), the system keep monitoring the applications some time then report all taints notifications and feed them in database (Dataset), this database we use it as dataset for the machine learning classification process, After collecting the dataset, we training the classifier on it, then we can make a testing sample and get results, using AI in this field aiming to make a decision if the samples are normal (benign) or infected (malware).

At the beginning, we install the Linux operating system (Ubuntu 10.04) on a dedicated pc with CPU Intel Core i3 2.4GHz, 4096MB RAM then downloading android source code from official Google website with version Android 4.1.1 (JellyBean), after that we make TaintDroid changes on the source code, finally building the source code again and testing it on Emulator.

3.2 Database (Dataset)

Our database consists of taints of 50 samples APKs, 50% of them are benign and the rest of them malwares.

For the benign applications, they are gathered through the official Android Market and some Alternative Android markets employing a cell phone running on Jellybean 4.1.1 Android OS. The applications are extracted from the cell phone through the use of a File Manager application called Astro. Each application extracted produces. apk file format of the applying and these files are forwarded to VirusTotal to verify their status as benign applications. The table 4 shows imbalanced distribution of applications downloaded for each classification from VirusTotal. Once the applications are processed through the API, most applications are classified as a Trojan, instead of the tagged classification.

Table 3 Count of Applications Downloaded

Type	Number of Apps Collected From VirusTotal	Number of Apps used for Training
Benign	50	10
Trojan	150	10
Virus	70	10
Spyware	30	10
Exploit	45	10

Malware applications are downloaded from online Android malware providers such as VirusTotal and Contagio. Applications downloaded from VirusTotal are tagged as malware by a minimum of 10 anti-virus engines.

Test applications are downloaded both from the Official Android market and some Alternative Android markets. Benign applications are downloaded from the Official Android Market and known-malware applications are downloaded from online Android malware providers such as VirusTotal and Contagio.

Figure 13 below demonstrate the method of building training and testing dataset as application repository, the classified applications have been verified by VirusTotal Malware Verification System.

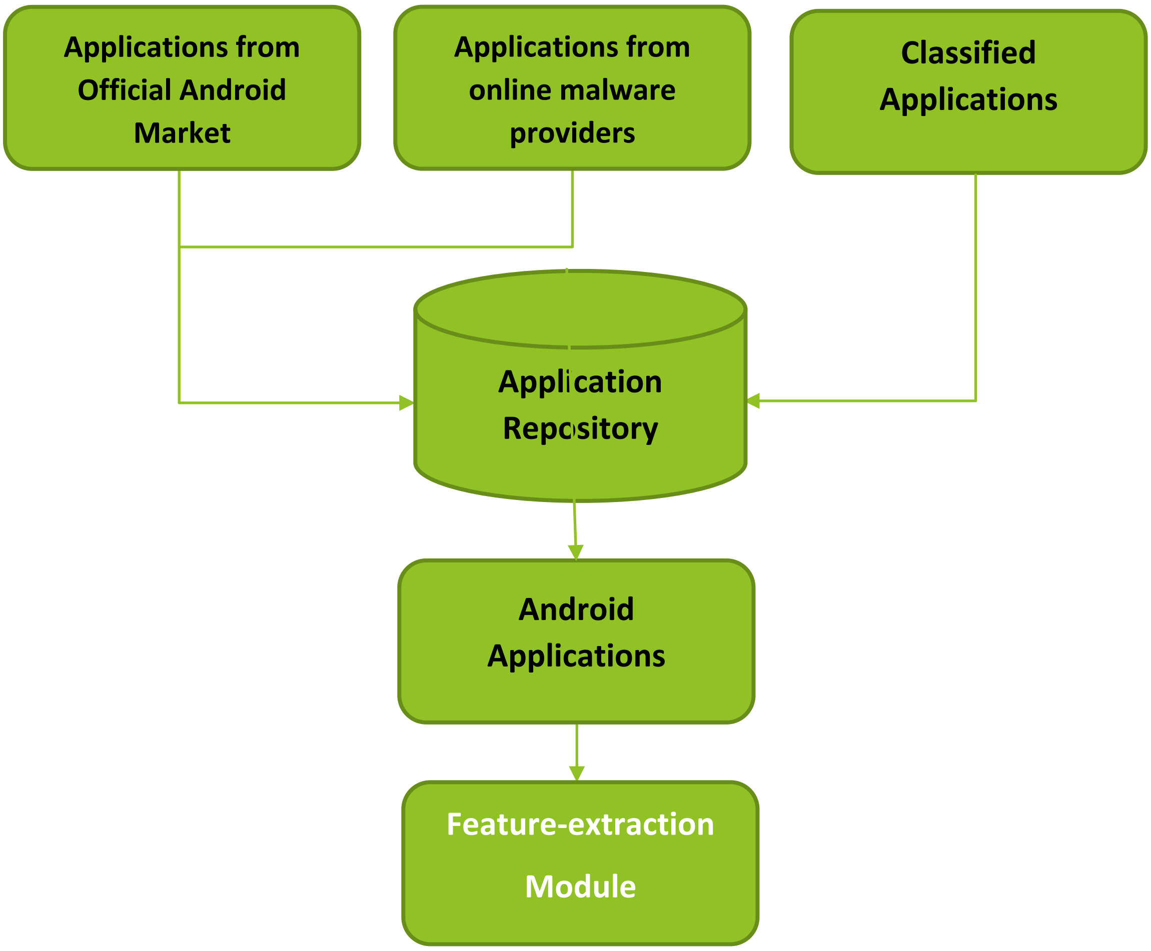

Applications from Official Android Market
Applications from online malware providers
Classified Applications
Application Repository
Android Applications
Feature-extraction Module

Figure 13 Application Acquisition Module

3.3 Feature Extraction

The Feature Extraction Module is the one that generates taints log from running applications retrieved from the application repository of the system. The activity log contains the taints (private data went out from wireless card interface) from application activity which are the features that the module retrieves. For these features to be extracted in Figure 13, the logs are processed by the Virtualization Submodule which handles monitoring and logging of application activity. Features acquired from the Virtualization Submodule are filtered through parsing before being forwarded to the Behavior-based Analysis Module.

Applications are run within the test cell phone Google Nexus S with modified Android version 4.1.1 (TaintDroid) to collect for logs of the taints that system catch of attempting of these applications (as feature extraction) to send any private data over the wireless card. A TaintDroid system, implemented on Android version 4.1.1 to do the job of catching any private data went out from wireless card of the device from any application. I developed an android tool that parsing the log and gathering the taints and stored them into a separate database file along with its classification as benign or any of malware types.

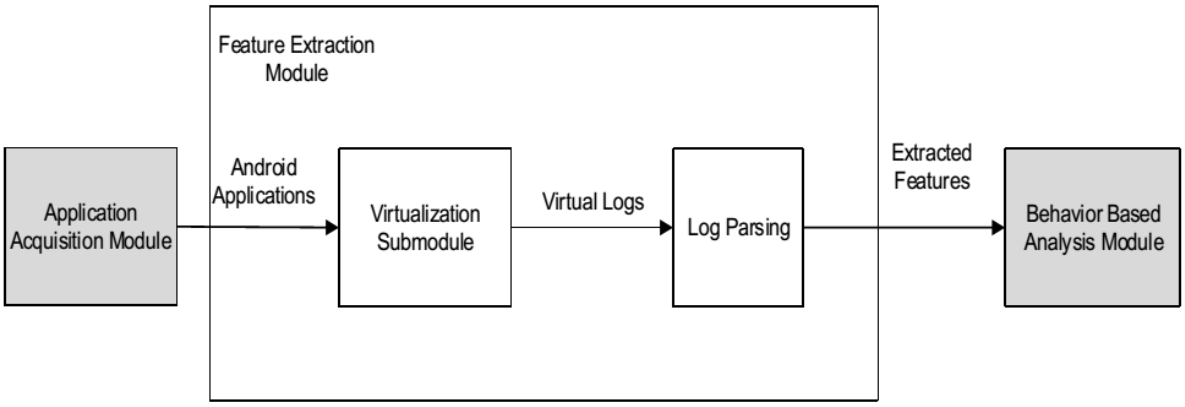

Feature Extraction Module
Application Acquisition Module
Android Applications
Virtualization Submodule
Virtual Logs
Log Parsing
Extracted Features
Behavior Based Analysis Module

Figure 14 Feature Extraction Module

Table 4 below explain the building of taint feature that extracted from the log after parsing process and describes each element of it briefly.

Table 4 Taint Featured extracted

Taint Feature	Description
Application Name	Defines the application which call the taint by sending private data out the cell phone. Ex. com.example.test
Type	Its define the datatype of the private data such as (IMEI, ICCID, SMS, ..)
Destination IP	This is the destination address of which the private data going, it provide also if it SSL or plain.
Message (Data)	The data itself such as like POST, GET request
Date	The date and time.

3.4 Behavior-based Analysis Module

The Behavior-based Analysis Module is responsible for classifying Android applications as either benign or malicious. This is done by employing machine learning algorithms for the generation of behavior models of malicious and benign applications. A training phase, separate from the system, is the one which identifies the behavior of the applications. This module identifies Android applications into two classifications namely: Malware or Benign.

For the training phase, behavior models for each type of Android application are generated by sampling a number of applications per each classification to run on different algorithms.

