ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

IJETMR
International Journal of Engineering Technologies and Management Research
A Knowledge Repository

DETECTION OF HIDDEN FUNCTIONALITIES OF SMART PHONE MALWARE APP USING PATTERN-MATCHING TECHNIQUES

Dr. M. Kathirvelu [*1]
[1] Professor, Department of ECE, GMR Institute of Technology, Rajam, Andhra Pradesh-532127, India

Abstract:
Malware developers are progressively using advanced techniques to defeat malware detection tools. One such technique commonly observed in recent malware samples consists of hiding and obfuscating modules containing malicious functionality in places that static analysis tools overlook. In this paper, we describe a dynamic analysis approach for detecting such hidden or obfuscated malware components distributed as parts of an app package. The key idea is behavioral differences between the original app and a number of automatically generated versions of it, where a number of modifications (faults) have been carefully injected. The differential signature is analyzed through a pattern-matching process driven by rules that relate different types of hidden functionalities with patterns found in the signature. A thorough justification and a description of the proposed model are provided.

Keywords: Computer Security; Malware; Mobile Computing.

Cite This Article: Dr. M. Kathirvelu. (2018). "DETECTION OF HIDDEN FUNCTIONALITIES OF SMART PHONE MALWARE APP USING PATTERN-MATCHING TECHNIQUES." *International Journal of Engineering Technologies and Management Research,* 5(11), 107-116. DOI: 10.5281/zenodo.2281616.

1. Introduction

Smart phones present a number of security and privacy concerns that are, in many respects, even more alarming than those existing in traditional computing environments. Most smart phones platforms are equipped with multiple sensors that can determine user location, gestures, moves and other physical activities, high-quality audio and video recording capabilities. Sensitive pieces of information that can be captured by these devices could be easily leaked by malware residing on the smart phone. In this paper we describe a tool for detecting, through reverse engineering, obfuscated functionality in components distributed as parts of an app package. Such components are often part of a malicious app and are hidden outside its main code components. The analyzing the behavioral differences between the original app and an altered version where a number of modifications (*faults*) have been carefully introduced. Such modifications are designed to have no observable effect on the app execution. The signatures that should detect the confirmed malicious threats are still mainly created manually, it is important to discriminate between samples that pose a new unknown threat, and those that are mere variants of known malware [2]. a class of smartphone malware that uses steganographic techniques to hide malicious executable components within their assets, such as

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

documents, databases, or multimedia files [8]. The fundamental deficiency in the pattern-matching approach to malware detection is that it is purely syntactic and ignores the semantics of instructions. In this paper, malware detection algorithm addresses this deficiency by incorporating instruction semantics to detect malicious program traits [4]. A lightweight method for detection of Android malware that enables identifying malicious applications directly on the smartphone [1]. A system that addresses this problem by relying on stochastic models of usage and context events derived from real user traces [9].

It has two differentiated major components: fault injection and differential analysis. The first one takes an entire package of candidate app as input and generates a fault-injected one. This is done by first extracting all components in the app and then identifying those suspicious of containing obfuscated functionality. Such identification is done by comparing specific statistical features of the component's contents with a predefined model for each possible type of resource (i.e., code, pictures and video, text files, databases, etc.). Faults are then injected into candidate components, which are subsequently repackaged, together with the unaltered ones, into a new app. This process admits simultaneous injection of different faults into different components and it is driven by a search algorithm that attempts to identify where the obfuscated functionality is hidden. Both the original and the fault-injected apps are then executed under identical conditions and their behavior is monitored and recorded in the form of form of two behavioral signatures. Such signatures are merely sequential traces of the activities executed by the app, such as for example opening a network connection, sending or receiving data, loading a dynamic component, sending an SMS, interacting with the file system, etc. Many mobile malware prevention techniques are ported from desktop or laptop computers. However, due to the uniqueness of smartphones [10], such as multiple-entrance open system, platform-oriented, central data management, vulnerability to theft and lost, etc., challenges are also encountered when porting existing anti-malware techniques to mobile devices. These challenges include, inefficient security solutions, limitations of signature-based mobile malware detection, lax control of third party app stores, and uneducated or careless users, etc.

An attacker may spoof the "Caller ID" and pretend to be a trusted party. Researchers also demonstrated how to spoof MMS messages that appeared to be messages coming from 611, the number the carriers use to send out alerts or update notifications [11]. Further, base stations could be spoofed too [5]. The differential signature is finally matched against a rule-set where each rule encodes a relationship between the type of presumably hidden functionality and certain patterns in the differential signature.

