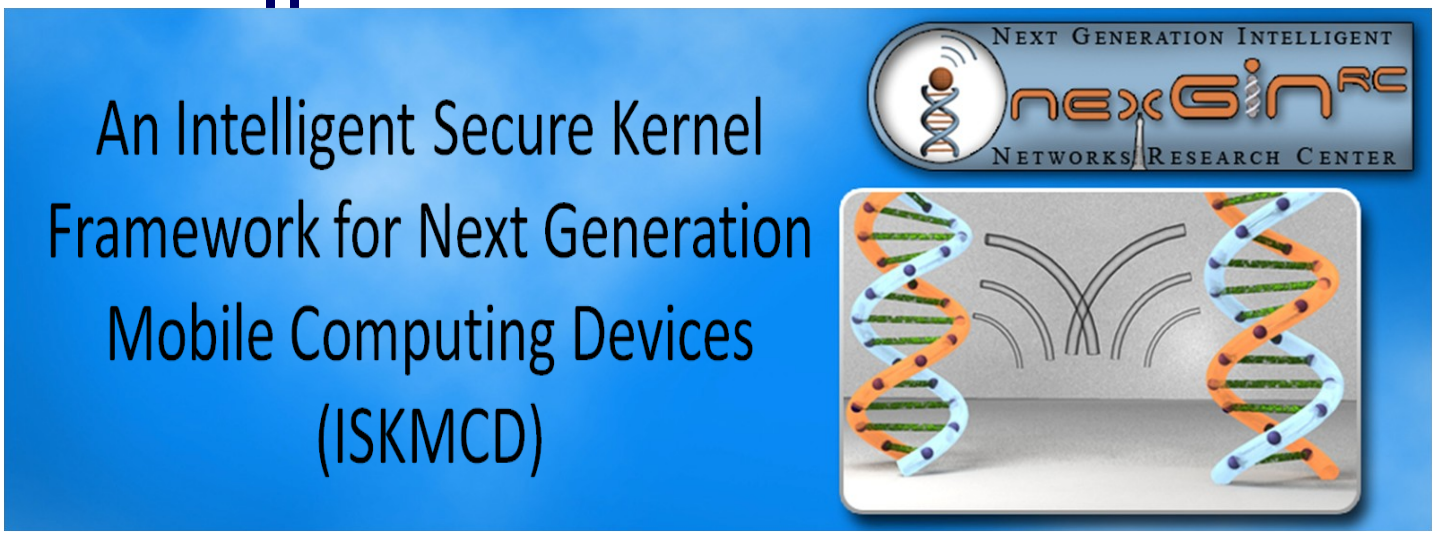


Technical Report

An Intelligent Secure Kernel Framework for Next Generation Mobile Computing Devices

(ISKMCD)

"TStructDroid: Realtime Malware Detection using In-execution Dynamic Analysis of Kernel PCB on Android"

Jan 28, 2013

National University of Computer & Emerging Sciences, Islamabad, Pakistan

TStructDroid: Realtime Malware Detection using In-execution Dynamic Analysis of Kernel Process Control Blocks on Android

Farrukh Shahzad, M. A. Akbar, Salman H. Khan and Muddassar Farooq

Abstract—As the smartphone devices have become a basic necessity and their use has become ubiquitous in recent years, the malware attacks on smartphone platforms have escalated sharply. As an increasing number of smartphone users tend to use their devices for storing privacy-sensitive information and performing financial transactions, there is a dire need to protect the smartphone users from the rising threat of malware attacks. In this paper, we present a realtime malware detection framework for Android platform that performs dynamic analysis of smartphone applications and detects the malicious activities through in-execution monitoring of process control blocks in Android kernel. Using a novel scheme based on information theoretic analysis, time-series feature logging, segmentation and frequency component analysis of data, and machine learning classifier, this framework is able to mine the hidden patterns in the execution behavior of applications and model it to detect real world malware applications for Android with very low false alarms. We have used a realworld dataset of 110 benign and 110 malware Android applications. The experiments show that our framework is able to detect the zero-day malware with over 98% accuracy and less than 1% false alarm rates. Moreover, the system performance degradation caused by or framework is only 3.73% on average for a low-end Android smartphone, making it ideal for deployment on resource constrained mobile devices.

Index Terms—Malware Detection, Dynamic Analysis, In-execution Malware detection, Android Security, Smartphone Security

1 Introduction

Mobile devices have shifted from being an item of luxury to an item of necessity over the past couple of decades. The swift technological improvements in developing low cost, high power processors and a ubiquitous access to the Internet has led to the popularity of smartphones. It has been reported that 419.1 million mobile devices were in the market by the beginning of year 2012 and are estimated to become 645 million by the end of year 2012[1] [1].

With an expected 10 billion mobile devices in the market by 2016 [2], the high popularity of smartphone devices has resulted in the development and use of a plethora of smartphone applications providing diverse functions such as communication, entertainment, productivity and financial transactions. A study of user behavior on smartphones, carried out by Kaspersky Lab [3], shows that users tend to use their smartphones for (1) storing personal information and documents (16%), (2) Internet surfing (62%), (3) email communication (53%), and (4) social networking (47%). There are several major companies providing smartphone platforms (operating systems) for the smartphone devices. Market share analysis for smartphones in second quarter of 2012 shows that Google's Android based smartphones form the majority of the overall smartphone market with an overall share of 64.1%, whereas the closest competitor is the Apple's iOS based smartphone (18.8% share) [4].

This high popularity of smartphone devices has turned the smartphone platforms in to an attractive target for malicious forces [5]. Due to its popularity and large market share, Google's Android platform is facing the major brunt of malware attack. A sixfold increase in malware targeting Android platform has been reported in just one quarter (Jul 2012 - Sep 2012), resulting in almost $175{,}000$ known malware on Android in Sep 2012 [6]. The majority of these malware belong to Trojans and Adware categories [7].

Apart from the popularity of Android platform resulting in increasing malware threats, there are other factors that contribute in increasing the threat level to this platform. Android platform supports an open-architecture for application distribution. This means that anyone can obtain an Android application from any source and install it on to the phone. Pirated versions of commercial Android applications are commonly available at shady online application stores. The official application store for Android, known as *Google Play Store*[2], enforces no strict application review process. This policy of open architecture is different from some other leading smartphone platforms such as iOS and Symbian which require strict application review, signing and run-time signature

• *Farrukh Shahzad, M. A. Akbar, Salman Khan and Muddassar Farooq are with the Next Generation Intelligent Networks Research Center, FAST-National University of Computer and Emerging Sciences, Islamabad, 44000, Pakistan.*
E-mail: {farrukh.shahzad, ali.akbar, salman.khan, muddassar.farooq}@nexginrc.org

Manuscript received November 15, 2012; revised February 15, 2013.