Features of applications extracted from the previous module are translated into an .arff file format for Weka to be able to process the collected data. Currently, the module only generates the accuracy results of the chosen algorithms given feature sets from each type of malware and of the benign applications.

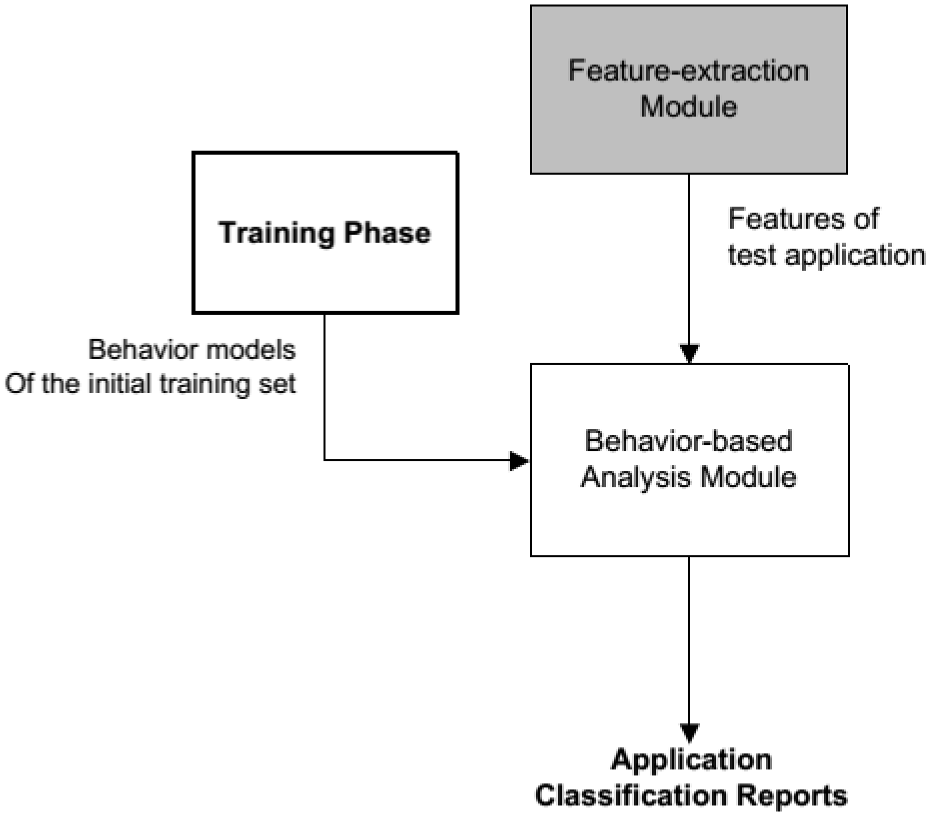
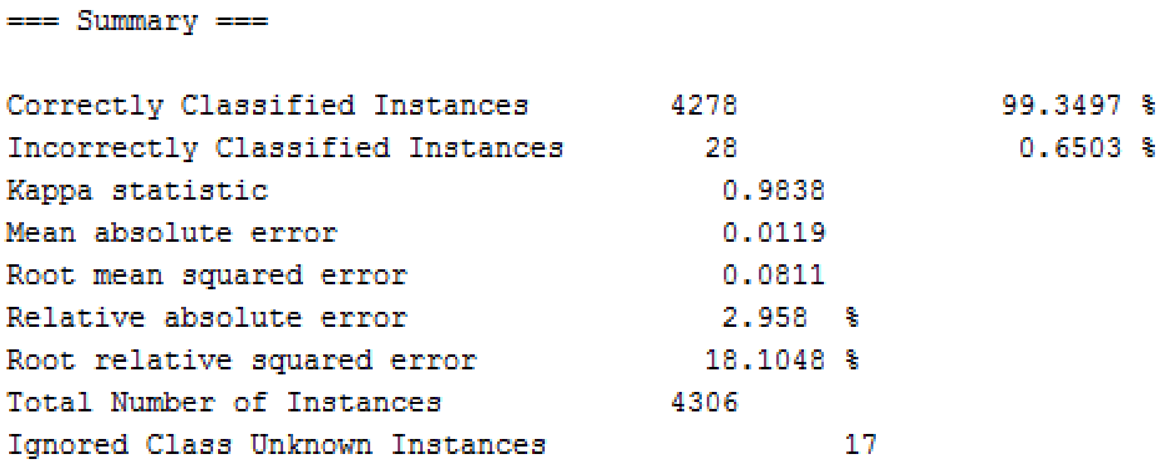

Feature-extraction Module
Training Phase
Features of test application
Behavior models Of the initial training set
Behavior-based Analysis Module
Application Classification Reports

Figure 15 Behavior-based Analysis Module

The algorithms used for this module include the Naïve Bayes algorithm for high bias in small data sets, the Decision Tree algorithms for its low bias and the Logistic Regression algorithm to accommodate for adjustments in the features. [19] Based on studies which used a similar system setup for malware detection, the aforementioned algorithms performed best based on the garnered False Positive Ratings and True Positive Ratio from the tests. [20] [21]. The best performing algorithm based on percentage of correctly classified instances, Kappa statistic, precision, true positive rate and false positive rate.

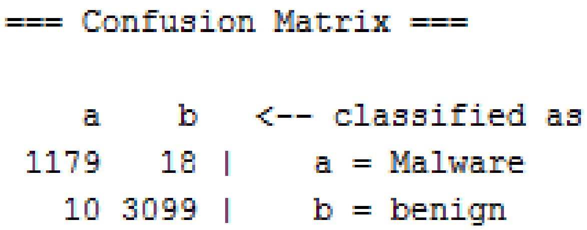


```
=== Summary ===

Correctly Classified Instances        4278               99.3497 %
Incorrectly Classified Instances        28                0.6503 %
Kappa statistic                          0.9838
Mean absolute error                      0.0119
Root mean squared error                  0.0811
Relative absolute error                  2.958  %
Root relative squared error             18.1048 %
Total Number of Instances             4306
Ignored Class Unknown Instances                  17
```

Figure 16 Weka Statistics Result

In the statistics summary above (Fig. 16), the percentage of the correct classified instances is 99% and for the incorrect is 0.65%. The correctly and incorrectly classified instances, often called accuracy or sample accuracy, are the percentage of the test instances that were correctly and incorrectly classified. These also refer to the case where the instances are used as test data. When it comes to classification, correctly and incorrectly classified instances are the most important figures and will be used in the study. [22]With these figures, correctness of the classification of the applications to the different class labels can be determined. The numbers of applications and the classification are shown in the Confusion Matrix below, where a and b are the class labels which in the study’s case, the malware types. There were 40 samples, so when you add up, a + b = 10 + 18.

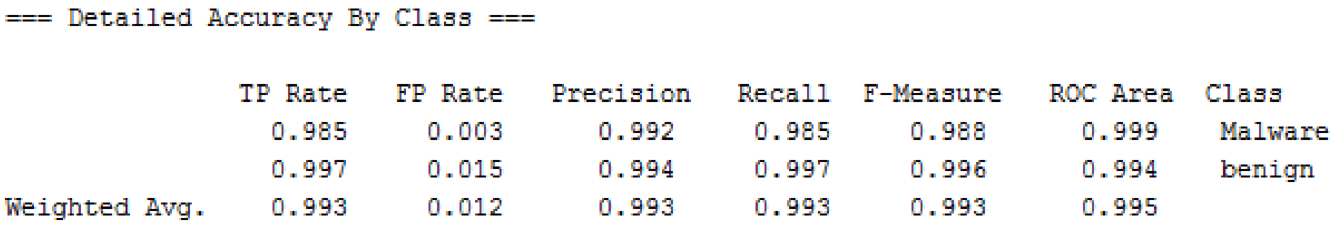


```
=== Confusion Matrix ===

    a    b   <-- classified as
 1179   18 |    a = Malware
   10 3099 |    b = benign
```

Figure 17 Weka Confusion Matrix

Kappa statistic (see Fig. 17) measures the agreement of prediction between the true classes and the classifications. A value greater than 0.0 means that the classifier is doing better than the chance and a value of 1.0 signifies complete or perfect agreement. However, the error rates are used for numeric prediction rather than classification tasks which are not relevant in the study.



```
=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure   ROC Area  Class
                 0.985     0.003     0.992     0.985     0.988      0.999    Malware
                 0.997     0.015     0.994     0.997     0.996      0.994    benign
Weighted Avg.    0.993     0.012     0.993     0.993     0.993      0.995
```

Figure 18 Weka Detailed Accuracy Result

The True Positive (TP) rate is the proportion of applications which were truly classified to a certain class and how much part of the class was captured. It is also equal to the Recall. The percentage of Trojan-labeled applications that are classified as Trojans could be determined using TP. False Positive (FP) rate is the proportion of examples which were classified to a certain class but belongs to a different class. With FP, the percentage of the application classified as Trojans but are Virus labeled can be generated. The Precision is the proportion of the examples which truly belong to a class among those where classified to a specific class. The F-Measure is simply (2*Precision*Recall/(Precision + Recall), a combined measure for Recall and Precision. This could actually be interpreted as the weighted average of Precision and Recall. [23]ROC, on the other hand, is the measure of certainty of the algorithm with the classification made.

3.5 Machine Learning

Different algorithms are tested to see whether which algorithm fares better.