2. Differential Fault Analysis Model

This section explains the theoretical background used for:

- Inject faults into apps;
- Represent behavioral differences between apps;
- Deduce properties from such behavioral differences considering injected faults and observed differences.

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

2.1. Fault Injection Model

An app P can be seen as a collection of components

$$P = \{c1, c2, \ldots, ck\}. \quad (1)$$

A component can be composed of a number of classes (i.e., code), but also other resources that are dynamically accessed, such as for example asset files. Components have a type, such as for example code, picture, video, database, etc. A type function τ (c) can be defined that returns the type of component c. Faults are then injected into candidate components, which are subsequently repackaged, together with the unaltered ones, into a new app [7]. The propagation of a fault through to an observable failure follows a well defined cycle. When executed, a fault may cause an error, which is an invalid state within a system boundary [7]. Most smartphone Trojans are related to activities such as recording calls, instant messaging, finding a location via GPS, or forwarding call logs and other vital data. According to [6], Smart Message System Trojans comprise a large category of mobile malware that run in an application‘s background and send SMS messages to a premium rate account owned by an attacker.

Fault conditions can be injected into an app by altering one or more of its components. If C is the set of all possible app components, a Fault Injection Operator (FIO) is a transformation.

$$\Psi^{c\boldsymbol{i}} : 2^{C} \rightarrow 2^{C}$$
$$\Psi^{c\boldsymbol{i}}(P) = P \setminus \{c\boldsymbol{i}\} \cup \{\Psi(c\boldsymbol{i})\}. \quad (2)$$

That is, $\Psi^{c\boldsymbol{i}}(P)$ returns a modified version of P *where* component ci has been replaced by $\Psi(c\boldsymbol{i})$. Depending on the functionality of c and on the nature of the modifications introduced by Ψ, replacing c by $\Psi(c)$ may (or may not) translate into observable differences in the execution of P.

In this paper, restrict to FIOs that make alterations to data components only, not to instructions. Data components include the value of variables found in the code and also asset files such as databases, pictures, audio and video files.

2.2. Modeling Differential Behavior

A key task in our system is the analysis of the behavioral differences between an original app and a slightly modified version of it after applying a FIO. Model to represent traces of activities and differences between such traces are given below.

2.3. Behavioral Signatures

An app interacts with the platform where it is executed by requesting services through a number of system calls. These define an interface for apps that need to read/write files, send/receive data through the network, place a phone call, etc. In some cases, there will be a one-to-one

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

correspondence between a behavioral activity and a system call, while in others a behavioral activity will encompass a sequence of system calls executed in a given order. It is assumed that

$$A = \{a1, a2, \ldots, an\} \quad (3)$$

is a set of all relevant and observable activities an app can execute. The execution flow of an app P may follow different paths depending on its inputs.

2.4. Differential Signatures

We are interested in analyzing the differences between two observed behaviors given by their respective behavioral signatures. We approach this problem as one of string-to- string correction, where differences are represented as the minimum number of edits operations needed to transform one signature into the other. Given a behavioral signature $\sigma = (s1, s2, \ldots, sn)$, we define the next three families of signature. Smartphone also feature high-quality audio and video recording capabilities. Sensitive pieces of information that can be captured by these devices could be easily leaked by malware residing on the Smartphone. Even apparently harmless capabilities have swiftly turned into a potential menace. For example [8], access to the accelerometer or the gyroscope can be used to infer the location of screen taps and, therefore, to guess what the user is typing (e.g., passwords or message contents). Most smartphone Trojans are related to activities such as recording calls, instant messaging, finding a location via GPS, or forwarding call logs and other vital data. According to [6], Smart Message System Trojans comprise a large category of mobile malware that run in an application‘s background and send SMS messages to a premium rate account owned by an attacker.

3. Differential Fault Analysis of Obfuscated Apps

There are two differentiated major blocks:
The first one generates a number of fault-injected apps. This process is carried out by first extracting all app components and identifying those of interest (CoIS3), i.e., those components suspicious of containing hidden functionality. An iterative process then selects candidate CoIs and injects faults into them. Both modified and unmodified components are then repackaged together into a new app.

The second block generates stimuli (user inputs and context) for both apps and executes them, generating a pair of behavioral signatures. The differential signature is then computed and matched against a database of patterns to identify the presence of hidden functionality.