1. Among these mobile devices, 144.3 million are smartphones [1].

2. https://play.google.com/

and integrity verification of smartphone applications. Lack of such mechanisms results in an open playfield for malware to attack the platform and take advantage of the vulnerabilities.

The major off-the-shelf mobile anti-malware products are introduced by companies such as McAfee, Kaspersky, Avast, Norton, and Lookout [8] etc. which are typically signature-based. On the other hand, static analysis based techniques detect malicious patterns from source code and executable binaries. Both, signature and static analysis based techniques are prone to evasion using common code obfuscation techniques and polymorphism. Moreover, zero-day (previously unknown) malware pass undetected by these products [9] [10]. As mobile devices are more resource-constrained than their desktop counterparts, a viable malware detection system should provide protection against malware attacks without putting great strain on computational power, memory resources and battery utilization [11]. Moreover, zero-day malware detection using efficient dynamic analysis, high detection accuracy, low false alarms and robustness to evasion are some of the key factors that must be taken into account while designing a solution for this problem.

To solve this emerging problem, we propose `TStructDroid`: a real-time malware detection framework that uses in-execution dynamic analysis of kernel Process Control Blocks on Android platform[3]. Using information theoretic analysis, time-series feature logging, segmentation and frequency component analysis of data, and machine learning classifier, this framework is able to detect real world malware applications for Android while producing very low false alarms. We have used a realworld dataset of 110 benign and 110 malware Android applications. The experiments show that our framework is able to detect the zero-day malware with over 98% accuracy and less than 1% false alarm rates. Moreover, the system performance degradation caused by or framework is only 3.73% on average for a low-end Android smartphone, making it ideal for deployment on resource constrained mobile devices.

The major contributions of our work are as follows:

- We present a novel scheme based on information theoretic analysis, time-series feature logging, segmentation and frequency component analysis of data from to mine the hidden patterns in the execution behavior of applications.
- Based on this scheme, we propose `TStructDroid`: a real-time malware detection framework that uses in-execution dynamic analysis of kernel Process Control Blocks on Android platform.
- We have used a relatively large realworld dataset of 110 benign and 110 malware Android applications to test our proposed framework.
- We design a series of experiments with both real-world and cross-validation scenarios to test the classification performance of the system. Moreover, the system performance degradation of the framework is also measured.

The rest of the paper is organized as follows. We begin with the description of methodology used to collect our benign and malaware datasets in Section 2. We also describe the formation of training and testing datasets for our carefully designed experiment scenarios. Afterwards, we present our novel scheme for extraction of hidden patterns in execution behavior of malicious processes by utilizing the kernel structures. The components of the scheme are accompanied by discussion and theoretical & empirical validation of the reasoning behind each step. In Section 4, the major components of our proposed *TStructDroid* framework are presented and the working of each component is discussed in detail. We present the performance evaluation of the framework in Section 5. We briefly describe the related work (Section 6) in the field of dynamic malware analysis on smartphone platforms and identify the shortcomings of recently proposed frameworks. Finally, we conclude with an outlook towards future work.

2 ANDROID - MALWARE & BENIGN DATASETS

The most important part of designing and evaluating a malware detection framework is the selection of the dataset. In this section, we discuss the approach towards selection of the benign and malware datasets and the brief characteristics of each of them. The malware detection framework extracts features derived from the process control blocks of the processes in the Android kernel at runtime[4]. 110 malware and 110 benign applications have been used in this study. The list of the malware and benign Android applications used in the datasets is given in Table 4.

2.1 Benign Dataset

We have selected 110 benign applications for our experiments. These applications have been chosen from the featured applications on the Google Play Store for Android applications[5]. We chose the top applications from different categories (Games, Image Viewers, Sketching tools, Text Editors, Image Editors, Android Utilities such as Recorder, Dialer, Maps etc., and Misc. applications) for the month of August 2012. The criteria for choosing these applications were: (1) the applications should be famous/top downloaded so that they are representative of commonly used user applications, (2) the applications should belong to different categories for a diversity in the behavior of the applications, and (3) a

3. The feasibility of using process control blocks for malware detection on Linux and Android have been previously established in literature [12][13].

4. The datasets containing the logged process control block features of both benign and malware applications collected for this study are available on the website http://www.nexginrc.org.

5. https://play.google.com/store. Last accessed on Dec 1, 2012.

balance should be maintained of user-interactive (e.g. editor) and automated/background-service (e.g. download manager) applications.

2.2 Malware Dataset

Malware for Android has expanded significantly in numbers during recent years. However, most of the anti-malware companies are reluctant in sharing the malware samples collected over a long period. The *Contagio Mobile Malware Mini Dump*[6] is a publicly available collection of Android malware. We have collected 110 malware applications from *Contagio Mobile Malware Mini Dump* (collected upto August 2012). The Android applications in the collected malware dataset belongs to the following malware categories: Trojans, Adware, Rootkits, Bots and Backdoors. Trojans make up for the majority of Android malware in our dataset. Following in the old Greek tradition of trojan horses, *Trojans* usually pose as legitimate and productive applications. Their is a hidden trigger that initiates the malicious activity in the background while the legitimate application executes in the front. Trojans typically infect other legitimate applications on the device to convert them to trojans as well. Some prominent trojan families[7] included in our dataset are: N.Zimto, LuckyCat, Qicsomo, FakeTimer, SMSZombie, Loozfon, ZFT, Instagram, FakeToken, Ginimi, VDLoaded, CounterClank, Gamex, DroidDreamLight, FakeInstaller and Arspam Alsalah [14]. This category of malware usually consists of applications that try to monitor a user's behavior (such as surfing patter, interests etc.) or try to steal important user data (such as SMS messages, photos, location information, banking information etc). Some well-known spyware families in our dataset include Plankton, DougaLeaker, Gonein60x, Steeks and FindAndCall etc. [14]. *Rootkits* infect a system's kernel and thus get full access to the system. The typical use of rootkit exploits is to hide activity of other malware installed on the system. Moreover, the rootkits are not run in limited permissions environment like other types of malware and thus are able to monitor and modify other applications and sensitive resources on the smartphone device. Our dataset contains some well-known Android rootkits such as Z4Root:three, and ITFUNZ.supertools [14]. Bots are malware applications that receive commands from a server and perform operations according to those instructions. They are typically a part of a large network of bots (known as botnet), and are used to launch distributed denial of service or spam attacks. The CI4.updater [14] bot is included in our dataset. Once a device has been infected through a particular vulnerability, a *backdoor* allows an attacker to create a hidden communication channel with the infected device. Even if the exploited vulnerability is patched later, the attacker can still access the device through the backdoor. DroidKungfu [14] is a very well-known example of backdoors in our dataset.