Whilst the algorithms to be tested are the following:

1. J48 (J48graft)
2. Random Forest
3. Multinomial Logistic Regression
4. Naive Bayes.

3.6 Our Algorithm

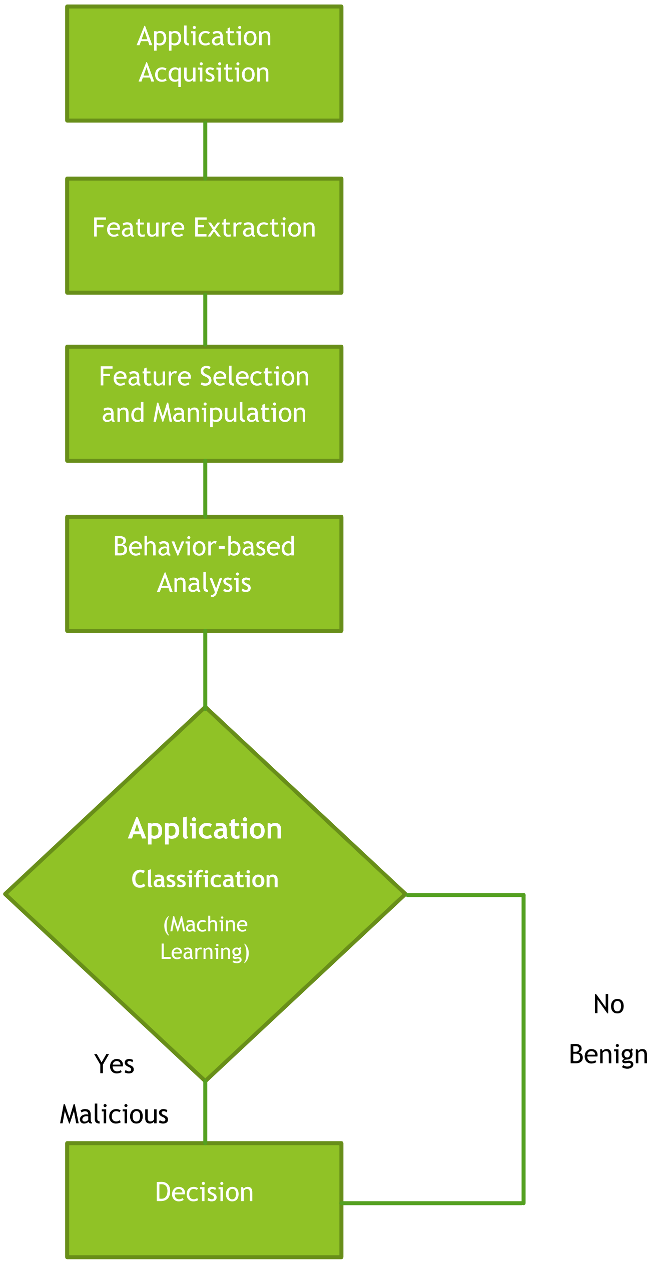


Figure 19 Algorithm Flow Chart

Algorithm starts with application acquisition module as shown in figure 19, it is responsible for application collection and repository, next deploying the application into testing environment to extract features which here is taints, then selections and correction process begin on features to prepare it to next module behavior-based analysis which selection the best machine learning algorithm by try collection of machine learning algorithms, then make applications classification as either benign or malicious.

```
Algorithm AMDA is
Input: benign applications (APK)
       Malware applications (APK)
Output: Decision TPR, ROC
For each APK (type) in repository do
     DB (taints)  ← TanitDroid(APK(t))
DB(taints) modified ← Feature Extraction
DB(taints) modified ← Feature selection
While there exists Taint t in DB(taints) modified do
    Decision(APK) ← Behavior-Based analysis Module (APK taints Collection)
Return Decision(APK)
(Note Decision(APK)  Application classification, Get the Decision if benign or
malicious)
```

Figure 20 Pseudo code algorithm

Figure 20 shows other way to demonstrate and explain the steps of the process in pseudo code form that starts with application acquisition module , it responsible for application collection and repository, next deploying the application into testing environment to extract features which here is taints, then selections and correction process begin on features to prepare it to next module behavior-based analysis which selection the best machine learning algorithm by try collection of machine learning algorithms, then make applications classification as either benign or malicious.

Chapter 4

4 Experiments, Results and Evaluation

In this chapter we focused on the experiments, results and evaluation that we have in this study. Section 4.1 we make a single case study to describe how we test the android application at runtime, analysis its behavior, gathering information and building the taints dataset, then we discussed a real test comparison of some machine learning classifiers and select the best of them, finally we get the results and explain it with charts, then in section 4.2 we illustrate how we obtain the learning data (taints), at section 4.3 we explain the application deployment and the development of tool called TaintCollector to collect the taints from log, section 4.4 log parsing methodology was discussed, then an algorithms tests being made to select the best classifier at section 4.5 and finally classification results and discussion in section 4.6.

Using TaintDroid and cell phone Google Nexus S, we have suitable testing environment, optimization for input generators, record visited view objects, reasoning the “meaning” of view objects and give appropriate input, give “OK” or “Next” higher priority than “Cancel”, generate formatted texts: email, phone number and etc., we decide to use the minimal direct graph approach while testing applications, now we can test if an application leaked our sensitive data in a very short time, in figure 21 shows a schematic diagram, which include ACG (Activity Call Graph) for application as an example.

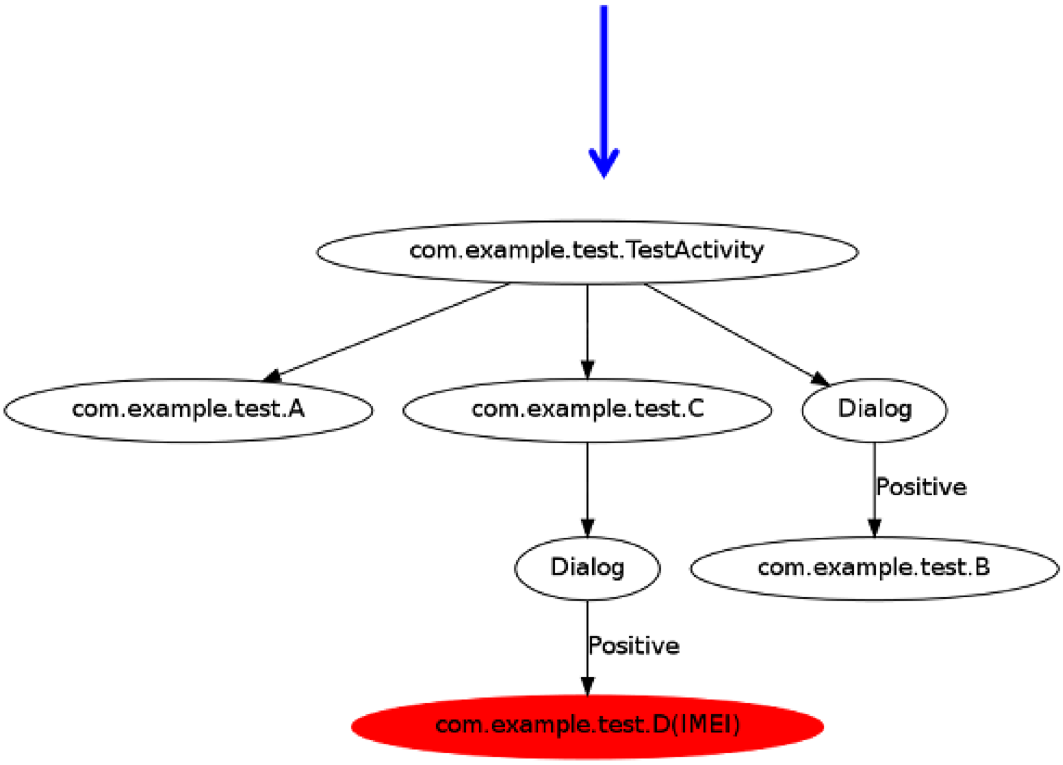
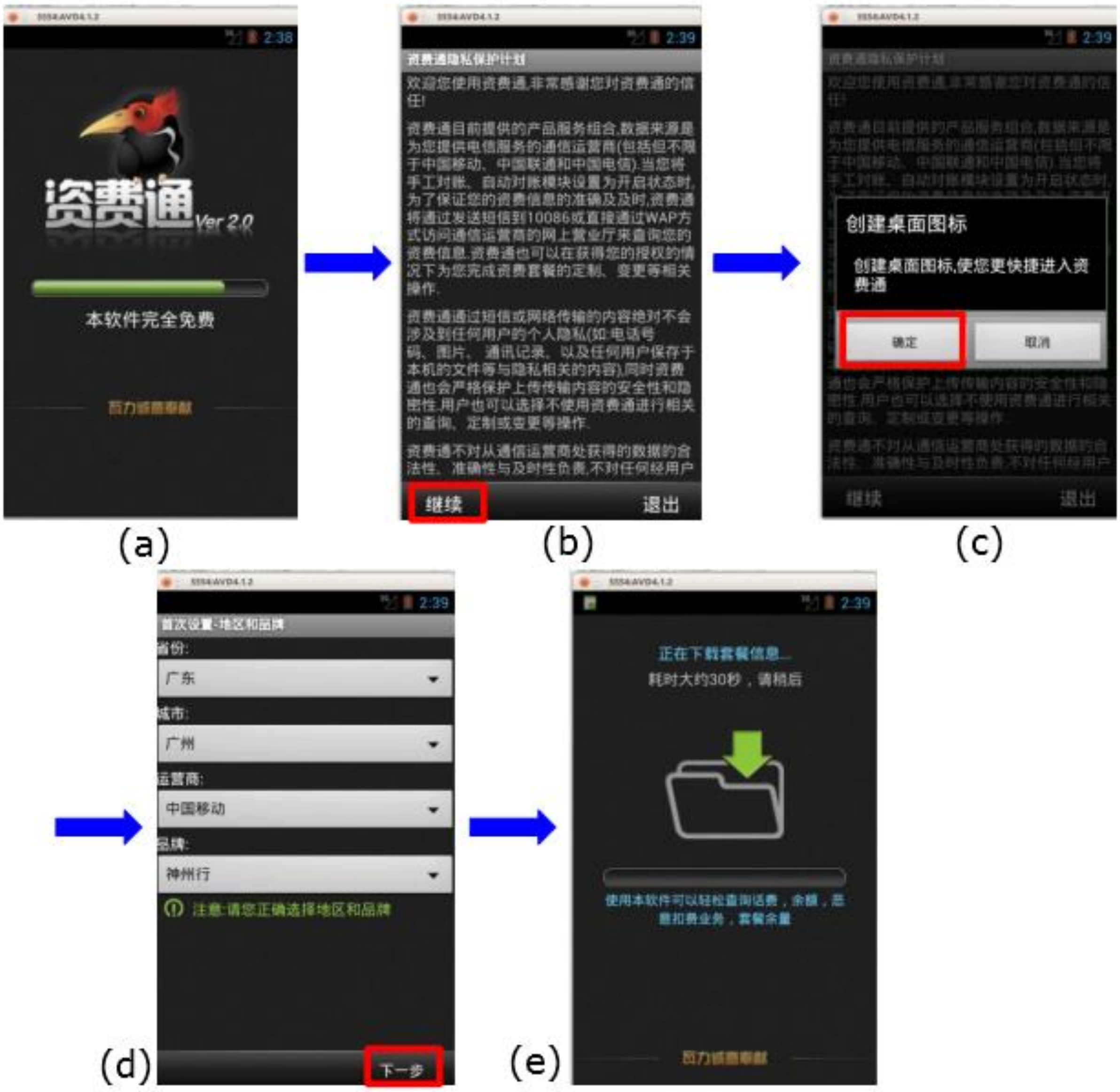