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

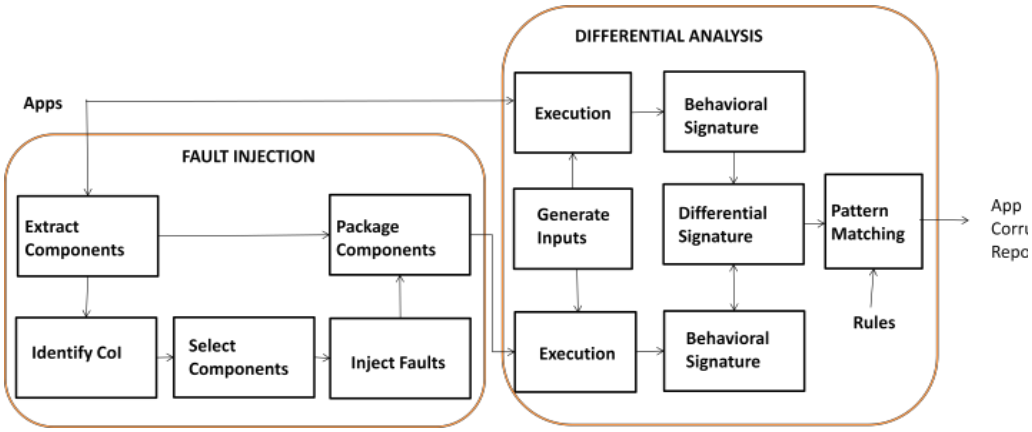

Figure 1: Architecture of Differential Fault Analysis.

3.1. Identifying Components of Interest

The first step in the analysis of an app is identifying components of interest (CoIs), i.e., parts of an app suspicious of containing hidden functionality. Such components will be later fault injected according to some strategy in order to analyze the resulting behavior. We say that a component c of type τ (c) in an app P is of interest if it does not fit a model M_τ (c) defined for all components of type τ (c). In the current version measures statistical features only, such as for example the expected entropy, the byte distribution, or the average size. Such features are computed from a dataset of components of the same type, such as text files, pictures, code, etc. For each model M, we assume a Boolean function test (c, M) that returns true if c complies with M, and false otherwise. For example, if M is a byte distribution, then test (c, M) could be a goodness of fit test (e.g., χ^2) between M and c's byte distribution. More formally

$$c \in \text{CoIS}(P) \Leftrightarrow \text{test}\ (c, M_\tau\ (c)) = \text{false}. \quad (4)$$

3.2. Generating Fault-injected Apps

Components of interests identified in the previous stage are injected with faults and reassembled, together with the remaining app components, to generate a faulty app P0. This process can generate several fault-injected apps, as there are multiple ways of applying different FIOs to different components in the set of CoIs. Fault-injected apps are generated one at a time and sent for differential analysis. If no evidence of malicious behavior is found in the differential analysis, the fault injection process is invoked again to generate a different faulty app, and so on.

Algorithm for obtaining CoIs from an app
Input:
App: $P = \{c1, c2, \ldots, ck\}$
Set of type normality models: $\{M1, M2, \cdots, Mn\}$
Set of FIOs: $\{\Psi1, \Psi2, \cdots, \Psi m\}$ Mode: normal/ exhaustive **Procedure:**
CoIS $\leftarrow \emptyset$
For each c $\in P$ **do**
if [test (c, M_τ (c)) = false] or

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

[(mode = exhaustive) and ($\exists\ \Psi i : \tau(\Psi i) = \tau(c)$)] **then**
CoIS ← CoIS $\cup\{c\}$
return CoIS

3.3. Applying Differential Analysis

Differential analysis between a candidate fault-injected app and the original app is carried out. The process comprises the following steps:
Generate an appropriate usage pattern u and context to feed both apps and extract their behavioral signatures, σ[P(u|t)] and σ[P0(u|t)]. Both the original and the fault-injected app are tested under the same conditions and using the same inputs. Note that this assumes that the execution of an app is completely deterministic.

Generate the differential signature Δ(σ[P(u|t)], σ[P0(u|t)]) from the behavioral signatures obtained above. Apply sequentially all rules Ri over
Δ(σ[P(u|t)], σ[P0(u|t)]) and return those for which a match is obtained.

4. Prototype Implementation

Prototype is implemented using Java and Python components and relies on a number of Android open source tools for specific tasks. App components are extracted using Androguard. After fault injection, components are repackaged into a modified app using Apk Tool. Monkey is used to generate a common sequence of events to interact with both the original app and the fault-injected app. These events should be generated specifically for each test to intelligently drive the GUI exploration i.e., to test code implementing different functionalities of the app. In its current version, uses Monkey to generate 5 classes of input events: activity launch, service launch, action buttons, screen touch, and text input.