It is important to understand that these categories of malware are loosely defined and a malware application may belong to a number of such categories at the same time. Another important factor which differentiates our malware dataset from those used in literature is the very high percentage of Trojanized/Spyware Android applications. Such smartphone malicious applications are interactive and typically appear to execute more like benign ones. The malicious code is typically run in parallel to typical application execution scenario such as a Trojan code patch in a popular game. This added challenge of detecting hidden pattern of malicious behavior is useful for testing the efficiency of our presented framework. Usage of raw execution information from the kernel is insufficient for detection of this tricky behavior. To meet this challenge, we present a novel scheme that analyzes and extracts hidden patterns in execution behavior utilizing information in kernel structures.

2.3 Creation of Training & Testing Datasets

After describing the composition of the benign and malware applications in our datasets, we now describe the two different methodologies that we have used for creating training and testing datasets using these benign and malware Android applications. We create two different scenarios: the first one is based on malware detection in real life, and the second one is based on the standard methodology typically used in machine learning literature.

2.3.1 Realtime Scenario:

In real life scenario, malware detection frameworks can typically detect the known malware using their signatures. The real significance of the dynamic malware detection frameworks lies in the detection of zero-day (previously unknown) malware.

To create a realtime, real-life scenario, we use only one application in the testing dataset. The rest of the benign and malware applications are included in the training dataset. In this way, we have 220 different training and testing set combinations using the 220 applications (110 benign and 110 malware) in our datasets.

2.3.2 Standard - Cross Validation Scenario:

The *10-fold cross validation* is the standard methodology proposed in machine learning literature for classification of datasets. We create ten training and testing dataset combinations (folds) such that in each fold, the testing dataset consists of randomly chosen 10% of all benign and 10% of all malware applications while the remaining applications are used in the corresponding training dataset.

6. http://contagiominidump.blogspot.com/. Last accessed - Dec 1, 2012.

7. For interested readers, we provide a complete list of individual malicious applications of all malware families present in our dataset [14].

Although the *real-time scenario* is more realistic, it can result in an over-fitting of the classification model during training skewing the results. Therefore, we also create scenarios based on the cross-validation strategy to verify that the high detection rate of our framework is statistically significant and resistant to the changes in the training datasets.

3 A Novel Scheme for Extraction of Hidden Patterns in Execution Behavior Utilizing Kernel Structures

To dynamically classify a process as benign or malicious, the underlying tenet for all dynamic malware detection frameworks is that **the execution pattern/behavior of benign and malicious processes differs significantly** [15][12]. To mine this difference in execution pattern, we monitor the change in frequency components of the extracted feature values from the process control block. In this section, we present our scheme for extraction of these hidden patterns. This scheme makes up the core of our malware detection framework. We also present and prove some important mathematical constructs and properties for this scheme, and attempt to validate the classification potential of the extracted information.

We have selected 99 preliminary parameters from the process control block (`task_struct` in Android kernel). Some of the interesting parameters include number of page frames, volunteer and in-volunteer context switches, number of page faults, virtual memory used, CPU time, number of page tables, file system resources, resource counters of signal structure etc. During the execution of a process, we periodically log the values of these parameters. The time interval for logging (we call it time-resolution δt) has been set as 10 milliseconds[8]. We process the logged preliminary features in batches (or windows).

Overall, our scheme for extraction of hidden patterns in execution behavior of processes from the logged preliminary features values consists of the following steps: (1) analyzing the extracted features, (2) shortlisting the features, (3) removing redundant instances, (4) creating windows (batches of instances), (5) performing time-series transforms, and (6) calculating statistical features from the transformed feature set. We now discuss these different operations one by one. We are using mathematical notations and formulations to make the discussion consistent.

3.1 Time-series Features Shortlisting

The first step is to remove all the redundant features that do not contribute significantly towards detection of malware and may mislead the classification process.

8. Later, we present empirical experiments that helped us choose this value.

Let $X_{(n,i)}$ be the random variable that identifies time-series values of a single feature f_i extracted from process control blocks of Android's applications process P_n.

$$X_{n,i} = (x_{n,i,1}, x_{n,i,2}, \ldots\ldots x_{n,i,T}) \, \forall x_{n,i,t}, 1 \leq t \leq T$$

so,

$$f_i = X_{n,i}, \ 1 \leq n \leq N_p$$

The logged preliminary features can be represented by the set $\mathcal{F}$:

$$\mathcal{F} = \forall f_i \mid 1 \leq i \leq N_f$$

Some of the preliminary features are 'indexers': they are used as identifiers or indexing purposes. Although they might be unique/distinct, they are assigned by the operating system regardless of the behavior of a process. Therefore, as a first step, we identify and remove such features $\mathcal{F}_{indexers}$ from our features list $\mathcal{F}$.

$$\mathcal{F}_{indexers} = \forall f_i \in \mathcal{F}, f_i \text{ is an indexing feature}$$

Afterwards, we use time-series difference, mean and variance of the features as statistical measures to distinguish between features with and without distinct values.

Defintion 1. ***Time-series Difference of a feature.*** *We define difference of a feature $\mathcal{D}_{f_i}(t)$ as the absolute difference between the consecutive time-series values of the random variable corresponding to that feature across all processes in a given time window.*

$$\mathcal{D}_{f_i}(t) = \sum_{n=1}^{N_p} (x_{n,i,t} - x_{n,i,t-1})$$

Defintion 2. ***Time-series Mean of a feature.*** *We define mean of a feature $\mathcal{M}_{f_i}(t)$ as the average of the random variable values corresponding to that feature over all processes in a given time window.*

$$\mathcal{M}_{f_i}(t) = \frac{1}{N_p} \sum_{n=1}^{N_p} x_{n,i,t}$$

Defintion 3. ***Time-series Variance of a feature.*** *We define variance of a feature $\mathcal{V}_{f_i}(t)$ as the mean of the squared deviation of the random variable values from the expected value, corresponding to that feature over all processes in a given time window.*

$$\mathcal{V}_{f_i}(t) = \sigma^2_{f_i}(t) = \frac{1}{N_p} \sum_{n=1}^{N_p} (x_{n,i,t} - \mu_{n,i,t})^2$$

We now define certain sets of features that need to be excluded from the preliminary features list. We consider ϵ as a very small and insignificant value.