Figure 21 ACG Example

4.1 Single Case Analysis

The Malware App in this example is intended to steal some privacy data. It connects to the Internet and sends the IMEI, IMSI out of your device after you click certain buttons. We consider both behaviors of reading the IMEI and accessing the Internet as sensitive behaviors. Because, the IMEI is the only ID of smartphone devices and accessing the Internet may leak private information.

In figure 22, (a) when this App is started, the Android system creates an instance of the app's main Activity (an "Activity" provides user interfaces) depicted in splash or logo page, which will pause 3 seconds and then start another Activity, (b) description page using an Intent (an Activity is started with an Intent in Android system). (b) There are two buttons for logging or entering into the app details. After you click one of them, (c) dialog pop-up on the screen when you confirm this dialog confirmation message, it will display the Activity shown in Figure 22 (d), which has form after submitting the form, (e) it switches to the final Activity shown in Figure 23. In the last Activity, it reads and sends out the device's IMEI and IMSI by using the sensitive APIs "*android.telephony.TelephonyManager.getDeviceId()*" and "*org.apache.http.client.HttpClient.execute()*" respectively.



资费通隐私保护计划

欢迎您使用资费通,非常感谢您对资费通的信任!

资费通目前提供的产品服务组合,数据来源是为您提供电信服务的通信运营商(包括但不限于中国移动、中国联通和中国电信).当您将手工对账、自动对账模块设置为开启状态时,为了保证您的资费信息的准确及及时,资费通将通过发送短信到10086或直接通过WAP方式访问通信运营商的网上营业厅来查询您的资费信息.资费通也可以在获得您的授权的情况下为您完成资费套餐的定制、变更等相关操作.

资费通通过短信或网络传输的内容绝对不会涉及到任何用户的个人隐私(如:电话号码、图片、通讯记录、以及任何用户保存于本机的文件等与隐私相关的内容),同时资费通也会严格保护上传传输内容的安全性和隐密性.用户也可以选择不使用资费通进行相关的查询、定制或变更等操作.

资费通不对从通信运营商处获得的数据的合法性、准确性与及时性负责,不对任何经用户

创建桌面图标

创建桌面图标,使您更快捷进入资费通

首次设置-地区和品牌

省份: 广东

城市: 广州

运营商: 中国移动

品牌: 神州行

注意:请您正确选择地区和品牌

正在下载套餐信息...

耗时大约30秒，请稍后

使用本软件可以轻松查询话费，余额，恶意扣费业务，套餐余量

Figure 22 Single case study

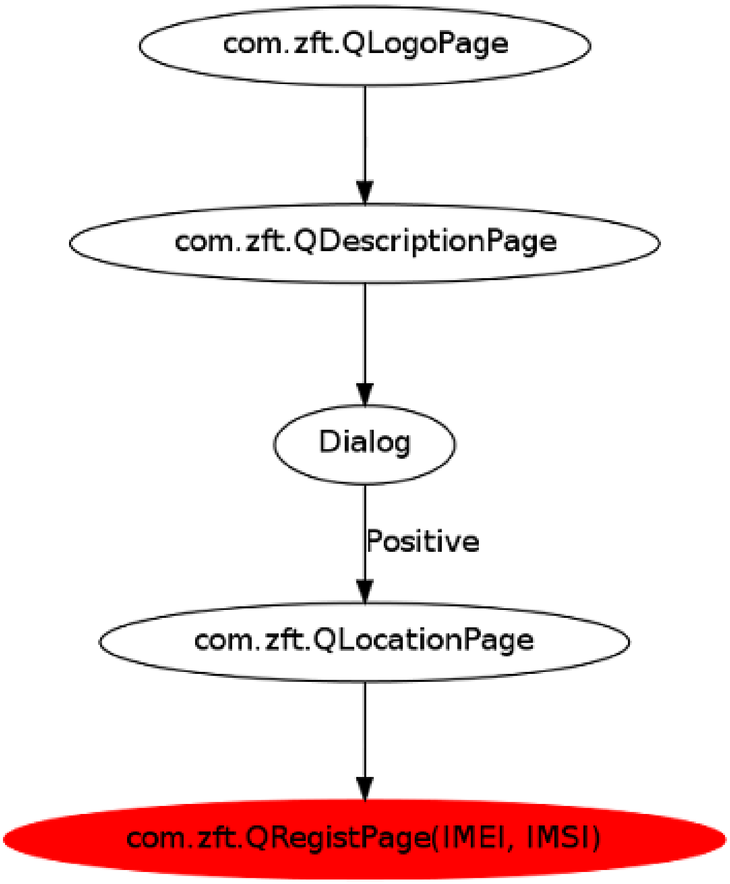


Figure 23 Single case study ACG

TaintDroid detect this process of sending IMEI and IMSI out of the device, after register the taints on file-base database which describes in table 5, taint contains some data, firstly application name, remote address, privacy data, date and time and the body request or message.

The detection of this outgoing privacy data acquire after the data go out from device because of this we can’t prevent this leakage or stolen of data, we just detect and analyze it, then making a decision of either benign or malware.

Table 5 Taint Message Sample

App	Remote Address	Privacy data	Date & Time	Request
Com.ztf.Q	173.255.234.26 (SSL)	IMEI, IMSI	12-23 15:32:40.946	secret=X8ljFrBNQhqMGTy9d4Of0g&sdkversion=2.7.1&hardware-android-device-id=000000000000000&hardware-a]

Case Study #1 with 25 Third-Party Apps

Table 6 Sample of 10 Third-Party Apps

#	App	IMEI	ICCID SIM card identifier	Address Book Contacts Provider	SMS	Location	Browser history	Total
1	com.example.sendimei	4	0	0	0	0	0	**4**
2	com.truecaller	16	0	0	0	0	0	**16**
3	com.netqin.ps	5	0	0	0	0	0	**5**
4	com.anghami	8	0	0	0	0	0	**8**
5	me.onemobile.android	21	15	0	0	0	0	**21**
6	devian.tubemate.home	31	0	0	0	0	0	**31**
7	com.talkray.client	0	0	9	0	0	0	**9**
8	com.syblatv.sybla	0	0	0	0	0	1	**1**
9	com.facebook.katana	0	0	4	671	10	0	**15**
10	com.nimbuzz	6	0	69	1398	1162	0	**24**

In this case study no. 1 in table 6 we test over twenty five various third party application, firstly we get it from application repository, then deploying into the testing environment TaintDroid. After that run each application for some time and try to click many explicit buttons, review major activities and screens, next waiting for the application to send privacy data out from the device, when it send any explicit data, TaintCollector tool will detect this leakage of data by monitoring and finally registered to the database. This statistics measured in table 6, it's from detected taints on the database it demonstrate the number of privacy data outgoing some application and total calculations.