Each app is then executed in a controlled environment using the stream of events generated above. For this purpose, we use Droidbox, a sandbox that allows monitoring various features related to the execution during a fixed, user-given amount of time. In order to generate behavioral signatures, it monitors the execution of 11 different activities:

- crypto: generated when calls to the cryptographic API are invoked
- net-open, net-read, net-write: associated with network I/O actitivites (opening a connection, receiving, and sending data)
- file-open, file-read, file-write: associated with file system I/O activities (opening, reading, and writing)
- sms, call: generated whenever a text message or a phone call is sent or received

4.1. COI Models

EXEFile Match: This model analyzes components of type Dalvik Executable Format (DEXFileMatch), Application Package file format (APKFileMatch), and Executable and Linkable Format (ELFFileMatch), i.e.,
τ(c) = hDEX,APK,ELFi.The model defined for these components is based on the magic number defined in the file header.

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

ImgFile Match: This model analyzes components of type picture, such as PNG, JPG, or GIF images, i.e., τ(c) = hPNG,⋯ ,JPGi. This model is based on the magic number defined in the file header

Encrypted or Compressed Match: This model matches any file whose entropy, measured at the byte level, exceeds a given threshold. In such a case, the file is considered to contain random or encrypted information and, therefore, is selected for fault analysis. We set the current threshold to 3.9. Such value was chosen after measuring the entropy of several files before and after being encrypted with DES.

Extension Mismatch: This model identifies files such that their magic numbers do not match the file extension. For instance, we found several APK files with DB extension and several encrypted files with JPG extension. We currently support two sub models: ImgFile Extension Mismatch and APKFile Extension Mismatch.

Text Script Match: This model analyzes components that match any ASCII text executable file, i.e., τ(c) = Script. This model is also based on the magic number defined in the file header. All CoIs described above are implemented in Python. The set can be easily extended to incorporate additional models by simply adding the corresponding module.

4.2. Differential Rules

The basic set of differential rules incorporated is comprised of 9 rules shown in Table 1. It will apply to indistinguishable FIOs and cover the most common examples of obfuscated functionality: network activity, file activity, data leakage, SMS activity, hidden payloads, update attacks, cryptographic activity, cryptographic payloads, and generic hidden functionality. To reduce the complexity of the search space, all basic rules apply to indistinguishable FIOs. However, for the sake of completeness our implementation incorporates several distinguishable FIOs, and new rules can be further added to match them. For instance, given an app that incorporates a DEX program used to enhance photos taken from the camera, we can use a rule to check whether this CoI actually does just that or not. Thus, if after applying a FIO over this component the differential signature shows, for instance, changes in network activity, we may suspect that the CoI contained other functionality piggybacked on the DEX.

Table 1: Basic indistinguishable differential rules implemented

Name	Contains	Rule
RNBC	Network Behavior Component	$\exists i1$: contains $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, *Del-net-open* $i1$ $\forall \exists i2$: contains $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Del$^{net-read}$ $i1$
RFBC	File Behavior Component	$\exists i1$: cont.$\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Del file – open $i1$ $\forall \exists i2$: contains $\Delta(\sigma[P], \sigma[\Psi^C(P)])$,Delfile – read $i1$ $\forall \exists i3$: contains $\Delta(\sigma[P], \sigma[\Psi^C(P)])$,Delfile - write $i2$
RDLC	Data Leak	$\exists i$: contains$\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Delleak I
RSBC	SMS Behavior	$\exists i$: contains$\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Delsms i
RPBC	Payload	$\exists i$: contains$\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Deldexload I

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

RUPC	Update Payload	$\exists i1$: contains˙ $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Delnet-read $i1$ $\wedge \exists i$: contains˙ $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Deldexload i
RCBC	Crypto	$\exists i$: contains˙ $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Delcrypto˙$I$
RCPC	Crypto Payload	$\exists i1$: contains˙ $\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Delcrypto˙ $i1$ $\wedge \exists i$: contains˙$\Delta(\sigma[P], \sigma[\Psi^C(P)])$, Deldexload i
RHFC	Hidden Function	¬equal˙ $\Delta(\sigma[P],\sigma[\Psi^C(P)])$,$\emptyset$

5. Results

An app which performs the activities of a malicious application is created and its operation is shown in figure 2.