The set of constant feature values $\mathcal{F}_{constant}$ is a set of features for which the time-series difference is insignificant for all processes.

$$\forall f_i \in \mathcal{F}, \qquad f_i \in \mathcal{F}_{constant} \iff |D_{f_i}(t)| < \epsilon$$

The set of null feature values $\mathcal{F}_{null}$ is a set of features for which the time-series mean is nearly equal to zero for all processes.

$$\forall f_i \in \mathcal{F}, \qquad f_i \in \mathcal{F}_{null} \iff |M_{f_i}(t)| < \epsilon$$

Mean and variance of the feature values determine the distribution of the feature. Distributions having identical mean and variance across different types of processes do not help in classification of those types. Therefore, the features $\mathcal{F}_{ident-dist}$ that have identical time-series mean and variance for both benign and malicious processes are a good candidate for removal.

$$\forall f_i \in \mathcal{F}, f_i \in \mathcal{F}_{ident-dist}$$

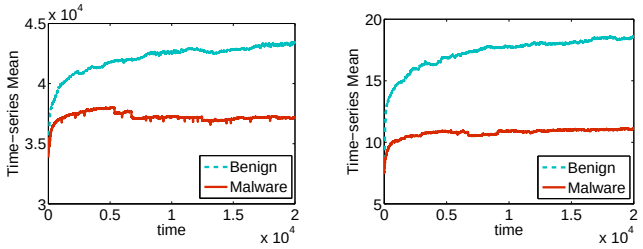
if and only if,

$$|M_{f_i}(t)_{benign} - M_{f_i}(t)_{malicious}| < \epsilon$$

and

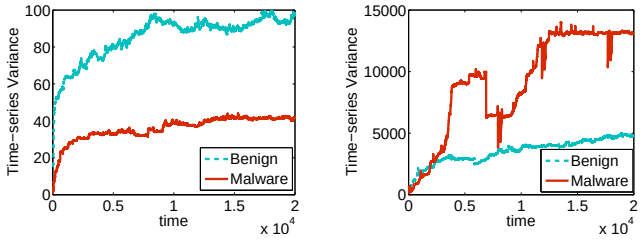
$$|V_{f_i}(t)_{benign} - V_{f_i}(t)_{malicious}| < \epsilon$$

Figures 1 and 2 show plots of time-series mean and variance of some features that have different time-series distribution and are, therefore, part of the shortlisted set of features.



(a) task→active_mm→hiwater_rss (b) task→fs_excl.counter

Fig. 1. Time-series Mean of Some Preliminary Features for Benign and Malicious Processes



(a) task→active_mm→mm_users (b) task→active_mm→map_count

Fig. 2. Time-series Variance of Some Preliminary Features for Benign and Malicious Processes

The features sets that we have identified in this subsection are excluded from the final shortlisted set of features. If p_{f_i} is the probability that a feature f_i is used for classification, then:

$$p_{f_i} = \begin{cases} 0 & \text{if } f_i \in \mathcal{F}_{indexers} \\ 0 & \text{if } f_i \in \mathcal{F}_{constant} \\ 0 & \text{if } f_i \in \mathcal{F}_{null} \\ 0 & \text{if } f_i \in \mathcal{F}_{ident-dist} \\ 1 & \text{Otherwise} \end{cases}$$

After exclusion of features without classification potential, the shortlisted features set $\mathcal{F}_{sel}$ (listed in Table 1) is given by:

$$\forall f_i \in \mathcal{F}, \qquad f_i \in \mathcal{F}_{sel} \iff p_{f_i} > 0$$

TABLE 1
Short-listed task_struct parameters ($\mathcal{F}_{sel}$) of TstructDroid framework for classification purpose

No.	Parameter(s)	Description
1	$task{\rightarrow}state$	The current processing state of process
2	$task{\rightarrow}usage.counter$	Task structure usage counter
3	$task{\rightarrow}prio$	It holds dynamic priority of a process
4	$task{\rightarrow}static\_prio$	Static priority or nice value of a process
5	$task{\rightarrow}normal\_prio$	It holds expected priority of a process
6	$task{\rightarrow}policy$	Scheduling policy of the process
7	$task{\rightarrow}active\_mm{\rightarrow}mmap{\rightarrow}vm\_pgoff$	The offset in vm_file in page-size units
8	$task{\rightarrow}active\_mm{\rightarrow}mmap{\rightarrow}vm\_truncate\_count$	Truncation count or restart address
9	$task{\rightarrow}active\_mm{\rightarrow}task\_size$	The size of task virtual memory space
10	$task{\rightarrow}active\_mm{\rightarrow}cached\_hole\_size$	If non-zero, the largest hole below free-area-cache
11	$task{\rightarrow}active\_mm{\rightarrow}free\_area\_cache$	First hole of size cached-hole-size or larger
12	$task{\rightarrow}active\_mm{\rightarrow}mm\_users$	Number of processes using this address space
13	$task{\rightarrow}active\_mm{\rightarrow}map\_count$	Number of memory regions of a process
14	$task{\rightarrow}active\_mm{\rightarrow}hiwater\_rss$	Maximum number of page frames owned by the process
15	$task{\rightarrow}active\_mm{\rightarrow}total\_vm$	Address space size of process (in terms of number of pages)
16	$task{\rightarrow}active\_mm{\rightarrow}shared\_vm$	Number of pages in shared file memory mapping of process
17	$task{\rightarrow}active\_mm{\rightarrow}exec\_vm$	Number of pages in executable memory mapping of process
18	$task{\rightarrow}active\_mm \rightarrow reserved\_vm$	Reserved virtual memory for a process
19	$task{\rightarrow}active\_mm{\rightarrow}nr\_ptes$	Number of page table entries of a process
20	$task{\rightarrow}active\_mm{\rightarrow}end\_data$	Final address of data section (indicates the length of data section)
21	$task{\rightarrow}active\_mm{\rightarrow}last\_interval$	Last fault stamp interval seen by this process
22	$task{\rightarrow}nvcsw$	Number of voluntary context switches of a process
23	$task{\rightarrow}nivcsw$	Number of involuntary context switches of a process
24	$task{\rightarrow}min\_flt$	Minor page faults occurred for a process
25	$task{\rightarrow}maj\_flt$	Major page faults occurred for a process
26	$task{\rightarrow}fs\_excl.counter$	It holds file system exclusive resources
27	$task{\rightarrow}fs{\rightarrow}lock$	The read-write synchronization lock used for file system access
28–32	$task{\rightarrow}signal{\rightarrow}utime, stime, gtime, cgtime, nvcsw$	Resource counters of signal structure for dead threads and child processes

3.2 Redundant feature-instance elimination

Now that we have shortlisted the features that have the potential to contribute positively towards classification

of benign and malicious processes, the next step is to eliminate redundant instances of feature values logged periodically. As we monitor the process control block of a process, the parameters in the process control block change significantly only if the process performs some new operation or tries to access new resources. While the process is running, the process control block may remain the same for a long time, resulting in numerous duplicate instances. To make the classification process faster, we add a new instance only if it is not a duplicate of the previous instance.