After repeating the same test in case study no. 2 but with various families of malwares. This statistics measured in table 7, it's from detected taints on the database it demonstrate the number of privacy data outgoing some application and total calculations.

Case Study #2 with 25 Malware Apps

Table 7 Sample of 10 Malware Apps

#	App	IMEI	ICCID SIM card identifier	Address Book Contacts Provider	SMS	Location	Browser history	Total
1	com.zeal.zealspydesign	0	0	0	20	0	0	20
2	com.hellospy.system	471	0	68	0	0	61	600
3	com.inospy	1	0	0	0	0	0	1
4	com.maher.wieghtcalculate	91	0	0	21	0	0	112
5	com.one.pushnew	25	0	0	6	17	0	48
6	com.tutusw.onekeyvpn	0	0	0	70	0	0	70
7	com.Beauty.Breast	12	0	0	12	0	0	24
8	com.Beauty.Leg	16	0	0	16	0	0	32
9	com.keji.unclear	291	0	0	47	47	0	385
10	mobi.rouwan.timezoneconvertor	49	0	0	9	3	0	61

4.2 Learning Data Acquisition

Known-malware applications are gathered manually from the expert system VirusTotal which provides credibility for the learning data. VirusTotal uses an observed minimum of forty Anti-virus engines which scans the applications. Applications downloaded from VirusTotal are tagged as malware by at least ten anti-virus engines. For the known-benign applications, they are gathered through the official Android Market using a mobile phone running on Jellybean 4.1.1 Android OS. The applications are extracted from the mobile phone through the use of a File Manager application called Astro. Each application extracted produces an .apk file format of the application and these files are forwarded to VirusTotal to verify their status as benign applications.

Table 8 Count of Applications Downloaded

Type	Number of Apps Collected from VirusTotal	Number of Apps used for Training
Benign	25	18
Malware	25	18

The table 8 shows an imbalanced distribution of applications downloaded for each classification from VirusTotal. When the applications are processed through the API, most applications are classified as Trojan, instead of the tagged classification.

4.3 Application Deployment

Applications are run inside a real physical device phone with model Google Nexus S within modified android version 4.1.1 (TaintDroid) to collect for logs of the behavior of these applications to be used by Weka.

A tool, TaintCollector, is used to obtain the outgoing privacy data (taints) made by the application that is running. These taints that have been collected are then stored into a separate file along with its classification as benign or as one of the types of malware.

4.4 Application Log Parsing

The application taints that have been generated by TaintCollector are collected and injected into the parser program. This parser program generates the ARFF (Attribute Relation File-Format) file to be used by Weka in classifying the applications. The program searches for specific taints made by the application inside the log file. The count of these taints are taken and then appended into the ARFF file. This is done for all desired taints to be taken for all the application logs.

4.5 Algorithm Testing

The ARFF file generated by the parser program is fed into Weka for the classification of the applications. Different algorithms are tested to see whether which algorithm fares better. The metrics to be checked for are the following:

1. True Positive Rate
2. Kappa Statistic

3. Receiver Operating Characteristic (ROC)

Whilst the algorithms to be tested are the following:

5. J48 (J48graft)
6. Random Forest
7. Multinomial Logistic Regression
8. Naive Bayes.

The behavior logs of test applications are produced and then processed through the use of the parser for test applications. The parser works with a set dictionary of relevant features based from the features of the behavior model from the training phase. An ARFF file for each test application is produced by the parser which is used for comparison with the behavior-model of the most accurate algorithm. The applications for this test are downloaded from different sources. These test applications came from VirusTotal and as well as form alternative Android markets namely: aptoide, mobogenie and 1Mobilemarket.

AMDA application classification results stored in the database are compared to the classification report from VirusTotal. AMDA collates the results and uses a counting mechanism as to how many tags of applications are made by the AV engines as Trojan, Spyware, Exploit, Virus and Unclassified. Again, the performance of the system is measured by the True Positive Rate, Kappa Statistic and the Receiver Operating Characteristic (ROC Curve).

4.6 Classification Results & Discussion

The training phase for the system undergoes a rigorous process for being able to generate the best behavior model for the system. There is a total of 30 number of tests made. As mentioned in the earlier section, each algorithm is tested with three different feature selection methods and without a feature selection method used. This is done four times and for each test, the number of applications used varied in number.

Table 9 Number of Applications per Training Phase

Training Phase	Number of Applications
Test 1	10
Test 2	20
Test 3	30
Test 4	36

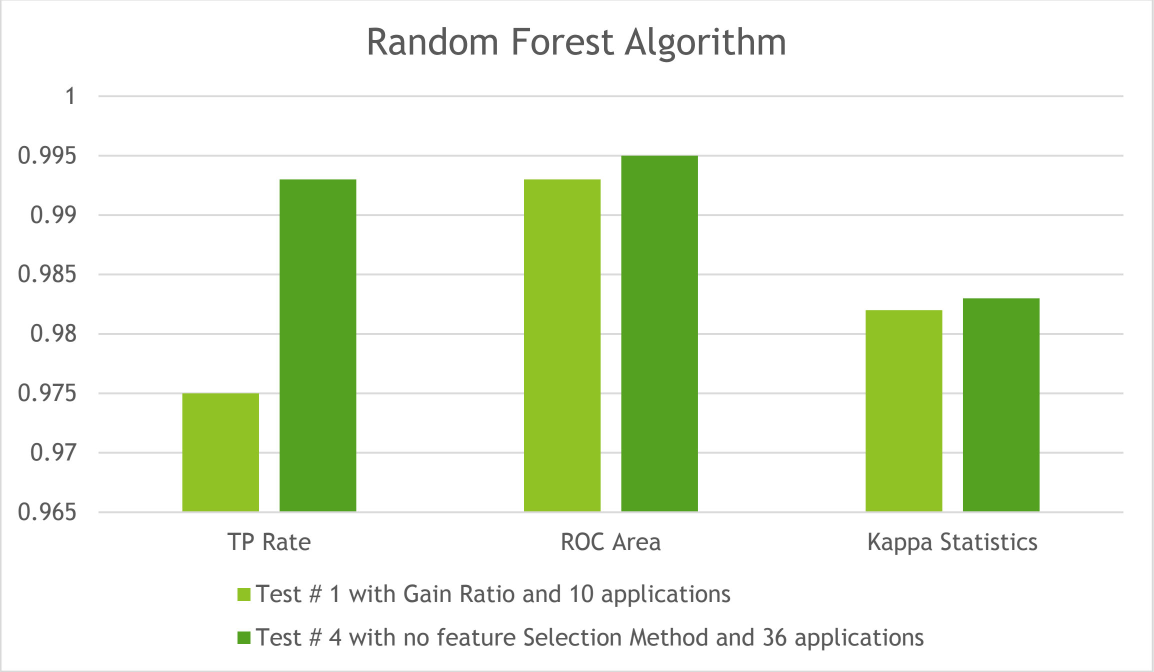


Figure 24 Summarized Results of Machine Learning Algorithms

For the training set, the Random Forest algorithm in Test 1 with Gain Ratio as the feature selection method and Test 4 with no feature selection method (See Figure. 24) garnered the best accuracy through measurement by True Positive Rate. Both tests achieved 90% accuracy. In addition, the Random Forest algorithm consistently outperformed the other algorithms. The behavior model from Training Phase Test 4 is chosen to be used for the system since it performed well even with a larger number of applications used and the test garnered a higher rate of ROC which means that the algorithm is definite with its classifications.

After knowing the best algorithm for classification through the training phase, gathering and processing of test applications follows.

Table 10 Applications Classified by AMDA

Type of Android Application	Number of Applications Classified
Benign	18
Malware	15

There are a total of 33 applications parsed by the system. The results are compared to the classification report from VirusTotal.