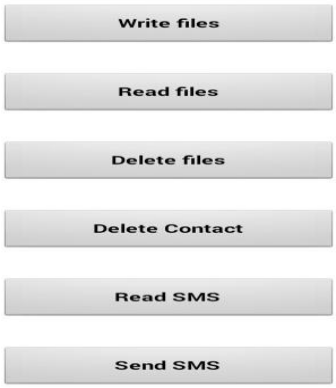

Figure 2: List of operations performed by Malware App

Create a file in our smart phone without our knowledge.
Read all the files in our phone and also can delete the ones they require.
Read all the contacts in our phone and can delete them automatically,
Read all the messages in our phone.
Send messages to others.

These are the basic operations which will be performed daily in our day to day life. But these activities will be done by the malicious app in the background without our knowledge. Now, this becomes a great threat to the security to our data since we might have some confidential information in our phone such as passwords, bank account numbers, ATM pin numbers etc.

5.1. Security App

All those malicious apps can be detected by this security app. And this app, will also give the details about what are all the operation that are done by those malicious apps.

The prototype allows to perform the analysis in parallel. We presently limit our implementation to a small number of CoI models, FIO operators, and differential matching operators. This architecture allows security experts to further extend this and configure their own operators based on their experience

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

Figure 3: Screen shot of operation done by Malicious App

6. Conclusions

In this paper we have presented a framework for malware analysis based on the notion of differential fault analysis. The architecture is described and provided a formal model of differential fault analysis. Additionally, we have an open source prototype implementation with a versatile design that can be the basis for further research in this area. Differential fault analysis in the way implemented is a powerful and novel dynamic analysis technique that can identify potentially malicious components hidden within an app package. Additionally, empowering dynamic analysis with a fault injection approach can be used to differentiate "gray" from legitimate behavior when analyzing gray ware. This is a good complement to static analysis tools, more focused on inspecting code components but possibly missing pieces of code hidden in data objects or just obfuscated.

References

[1] Arp.D, Spreitzenbarth.M, Ubner.M.H, Gascon.H, and Rieck.K, "Drebin: Effective and explainable detection of android malware in your pocket," in Proc. NDSS, February 2014.
[2] Egele.M, Scholte.T, Kirda.E, and Kruegel.C, "A survey on automated dynamic malware-analysis techniques and tools," ACM Comput. Surv., vol. 44, no. 2, pp. 6:1–6:42, Mar. 2012.
[3] Cai.L and Chen.H, "Touchlogger: inferring keystrokes on touch screen from smartphone motion," in Proc. USENIX, ser. HotSec'11, Berkeley, CA, USA, 2011, pp. 9–9.
[4] Christodorescu.M, Jha.S, Seshia.S, Song.D, and Bryant.R, "Semantics-aware malware detection," in Security and Privacy, 2005 IEEE Symposium on, May 2005, pp. 32–46.
[5] D. Perez and J. Pico, "A practical attack against GPRS/EDGE/UMTS/HSPA mobile data communications," Black Hat DC, 2011.
[6] G. Suarez-Tangil, J. E. Tapiador, P. Peris, and A. Ribagorda, ―Evolution, detection and analysis of malware for smart devices‖, IEEE Comms. Surveys & Tut., vol. 16, no. 2, pp. 961–987, May 2014.

ISSN: 2454-1907
DOI: 10.5281/zenodo.2281616

[7] L. K. Yan and H. Yin, ―Droidscope: seamlessly reconstructing the os and Dalvik semantic views for dynamic Android malware analysis, ‖ in Proc. USENIX, ser. Security '12. Berkeley, CA, USA: USENIX Association, 2012, pp. 29–29.
[8] Shabtai, L.Tenenboim-Chekina, D.Mimran, L.Rokach, B.Shapira, Y.Elovici. Mobile malware detection through analysis of deviations in application network behavior‖., Department of Information Systems Engineering,2014.
[9] Suarez-Tangil.G, Conti.M, Tapiador.J.E and PerisLopez.P, "Detecting targeted smartphone malware with behavior-triggering stochastic models," in ESORICS 2014, ser. LNCS, vol. 8712. Springer International Publishing, 2014, pp. 183–201.
[10] Y. Wang, K. Streff, and S. Raman, "Smartphone Security Challenges," Computer (Long. Beach. Calif)., vol. 45, no. 12, pp. 52–58, Dec. 2012.
[11] Z. Lackey and L. Miras, "Attacking SMS," BlackHat 2009, 2009.

*Corresponding author.
E-mail address: mkathirvelu77@ gmail.com