Defintion 4. ***Instance.*** *If x_{n,i,t_j} be the value of feature $f_i \in \mathcal{F}_{sel}$ for process N_n at time instance t_j, we define an Instance $\mathcal{I}_n(t_j)$ as the set of values of the selected features logged for the current process at the time instance t_j, then:*

$$\mathcal{I}_n(t_j) = \{x_{n,i,t_j}\} \mid 1 \leq i \leq N_{f_{sel}}$$

Defintion 5. ***Difference between Instances.*** *If x_{n,i,t_j} be the value of feature $f_i \in \mathcal{F}_{sel}$ for process N_n at time instance t_j, $\mathcal{I}_n(t_j)$ be the corresponding instance set, and $\mathcal{I}_n(t_{j-1})$ be the instance set at time t_{j-1}, then we define difference between the instances as the maximum absolute difference between corresponding feature values.*

$$D_{inst}(\mathcal{I}_n(t_j)) = \max |x_{n,i,t_j} - x_{n,i,t_{j-1}}|$$

for $1 \leq i \leq N_{f_{sel}}$.

We remove redundant instances by keeping only one instance from consecutive instances with a difference of zero. The selected instances $\mathcal{I}_{sel}$ are given by:

$$\mathcal{I}_n(t_j) \in \mathcal{I}_{sel} \iff D_{inst}(\mathcal{I}_n(t_j)) > 0$$

3.3 Time-series Segmentation and Frequency Information Extraction

After shortlisting the important features and removing the redundant instances, we need to extract sufficient information from the remaining instances that would allow us to train and test the information using a machine learning classifier. The first step is to segment the time-series data (Instances) in to different blocks/windows. For each window, we extract the frequency components information using *Discrete Cosine Transform*.

For a given time-series window (segment) k of size T and process N_n, the frequency information $\mathcal{C}_i(\omega)$ of feature $f_i \in \mathcal{F}_{sel}$ is given by:

$$\mathcal{C}_i(\omega) = \alpha(\omega) \times \sum_{j=1}^{T} (x_{n,i,t_j}) \cos(\frac{\pi 2j+1}{2T})$$

where

$$\alpha(\omega) = \begin{cases} \sqrt{\frac{1}{T}} & \text{for } \omega = 0 \\ \sqrt{\frac{2}{T}} & \text{for } \omega \neq 0 \end{cases}$$

Each instance $\mathcal{I}_j$ contains sets of frequency components information for all features.

$$\mathcal{I}_j = \{\mathcal{C}_i(\omega)\} \mid 1 \leq i \leq N_{f_{sel}}$$

The window $\mathcal{W}_k$ is a set of $WinSize$ such instances.

$$\mathcal{I}_j \in \mathcal{W}_k \mid WinSize \times (k-1) < j \leq WinSize \times k$$

3.4 Variance Accumulation for Time-series Segments

We have calculated the frequency components information of time-series segments for each feature. We measure the change in process execution behavior through the statistical measure *Variance*. As the change in process behavior is usually a gradual process and not a sudden one, we accumulate these changes over a period of time using *Cumulative Variance*. In this way, the gradual changes add up and give us a better estimation of the overall change in the frequency components of the time-series segments, thus enabling us to identify the hidden patterns in the execution of the process.

We calculate variance $\sigma^2_{W_k}$ of a window $\mathcal{W}_k$ as:

$$\sigma^2_{W_k} = \frac{1}{N} \sum_{i=1}^{N} (I_k - \mu_{W_k})$$

where μ_{W_k} is the mean (average) of the frequency component values for all instances in window $\mathcal{W}_k$.

We start with an initial value of cumulative variance $g_0 = 0$. We calculate cumulative variance g_k for each window $\mathcal{W}_k$ as:

$$g_k = \sigma^2_{W_k} + g_{k-1}$$

Figure 3 shows the time-series cumulative variances corresponding to some selected features for benign and malicious processes. The change in execution pattern for benign and malicious processes is succinctly captured in these graphs. A machine learning classifier can learn this gradual change in the frequency components of these feature values and generate rules which can be later used to distinguish between benign and malicious processes.

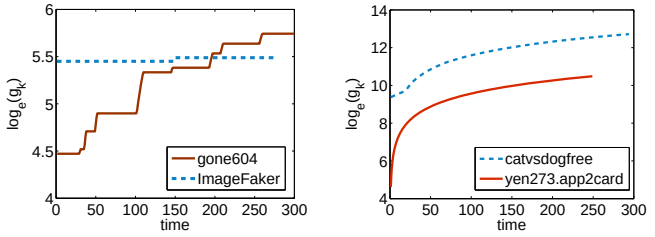
Now, we list and prove some important properties regarding the cumulative variance model that we have formulated for classification. In summary, we want to prove that the accumulated variance converges after some time (*Theorem 1* and can be modeled with a linear autoregressive model which is stable. By proving these properties, and then empirically plotting the coefficients of this model for our benign and malicious datasets in juxtaposition (Figure 5), we establish our assertion that the presented scheme is able to extract valuable hidden patterns in execution behavior of the processes, and thus, it is very well suited for classification.

Lemma 1. *Let g_k be the cumulative variance on window k of size S where $1 \leq k \leq K$. k is defined as $K = T/S$, then:*

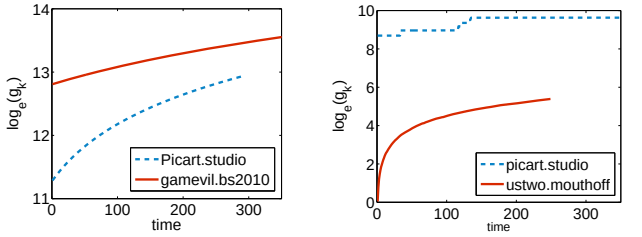
$$g_k = \sum_{k=1}^{K} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])$$

Proof: We have defined cumulative variance g_k as

$$g_k = g_{k-1} + f(X_k)$$



(a) `task→policy` (b) `task→active_mm→mm_users`



(c) `task→active_mm→shared_vm` (d) `task→active_mm→last_interval`

Fig. 3. Time-series Cumulative Variance for Benign and Malicious Processes

where

$$f(X_k) = \sigma^2(X_k)$$

Expanding g_{k-1} and iterating, we get

$$g_k = f(X_1) + f(X_2) + f(X_3) + ... + f(X_K)$$

Substituting $f(X_k) = \sigma^2(X_k)$,

$$g_k = \sigma^2(X_1) + \sigma^2(X_2) + \sigma^2(X_3) + \ldots + \sigma^2(X_K) \quad (1)$$

We know that:

$$\sigma^2(X_k) = \mathrm{E}[(X_k - \mu)^2]$$

$$\sigma^2(X_k) = \mathrm{Cov}(X_k, X_k)$$

$$\therefore\ \sigma^2(X_k) = \mathrm{E}[X_k^2] - \mathrm{E}^2[X_k] \quad (2)$$

Now, by combining Equation (1) and (2):

$$g_k = \mathrm{E}[X_1^2] - \mathrm{E}^2[X_1] + \ldots + \mathrm{E}[X_K^2] - \mathrm{E}^2[X_K] \quad (3)$$