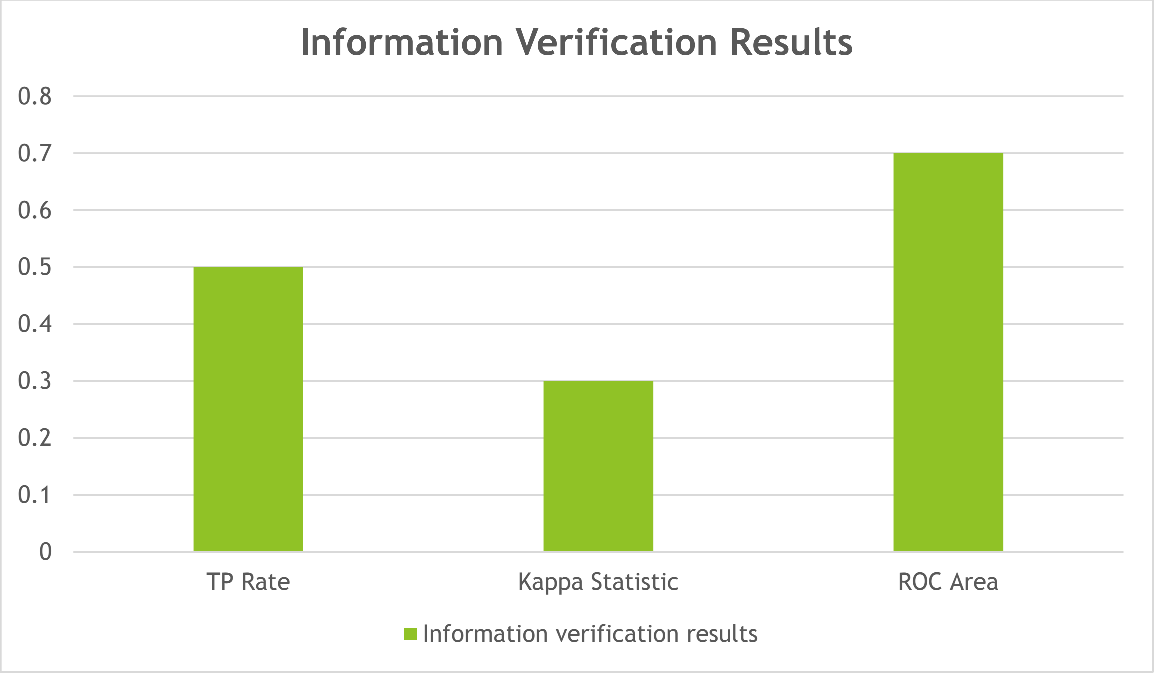


Figure 25 Information Verification Results

When the results of the AMDA System are validated to the results garnered through VirusTotal, TP Rate measurement exacted to 46.2%, Kappa Statistic to 27.17% and the ROC Area measured 67.5%. The results above constitute quite a low accuracy for classification of the types of Android applications. The ROC Area, being above the 50% mark, means that the system is mostly certain of its classifications. A low measure was garnered by the Kappa Statistic which means that the system encountered a dataset with mostly random attributes. Further checking deep into the taints is made to identify other measurement of analysis and problems. It is found that fourteen of the taints features exhibit the same characteristics for pairs of malware types. Virus and Exploit applications typically measure the same for these taints. Trojan and Spyware applications are paired for the mentioned taints. With that information, it can be derived that malware applications exhibit the same behaviors which explains why the results of the classification is low. Instead of having 4 classifications for Malware, it is simplified into just Malware versus Benign classifications.

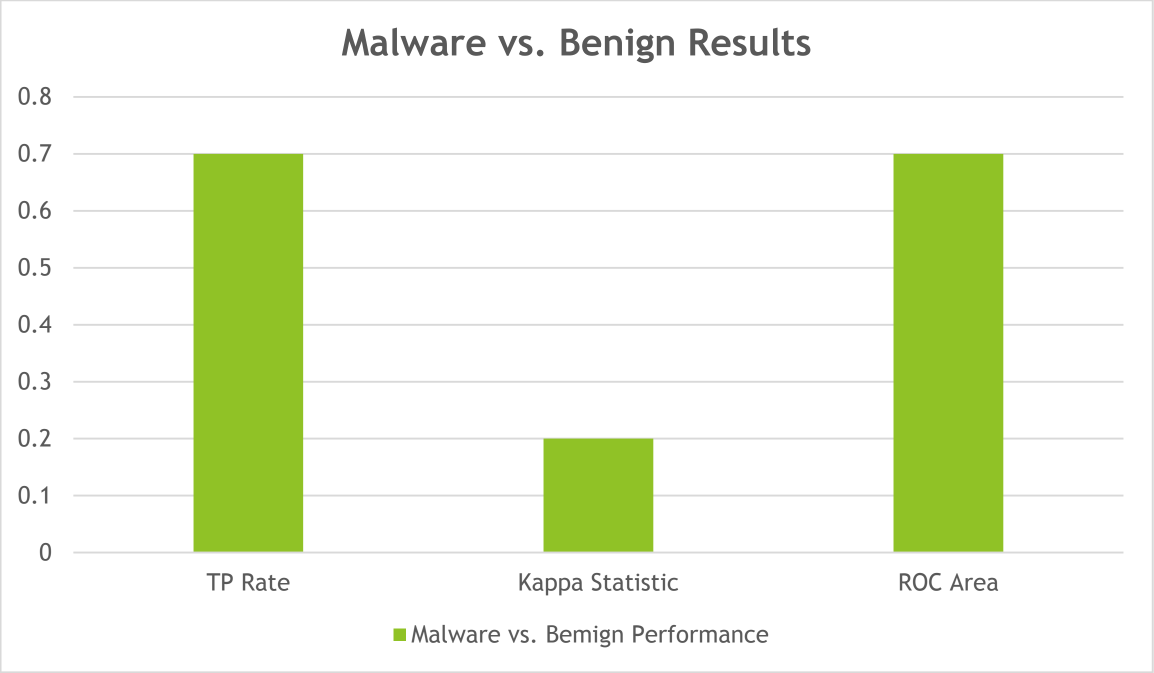


Figure 26 Malware vs. Benign Results

The TP Rate measurement increased to 74.7%, the Kappa Statistic to 23.17% and the ROC area as 72.1% in figure 26. The TP Rate achieved a significantly higher value percentage compared to the previous result which indicates that the system is able to better correctly classify the applications. The Kappa Statistic measured is almost the same as the previous test. This is expected since the same dataset is used as with the previous test. The ROC Area still achieved a high measurement which indicates that the system is mostly certain of the classifications made.

Chapter 5

5 Conclusion and Future Work

In this chapter, we describes and summarizes the core findings of this thesis. Section 5.1 presented the derived conclusion of our study and some advices and points for researchers to complete the research in the same field in future work at section 5.2.

5.1 Conclusion

Finally, we have two contribution in this thesis, first of all we gather more than 50 android applications taints into dataset or database, 50% of them normal applications (benign) and the rest are malwares as we discussed before, this dataset rarely exist on the internet and no one provide it before.

The second that our system given, the capability to classify unknown applications based from its data can be used to categorize different Android applications in the market. With the web crawler at hand, the system has the potential to automatically download and classify new applications uploaded to the different alternative markets. Other than these, the system has the ability to classify malware to different types using behavior-based analysis. With this at hand, the system can act as an Anti-Virus that could easily provide classification results to users. However, expert systems or different classification sources change classifications from time to time. This happens when more Anti-virus engines are able to classify applications as from when the application was first classified or because there are more and more malware families being identified. With this, there is a clear lack of standards in the classification scheme of applications. This lack of standards contributes to the futility of classifying malware into different classifications other than just classifying it as malware. Another factor would be that malware families would have variants of other malware families which makes it even more difficult to distinguish between malware types [24].

This study tries to take the lead of the way of detection malwares using dynamic analysis in specific dynamic taint analysis this method based on android application analysis at run time then monitoring and logging the information flow out of the device from any port like wireless card interface or Bluetooth, specially private data and secure info such as credit card info, SMS, contacts, IMEI . etc, Our malware dataset consist of 50 Android applications for this research 50% of them benign and the rest malwares. Finally we feed the machine learning algorithm with data to classify it and we measure the accuracy and detection ratio it reach 74.7% this result being satisfied and good enough because of variety of malwares in real life and difficulties on classifying them such like Trojans, spywares, exploits and viruses application.

5.2 Future Work

Further work to be done is to gather more samples for each type of Android application for further testing of the machine learning algorithms. More features are also to be added for monitoring to be able to create more distinction with regards to the behavior of the applications. A behavior model will then be generated to be able to classify Android applications. Classification results of the system are to be verified with VirusTotal for its validity as a metric to test if the generated behavior models are accurate.

Also further work to be done is the ability to detect advanced malware attacks such as Zero-day attack. Implementation of Behavior-based analysis with permission-based can also be done to determine malicious Android applications. Administrative User interface and an AMDA Android Application will allow easier analysis and access of the system.

6 References

[1] Symantec, "INTERNET SECURITY THREAT REPORT 2015," APRIL 2015. [Online]. Available: https://www4.symantec.com/mktginfo/whitepaper/ISTR/21347932_GA-internet-security-threat-report-volume-20-2015-social_v2.pdf.

[2] D. Bryan and M. Shivakant, "On Rootkit and Malware Detection in Smartphones," in *IEEE*, 2010.

[3] W. Te-En, Ching-Hao, B. J. Albert, L. Hahn-Ming, W. Horng-Tzer and W. Dong-Jie, "Android Malware Detection via a Latent Network Behavior Analysis," in *IEEE*, 2012.

[4] I. Takamasa, T. Keisuke and K. Ayumu, "Kernel-based Behavior Analysis for Android Malware Detection," in *IEEE*, 2011.

[5] X. Luo, "Access Control Research Based on Trusted Computing Android Smartphone," in *IEEE*, 2013.

[6] Y. ZheMin and Y. Min, "LeakMiner: Detect information leakage on Android with static taint analysis," in *IEEE*, 2012.

[7] MoutazAlazab, L. B. VeelashaMoonsamy and L. Patrik, "Analysis of Malicious and Benign Android Applications," in *IEEE*, 2012.

[8] B. Thomas, B. Leonid, S. Aubrey-Derrick, A. C. Seyit and A. Sahin, "An Android Application Sandbox System for Suspicious Software Detection," in *IEEE*, 2010.

[9] B. M., F. F. and L. and B., "On the effort to create Smartphone worms in windows mobile," in *IEEESMC*, 2007.

[10] E. Schwartz, T. Avgerinos and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)," in *Security and Privacy (SP), 2010 IEEE Symposium on*, 2010.

[11] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel and A. Sheth, "TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones.," in *OSDI*, 2010.

[12] S. Asaf, F. Yuval and E. Yuval, "Automated Static Code Analysis for Classifying Android Applications UsingMachine Learning," in *IEEE*, 2010.