Hence,

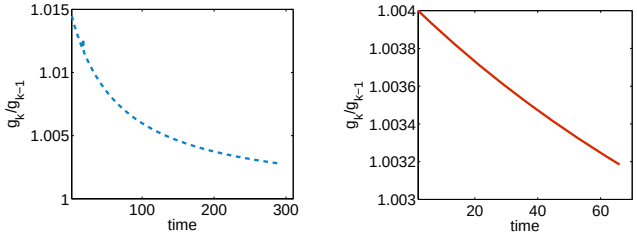
$$g_k = \sum_{k=1}^{K} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k]) \quad (4)$$

□

Theorem 1. *The function* $\frac{g_k}{g_{k-1}}$ *is given by*

$$\frac{g_k}{g_{k-1}} = 1 + \frac{\sigma^2(X_k)}{g_{k-1}} \quad (5)$$

and it is a bounded function and converges to 1, as time $T \to \infty$



(a) Benign Processes (b) Malware Processes

Fig. 4. Convergence of Cumulative Variance Rate for `task→usage.counter` (Theorem 1)

Proof: Using equation 4 in Lemma 1

$$\begin{aligned}
\frac{g_k}{g_{k-1}} &= \frac{\sum_{k=1}^{K} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} \\
&= \frac{\sum_{k=1}^{K} (\mathrm{E}[X_k^2])}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} - \frac{\sum_{k=1}^{K} (\mathrm{E}^2[X_k])}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} \\
&= \frac{\sum_{k=1}^{K-1} \mathrm{E}[X_k^2] + \mathrm{E}[X_K^2]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} - \frac{\sum_{k=1}^{K-1} \mathrm{E}^2[X_k] + \mathrm{E}^2[X_K]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} \\
&= \frac{\mathrm{E}[X_K^2]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} - \frac{\mathrm{E}^2[X_K]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} \\
&\quad + \frac{1}{1 - \frac{\sum_{k=1}^{K-1} \mathrm{E}^2[X_k]}{\sum_{k=1}^{K-1} \mathrm{E}[X_k^2]}} - \frac{1}{1 - \frac{\sum_{k=1}^{K-1} \mathrm{E}[X_k^2]}{\sum_{k=1}^{K-1} \mathrm{E}^2[X_k]}}
\end{aligned}$$

When accumulated variance is large,

$$\sum_{k=1}^{K} \mathrm{E}[X_k^2] \gg \sum_{k=1}^{K} \mathrm{E}^2[X_k]$$

Therefore, as $\lim_{g_{k-1} \to \infty}$

$$\frac{\sum_{k=1}^{K-1} \mathrm{E}^2[X_k]}{\sum_{k=1}^{K-1} \mathrm{E}[X_k^2]} \to 0, \quad \frac{\sum_{k=1}^{K-1} \mathrm{E}[X_k^2]}{\sum_{k=1}^{K-1} \mathrm{E}^2[X_k]} \to \infty$$

and

$$\frac{\mathrm{E}[X_K^2]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])}, \quad \frac{\mathrm{E}^2[X_K]}{\sum_{k=1}^{K-1} (\mathrm{E}[X_k^2] - \mathrm{E}^2[X_k])} \to 0$$

thus, by applying the limits to the above equation for $\frac{g_k}{g_{k-1}}$ becomes

$$\lim_{g_{k-1} \to \infty} \frac{g_k}{g_{k-1}} = 1$$

□

Theorem 2. *The presented model – variance accumulation of frequency components obtained using DCT, over the time*

varying windows of PCB parameters of benign and malicious processes – is linear and stable.

Proof: $AR(p)$ represent an autoregressive model of order p. We define g_k according to $AR(p)$ model as follows:

$$g_k = c + \sum_{i=1}^{p} \varphi_i g_{k-i} + \epsilon_k$$

where $\varphi_1 \dots \varphi_p$ are the parameters of the model, p is empirically evaluated as 4 and $c = 1$. For the model to be stable, the roots of polynomial $z^p - \sum_{i=1}^{p} \varphi z_{p-i}$ must lie in the unit circle.

AR parameters are calculated using method of moments (Yule walker equations)

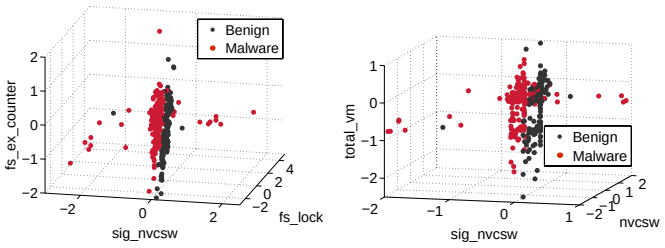
$$\gamma_m = \sum_{k=1}^{p} \varphi_k \gamma_{m-k} + \sigma^2_{\epsilon_k} \delta \tag{6}$$

Where, $m = 1, ..., p$, $\sigma^2_{\epsilon_k}$ is the standard deviation of the input noise ϵ_k, γ_m is the autocorrelation function of g_k. Equation (6) forms a system of equations that can be represented in matrix notation and solved for $\{\varphi_k; m = 1, 2, 3....., p\}$, once autocorrelation function γ_m of g_k is known. For $m = 0$, we solve separately using the following equation.

$$\gamma_0 = \sum_{k=1}^{p} \varphi_k \gamma_{-k} + \sigma^2_{\epsilon}$$

This can be solved for σ^2_{ϵ} once φ_m are known. After evaluating the model fit in all processes, we find that the average model fit measure is $\approx 85\%$. This proves our initial hypothesis that the variance accumulation of frequency components obtained using DCT, over the time varying windows of PCB parameters of benign and malicious processes can be modeled by a linear, stable statistical model. By visualizing the coefficients of statistical autoregressive (AR) model (see Figure 5), we can make sure that the underlying process model of benign and malicious applications is intrinsically different.

□



(a) One set of three coefficients

(b) Another set of three coefficients

Fig. 5. 3D Visual Representation of Autoregressive Models for Benign and Malware processes

4 TSTRUCTDROID FRAMEWORK

In this section, we present the architecture of *TStructDroid* framework for detection of malware applications on Android smartphones. This framework employs the in-execution dynamic analysis scheme presented in Section 3, based on time-series analysis of features obtained from monitoring of kernel process control blocks[9] on Android powered smartphones.

When a new application is launched on Android, the Binder IPC mechanism is used for sending a message to the Zygote process. The Zygote process is a special process on Android which is basically an instance of Dalvik VM with core libraries loaded as read-only. The Zygote process forks a new process, resulting in the new application being launched in a new Dalvik VM without unnecessary copying of shared core libraries [16]. As a consequence of the `fork` system call from Zygote, the kernel creates a child process and adds its process control block (`task_struct`) to a doubly linked circular linked list. The scheduler is responsible for execution of these processes in a multi-tasking manner.