[13] P. F. a. N. Lachiche, "Naïve Bayesian Classification of Structured Data," 1 November 2012. [Online]. Available: http://www.cs.bris.ac.uk/~flach/papers/mlj04-1BC-final2.pdf.

[14] H. Zhang, "The Optimality of Naïve Bayes," 1 November 2012. [Online]. Available: http://courses.ischool.berkeley.edu/i290-dm/s11/SECURE/Optimality_of_Naive_Bayes.pdf.

[15] I. Rish, "An Emprical Study of the naïve Bayes Classifer," 5 November 2012. [Online]. Available: http://www.cc.gatech.edu/~isbell/reading/papers/Rish.pdf.

[16] L. B. a. A. Cutler, "Random Forests," 3 November 2012. [Online]. Available: http://statwww.berkeley.edu/users/breiman/RandomForests/cc_home.htm#intro.

[17] I. D. Z. a. P. E. P. S. Kotsiantis, "Supervised Machine Learning: A Review of Classification and Combining Techniques," 2 November 2012. [Online]. Available: http://www.cs.bham.ac.uk/~pxt/IDA/class_rev.pdf.

[18] L. M. a. G. Hutcheson, "Dictionary of Quantitative Methods in Management," 4 November 2012. [Online]. Available: http://www.researchtraining.net/addedfiles/READING/MNLmodelChapter.pdf.

[19] Oracle, "Data Mining Concepts: Regression," 30 October 2012. [Online]. Available: http://docs.oracle.com/cd/B28359_01/datamine.111/b28129/regress.htm#DMCON005.

[20] B. Sans, "On the Automatic Categorisation of Android Applications," 5 June 2015. [Online]. Available: http://paginaspersonales.deusto.es/isantos/publications/2012/Sanz_2012_CCNC_Android_Apps_Categorisation.pdf.

[21] A. S. a. C. Glezer, "Andromaly a behavioral malware detection framework for android devices," 2010. [Online]. Available: http://posgrado.escom.ipn.mx/biblioteca/%E2%80%9CAndromaly%E2%80%9D%20a%20behavioral%20malware%20detection.pdf.

[22] J. Tiedemann, "Interpreting Weka Output," 5 November 2012. [Online]. Available: http://www.let.rug.nl/tiedeman/ml06/InterpretingWekaOutput.

[23] M. J. P. K. a. M. J. T.Borovicka, "Selecting Representative Data Sets," 2 December 2012. [Online]. Available: http://cdn.intechopen.com/pdfs/39037/InTechSelecting_representative_data_sets.pdf.

[24] E. Labs, "Trends for 2013: Astounding growth of mobile," 2013. [Online]. Available: http://go.eset.com/us/resources/whitepapers/Trends_for_2013_preview.pdf.

[25] K. C. a. A. Y. J. Chan, "Detecting the Nature of Change in an Urban Environment: A Comparison of Machine Learning Algorithms," *American Society for Photogrammetry and Remote Sensing,* pp. 213-225, 2001.

[26] T. B. a. e. al, "An Android Application Sandbox System for Suspicious Software Detection," in *Proceedings of the 5th IEEE International Conference on Malicious and Unwanted Software*, 2010.

7 Appendices

Appindex1: TaintCollector Code.

- TaintDroidCollector.java

```
package org.app.analysis;
import org.app.analysis.R;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.View;
import android.view.View.OnClickListener;
import android.widget.Button;

public class TaintDroidCollector extends Activity {
    private static final String TAG =
TaintDroidCollector.class.getSimpleName();
    private OnClickListener onClickStartButton = new OnClickListener() {
        public void onClick(View v) {
            Intent i = new Intent(getApplicationContext(),
TaintDroidCollectorService.class);
            startService(i);
        }
    };
    private OnClickListener onClickStopButton = new OnClickListener() {
        public void onClick(View v) {
            Intent i = new Intent(getApplicationContext(),
TaintDroidCollectorService.class);
            stopService(i);
        }
    };

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.control);
        Button b = (Button) findViewById(R.id.StartButton);
        b.setOnClickListener(onClickStartButton);
        b.setEnabled(true);
        b = (Button) findViewById(R.id.StopButton);
        b.setOnClickListener(onClickStopButton);
        b.setEnabled(true);
    }
}
```

- TaintDroidCollectorDetail.java

```
package org.app.analysis;
import org.app.analysis.R;
import android.app.Activity;
import android.os.Bundle;
import android.util.Log;
import android.widget.TextView;
public class TaintDroidCollectorDetail extends Activity {
    private void setInfo() {
        Bundle b = getIntent().getExtras();
        String appname =
b.getString(TaintDroidCollectorService.KEY_APPNAME);
        String ipaddress =
b.getString(TaintDroidCollectorService.KEY_IPADDRESS);
        String taint = b.getString(TaintDroidCollectorService.KEY_TAINT);
        String data = b.getString(TaintDroidCollectorService.KEY_DATA);
        int id = b.getInt(TaintDroidCollectorService.KEY_ID);
        String timestamp =
b.getString(TaintDroidCollectorService.KEY_TIMESTAMP);

        TextView tv = (TextView) findViewById(R.id.DetailAppTextView);
        tv.setText(appname);

        tv = (TextView) findViewById(R.id.DetailIPTextView);
        tv.setText(ipaddress);

        tv = (TextView) findViewById(R.id.DetailTaintTextView);
        tv.setText(taint);

        tv = (TextView) findViewById(R.id.DetailDataTextView);
        tv.setText(data);

        tv = (TextView) findViewById(R.id.DetailTimestampTextView);
        tv.setText(timestamp);
    }

    @Override
    public void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.detail);
        setInfo();
    }
}
```

- TaintDroidCollectorService.java

```
package org.app.analysis;
import java.io.*;
import java.util.*;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.net.InetAddress;
import java.net.UnknownHostException;
import java.util.concurrent.*;
import org.app.analysis.R;
import android.annotation.SuppressLint;
import android.app.ActivityManager;
import android.app.Notification;
import android.app.NotificationManager;
import android.app.Notification.Style;
import android.app.PendingIntent;
import android.app.Service;
import android.app.ActivityManager.RunningAppProcessInfo;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.net.Uri;
import android.os.Bundle;
import android.os.IBinder;
import android.util.Log;
import android.widget.Button;
import android.widget.RemoteViews;

public class TaintDroidCollectorService extends Service {
    private static final String TAG =
TaintDroidCollectorService.class.getSimpleName();
    private static final String filename =  "taints.csv";
    private static Hashtable<Integer, String> ttable = new
Hashtable<Integer, String>();
    static {
        // ttable.put(new Integer(0x00000000), "No taint");
        ttable.put(new Integer(0x00000001), "Location");
        ttable.put(new Integer(0x00000002), "Address Book
(ContactsProvider)");
        ttable.put(new Integer(0x00000004), "Microphone Input");
        ttable.put(new Integer(0x00000008), "Phone Number");
        ttable.put(new Integer(0x00000010), "GPS Location");
        ttable.put(new Integer(0x00000020), "NET-based Location");
        ttable.put(new Integer(0x00000040), "Last known Location");
        ttable.put(new Integer(0x00000080), "camera");
        ttable.put(new Integer(0x00000100), "accelerometer");
        ttable.put(new Integer(0x00000200), "SMS");
        ttable.put(new Integer(0x00000400), "IMEI");
        ttable.put(new Integer(0x00000800), "IMSI");
        ttable.put(new Integer(0x00001000), "ICCID (SIM card identifier)");
        ttable.put(new Integer(0x00002000), "Device serial number");
        ttable.put(new Integer(0x00004000), "User account information");
        ttable.put(new Integer(0x00008000), "browser history");
    }

    private volatile static boolean isRunning = false;

    public static final String KEY_APPNAME = "KEY_APPNAME";
    public static final String KEY_IPADDRESS = "KEY_IPADDRESS";
    public static final String KEY_TAINT = "KEY_TAINT";
    public static final String KEY_DATA = "KEY_DATA";
    public static final String KEY_ID = "KEY_ID";
    public static final String KEY_TIMESTAMP = "KEY_TIMESTAMP";

    public static class Starter extends BroadcastReceiver {
        @Override
        public void onReceive(Context context, Intent intent) {
            if(!isRunning && intent.getAction() != null) {
                if(intent.getAction().equals(Intent.ACTION_USER_PRESENT)) {
                    context.startService(new Intent(context,
TaintDroidCollectorService.class));
                }
            }
        }
    };

    private BlockingQueue logQueue;
    private static final int LOGQUEUE_MAXSIZE = 4096;
    private volatile boolean doCapture = false;
    private Thread captureThread = null;
    private class Producer implements Runnable {
      private final BlockingQueue queue;
      Producer(BlockingQueue q) { queue = q; }
        public void run() {
            LogcatDevice lc = LogcatDevice.getInstance();
            while(doCapture && lc.isOpen()) {
                try {
                    // read an entry and insert it to our content provider
                    LogEntry le = lc.readLogEntry();
                    if(le != null) {
                        queue.put(le);
                    }
                }
                catch(Exception e) {
                    Log.e(TAG, "Could not read a log entry: " +
e.getMessage());
                    e.printStackTrace();
                }
            }
        }
    };

    private volatile boolean doRead = false;
    private Thread readThread = null;
    private class Consumer implements Runnable {
      private final BlockingQueue queue;
      Consumer(BlockingQueue q) { queue = q; }
        public void run()
        {
            LogEntry prev = null;
            while(doRead) {
                  try {
                        LogEntry le = (LogEntry)queue.take();
                    processLogEntry(le);
                  }
                  catch (InterruptedException e) {
                        Log.e(TAG, "Could not read log entry: " +
e.getMessage());
                  }
            }
        }
    }

    private String get_processname(int pid) {
        ActivityManager mgr = (ActivityManager)
getApplicationContext().getSystemService(
            Context.ACTIVITY_SERVICE);

        String pname = "";
        List<RunningAppProcessInfo> apps = mgr.getRunningAppProcesses();
        for(RunningAppProcessInfo pinfo : apps) {
            if(pinfo.pid == pid) {
                pname = pinfo.processName;
                break;
            }
        }

        return pname;
    }
```

```
private String get_ipaddress(String msg) {
  Pattern p = Pattern.compile("\\((.+)\\) ");
    Matcher m = p.matcher(msg);

    if(m.find() && m.groupCount() > 0) {
        String result = m.group(1);
        // remove trailing junk
        if (result.contains(")"))
              result = result.substring(0,result.indexOf(")")-1);
        return result;
    }
    else {
        return null;
    }
}

private String get_taint(String msg) {
  // match hex digits
  Pattern p = Pattern.compile("with tag 0x(\\p{XDigit}+) ");
    Matcher m = p.matcher(msg);

    if(m.find() && m.groupCount() > 0) {

        String match = m.group(1);

        // get back int
        int taint;
        try {
            taint = Integer.parseInt(match, 16);
        }
        catch(NumberFormatException e) {
            return "Unknown Taint: " + match;
        }

        if(taint == 0x0) {
            return "No taint";
        }

        // for each taint
        ArrayList<String> list = new ArrayList<String>();
        int t;
        String tag;

        // check each bit
        for (int i=0; i<32; i++) {
              t = (taint>>i) & 0x1;
              tag = ttable.get(new Integer(t<<i));
            if(tag != null) {
                list.add(tag);
            }
        }

        // build output
        StringBuilder sb = new StringBuilder("");
        if(list.size() > 1) {
            for(int i = 0; i < list.size() - 1; i++) {
                sb.append(list.get(i) + ", ");
            }
            sb.append(list.get(list.size() - 1));
        }
```

```
            else {
                if(!list.isEmpty()) {
                    sb.append(list.get(0));
                }
            }

            return sb.toString();
        }
        else {
            return "No Taint Found";
        }
    }

    private String get_data(String msg) {
        int start = msg.indexOf("data=[") + 6;
        return msg.substring(start);
    }

      private int noti_id = 0;

    private void sendTaintDroidNotification(int id, String ipaddress, 
String taint, String appname, String data, String timestamp) {
        Notification notification = new Notification.BigTextStyle(
        new Notification.Builder(this)
        .setContentTitle("TaintDroid alert")
        .setContentText(appname)
        .setSmallIcon(R.drawable.icon))
        .bigText(appname+"\n"+ipaddress+"\n"+taint)
        .build();

        // set intent to launch detail
        Bundle extras = new Bundle();
        extras.putString(KEY_APPNAME, appname);
        extras.putString(KEY_IPADDRESS, ipaddress);
        extras.putString(KEY_TAINT, taint);
        extras.putString(KEY_DATA, data);
        extras.putInt(KEY_ID, id);
        extras.putString(KEY_TIMESTAMP, timestamp);


        //save to file
        String path = 
getApplicationContext().getFilesDir().getAbsolutePath();
        String info = id  + " , " + appname + " , " + ipaddress + " , " + 
                        taint + " , " + timestamp + " , " + data +" \n";
        FileOutputStream outputStream;

        try { //path + "/" +
          outputStream = openFileOutput(filename, MODE_APPEND | 
MODE_PRIVATE);
          outputStream.write(info.getBytes());
          outputStream.close();
         }catch (IOException e) {
              Log.e("Exception", "File write failed: " + e.toString());
          }

        Intent i = new Intent(this, TaintDroidCollectorDetail.class);
        i.addFlags(Intent.FLAG_ACTIVITY_NO_HISTORY);
        i.putExtras(extras);
        PendingIntent pi = PendingIntent.getActivity(this, id, i,
PendingIntent.FLAG_UPDATE_CURRENT);
        notification.contentIntent = pi;

        // set led
        notification.ledOnMS = 500;
        notification.ledOffMS = 500;
        notification.ledARGB = 0x00ff0000;
        notification.flags |= Notification.FLAG_SHOW_LIGHTS;

        // send it
        NotificationManager mgr = (NotificationManager)
getSystemService(NOTIFICATION_SERVICE);
            mgr.notify(noti_id, notification);
            noti_id ++;
    }

    private boolean isTaintedSend(String msg) {
        // covers "libcore.os.send" and "libcore.os.sendto"
        return msg.contains("libcore.os.send");
    }

    private boolean isTaintedSSLSend(String msg) {
      return msg.contains("SSLOutputStream.write");
    }

    private void processLogEntry(LogEntry le) {
            String timestamp = le.getTimestamp();
            String msg = le.getMessage();
        boolean taintedSend = isTaintedSend(msg);
        boolean taintedSSLSend = isTaintedSSLSend(msg);
        if(taintedSend || taintedSSLSend) {
            String ip = get_ipaddress(msg);
            String taint = get_taint(msg);
            String app = get_processname(le.getPid());
            String data = get_data(msg);
            if (taintedSSLSend)
                  ip=ip+" (SSL)";

            sendTaintDroidNotification(le.hashCode(), ip, taint, app, data,
timestamp);
        }
    }

    @Override
    public IBinder onBind(Intent intent) {
        // we don't bind to this service
        return null;
    }

    @Override
    public void onCreate() {
      File file = new File(this.getFilesDir(), filename);
        return;
    }

    @Override
    public int onStartCommand(Intent intent, int flags, int startId) {
        if(isRunning) {
            return START_NOT_STICKY;
        }
        logQueue = new ArrayBlockingQueue<LogEntry>(LOGQUEUE_MAXSIZE);
        this.captureThread = new Thread(new Producer(logQueue));
        captureThread.setDaemon(true);

        this.readThread = new Thread(new Consumer(logQueue));
        readThread.setDaemon(true);

        try {
            LogcatDevice.getInstance().open();
        }
        catch(IOException e) {
            Log.e(TAG, "Could not open the log device: " + e.getMessage());
            return START_NOT_STICKY;
        }

        this.doCapture = true;
        captureThread.start();

        this.doRead = true;
        readThread.start();

        isRunning = true;

        return START_NOT_STICKY;
    }

    @Override
    public void onDestroy() {
        super.onDestroy();

        // stop the thread
        this.doCapture = false;
        this.doRead = false;

        // close the log
        try {
            LogcatDevice.getInstance().close();
        }
        catch(IOException e) {
            Log.e(TAG, "Could not close the log device properly: "
                    + e.getMessage());
        }

        // destroy the thread
        this.captureThread = null;
        this.readThread = null;
    }
}
```

- LogEntry.java

```
package org.app.analysis;

import android.util.Log;
import java.util.Date;

public class LogEntry {
    private static final String LOGTAG = LogEntry.class.getSimpleName();
    private String timestamp;
    private int pid;
    private String tag;
    private String message;

    private LogEntry() {
    }

    public static LogEntry fromLine(String line) {
        String[] tokens = line.split("\\s+");

        // skip over "--------- beginning of /dev/log/system" etc
        if (tokens[0].equals("---------"))
            return null;

        LogEntry le = new LogEntry();

        le.timestamp = tokens[0]+" "+tokens[1];

        le.tag = 
tokens[2].substring(tokens[2].indexOf("/"),tokens[2].indexOf("("));

        le.pid = 
Integer.valueOf((line.substring(line.indexOf("(")+1,line.indexOf(")"))).tri
m());

        int messageStart = line.indexOf("): ");
        le.message = line.substring(messageStart);

        return le;
    }

    public String getTimestamp() {
        return this.timestamp;
    }

    public int getPid() {
        return this.pid;
    }

    public String getTag() {
        return this.tag;
    }

    public String getMessage() {
        return this.message;
    }
}
```

- LogcatDevice.java

```
package org.app.analysis;

import java.io.*;

import android.util.Log;

public class LogcatDevice {
    private static final String LOGTAG =
LogcatDevice.class.getSimpleName();

    private Process logcatProcess = null;
    private BufferedReader br = null;

    private LogcatDevice()
    {
    }

    private static LogcatDevice instance = new LogcatDevice();

    public static LogcatDevice getInstance() {
        return instance;
    }

    public void open() throws IOException {
        this.logcatProcess = Runtime.getRuntime().exec("logcat -v time *:S
TaintLog:*");
        this.br = new BufferedReader(new
InputStreamReader(logcatProcess.getInputStream()));
    }

    public boolean isOpen() {
        return(this.br != null);
    }

    public LogEntry readLogEntry() throws IOException {
        if(!isOpen()) {
            throw new IOException("must open log first");
        }

        String line = this.br.readLine();
        LogEntry le = LogEntry.fromLine(line);
        return le;
    }

    public void close() throws IOException {
        if(isOpen()) {
            this.logcatProcess.destroy();
            this.br.close();
            this.br = null;
        }
    }
}
```