The *TSStructDroid* framework runs in the kernel space as a (Loadable kernel module) LKM. Rooting is required on an Android supported device to get root privileges for loading and executing the kernel module. We have compiled and tested the kernel module on Samsung Galaxy Young device with Android Gingerbread distribution (Android 2.3.6, Kernel version 2.6.35.7). The kernel module executes periodically and performs the dynamic analysis of the user processes under execution.

The framework consists of the following three components: (1) Features Logger, (2) Features Analyzer & Processor, and (3) Classification Engine. The kernel module periodically extracts and dumps the contents of the process control blocks of the running processes from the doubly linked list of `task_structs` using the Features Logger component. Features Analyzer & Processor component is responsible for analyzing the extracted features, shortlisting the features, removing redundant instances, performing time-series transforms, and calculating statistical features from the transformed feature set. In the end, the Classification Engine uses tree based classifier to build classification rules and uses voting on a window of feature instances to determine if the process in execution is performing any malicious activity.

The architecture of the proposed framework is shown in Figure 6. Now we explain the working of each component of the framework in detail.

4.1 Features Logger

From the 99 preliminary parameters from the process control block (`task_struct` in Android kernel), our

9. Process Control Blocks in Linux/Android kernel are typically referred to as Task Structures (`task_struct`).

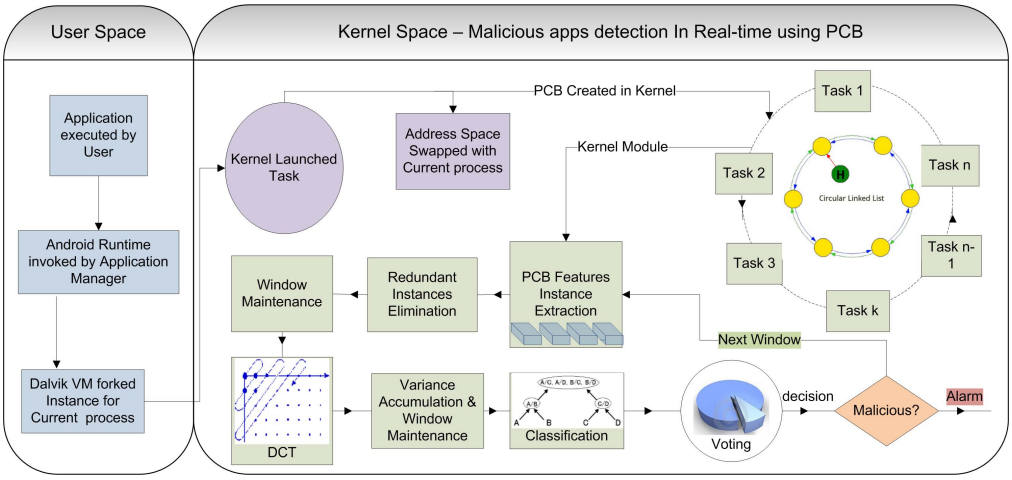


Fig. 6. Block diagram of malicious applications detection on Android in realtime using process control blocks (task_struct) - process flow in user and kernel space

scheme has shortlisted 32 features as described in Section 3. These shortlisted features are also listed in Table 1. The Features Logger component is responsible for logging these parameters. The logging is done periodically and for each process control block in the `task_struct` circular linked list. The feature analyzer processes the logged preliminary features in batches (or windows).

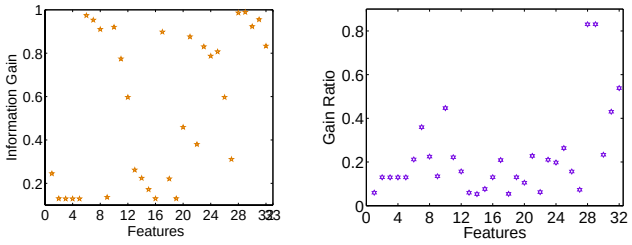
4.2 Features Analyzer

The Features Analyzer component analyzes the extracted features and implements the presented scheme for extraction of hidden patterns. It is responsible for removing redundant instances, creating windows (batches of instances), performing time-series transforms, and calculating statistical features from the transformed feature set (described in detail in Section 3).

4.3 Classification Engine

After mining the variation in frequency components of the current time-series segment (representing the execution behavior of the processes), we use a machine learning based classifier to judge a process as benign or malicious. To choose a classifier, we first visualize the statistical information in the frequency components of the shortlisted features. Information Gain (IG) and Information Gain Ratio (GR) of the frequency components are extracted. Figure 7 shows graphs of the IG and GR for the frequency components related to some selected features.

It turns out that the selected features have high Information Gain and Information Gain Ratio. Decision-Tree based classifiers use IG and GR for building and pruning decision tree models. Therefore, a decision tree based classifier such as J48 is an ideal choice for our framework. Moreover, in [17], it is shown that J48 is resilient to class noise because it avoids over fitting during learning and also prunes a decision tree for an optimum performance.



(a) Information Gain (b) Information Gain Ratio

Fig. 7. Information Theoretic Analysis of Frequency Components Related to Shortlisted Features Set $\mathcal{F}_{sel}$

4.4 Voting Method & Alarm

The classifier gives *Benign*|*Malicious* verdict for the frequency components in each time-series segment. We use a voting method for making a final decision. If among $\mathcal{W}_{vote}$ consecutive segments, more than half of the segments are declared as malicious, an alarm is raised and the process is killed. Otherwise, the process (and its dynamic analysis) continues to execute.

5 Performance Evaluation

In this section, we present the performance evaluation of our proposed framework. We have evaluated our framework on a dataset of 110 benign and 110 malicious real-world Android applications.

5.1 Classification Performance

In a typical two-class problem, such as malicious process detection, the classification decision of an algorithm may fall into one of the following four categories: (1) true positive (TP), classification of a malicious process as malicious, (2) true negative (TN), classification of a benign process as benign, (3) false positive (FP), classification of a benign process as malicious, and (4) false negative

(FN), classification of a malicious process as benign. We measure the classification performance using three standard parameters: (1) Detection Rate ($DR = \frac{TP}{TP+FN}$), (2) False Alarm Rate ($FAR = \frac{FP}{FP+TN}$), and (3) Detection Accuracy ($DA = \frac{TP+TN}{TP+TN+FP+FN}$).

As described in Section 2, we evaluate our framework using two different scenarios (Real-time Scenario and Standard cross-validation Scenario). We vary the time-resolution (δt) after which features are logged and the size of each segment (T). The classification results for Real-time scenario and Cross-validation scenario experiments are listed in Table 2. The time resolution δt is varied from 10ms to 40ms, and Segment Size T is varied between $5, 10, 20$ and 40 instances of frequency components information.

5.1.1 Real-time Scenario:

As described in Section 2, there are 220 different training and testing datasets in this scenario, each representing a case of zero-day malware detection. The results in Table 2 indicate that our framework is able to detect zero-day malware applications on Android with a detection rate of above 98% and a False Alarm Rate below 1% consistently. The overall detection accuracy lies within $98.6 - 99.5\%$. These results illustrate the strength of our framework in detecting zero-day malware by analyzing the in-execution patterns in the process control block structure of the applications.

5.1.2 Standard Cross-validation Scenario:

As described in Section 2, we create 10 different folds for training and testing. The results shown in the Table 2 indicate that our framework is able to detect malware applications on Android with a detection accuracy of above 92% consistently. The detection rate lies within $90 - 93.6\%$. This means that the framework detected at least 9 out of every 10 randomly chosen malware applications for which it was not trained. However, the False Alarm Rate is relatively higher and varies between 5.4% and 7.3%.

We now discuss the effect of changing the values of different choice parameters of the framework. For each choice parameter, we first describe the expected effect of a variation in its value, and then validate this reasoning through the presented experimental results.

5.1.3 Effect of change in Time Resolution (δt):

As specified earlier, we use the term *time resolution* (δt) to specify the time interval after which values of parameters in the process control block structure of a process are logged. There are some important trade-offs that need to be kept in mind while choosing a suitable value for δt. A small value of δt means that the process will be monitored more frequently resulting in a detection of minor variations in the parameters. This is especially useful for processes that carry out a distinct malicious activity for a very short time. However, this can also lead to higher false alarms as the short time change in activity may not be a strong indicator of a malicious activity. Choosing a larger value for δt can significantly help in reducing the processing overhead as the monitoring of the process is done less frequently resulting in fewer context switches and fewer memory and timing spent in classification. From Table 2, we observe that a change in δt has little effect on the detection rate. For example, the detection rate stays 99.09% (Real-time) and 90% when δt is varied from 10ms to 40ms ($T = 5, \mathcal{W}_{vote} = 30$). However, the FAR decreases from 7.27% to 6.36% (Cross-validation) when δt is varied from 10ms to 40ms ($T = 40, \mathcal{W}_{vote} = 30$). The overhead of the framework decreases by a factor of 4 for $\delta t = 40$ms as compared to $\delta t = 10$ms.

5.1.4 Effect of change in Segment Size (T):

As specified earlier, we divide the time-series data into segments of fixed size before calculation of frequency components. We expect the increase in the number of instances in a segment (T) to result in a slight increase in detection accuracy as better frequency component analysis could be performed. However, a large value of T means that each segment would be processed after a longer delay, hence the delay in the detection of a malware would be increased. The results in Table 2 support our reasoning. There is a slight increase in detection accuracy (99.1% to 99.55% for Real-time, 92.27% to 93.64% for Cross-validation) when Segment Size T is increased from 5 instances to 40 instances ($\delta t = 40$ms$, \mathcal{W}_{vote} = 30$).

5.1.5 Effect of change in Voting Window Size ($\mathcal{W}_{vote}$):

The classifier gives *Benign|Malicious* verdict for the frequency components in each time-series segment. As described earlier, we use voting on a window of segments to make the decision. Similar to the effect of variation in segment size, an increase in the size of this parameter $\mathcal{W}_{vote}$ should increase the detection accuracy and decrease the false alarm rate. A large value of $\mathcal{W}_{vote}$ however would increase the detection delay, and might also result in missing tiny malicious activities/applications which only run for a short time. The results in Table 2 have been reported for a fixed value of $\mathcal{W}_{vote} = 30$ segments. We empirically chose this value by varying the value of this window and observing the change in DA and FAR. Figure 8 shows that a value of $\mathcal{W}_{vote} = 30$ segments results is an optimal choice for increasing DR and reducing FAR for Realtime scenario ($\delta t = 10$ms$, T = 10$). The results for cross-validation scenario are similar to Figure 8 and are therefore left out to avoid repetition.

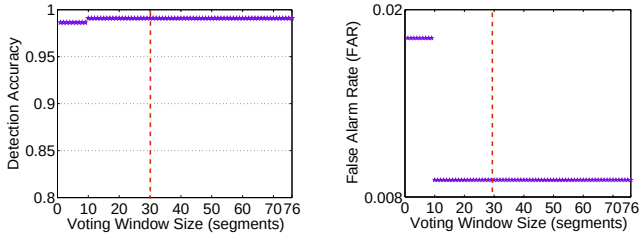
It is evident from these results that the framework is able to detect zero-day malware with a high detection accuracy, and produces low false alarm rate.

5.2 Processing Overheads

Classification accuracy of a malware detection framework is very important. However, if this accuracy comes

TABLE 2
Classification Performance Results for Real-time & Cross-validation scenarios

Time Resolution (δt ms)	Segment Size (T instances)	Voting Window Size ($\mathcal{W}_{vote}$ segments)	Real-Time Scenario (%)			Cross-validation Scenario (%)		
			DR	FAR	DA	DR	FAR	DA
10	5	30	99.09	0.91	99.1	90	5.45	92.27
10	10	30	99.09	0.91	99.1	90	5.45	92.27
10	20	30	99.09	0.901	99.55	90.91	5.45	92.73
10	40	30	98.18	0.901	98.64	93.64	7.27	93.18
20	5	30	99.09	0.909	99.1	90	5.45	92.27
20	10	30	99.09	0.909	99.1	90.91	5.45	92.73
20	20	30	99.09	0.909	99.1	90.91	5.45	92.73
20	40	30	98.18	0.909	98.64	90	5.45	92.27
40	5	30	99.09	0.909	99.1	90	5.45	92.27
40	10	30	99.09	0.909	99.1	90.91	5.45	92.73
40	20	30	99.09	0.91	99.55	90.91	5.45	92.73
40	40	30	100	0.91	99.55	93.64	6.36	93.64



(a) Detection Accuracy (b) False Alarm Rate

Fig. 8. Effect of change in Voting Window Size $\mathcal{W}_{vote}$ (Real-time Scenario, $\delta t = 10$ms, $T = 10$ instances)

with an excessive performance overhead, users are more likely to disable/remove it in an attempt to make their smartphones more responsive. In this section, we discuss the processing overhead imposed by our proposed framework.

5.2.1 Device under test:

The experiments were performed on a Samsung Galaxy Young S5360 device with an 832 MHz ARMv6 processor, 290 MB RAM and 160 MB built-in storage. We decided to use a low end device for the experiments so as to benchmark the performance of our framework in a strictly limited memory and processing resources configuration.

5.2.2 Estimation of processing overhead:

To calculate the processing overhead, we first estimated the running time of all components of the framework in one classification cycle. Assume that we use a time-resolution of $\delta t = 10$ms. Moreover, for the purpose of this discussion, we choose a segment size $T = 10$ instances and voting window size $\mathcal{W}_{vote} = 30$ segments. One classification cycle would take place in a time period of $\delta t \times T \times \mathcal{W}_{vote} = 3000$ms.

The features are logged after time δt. Therefore, there is one context switch and 32 features are logged after every 10ms. Through multiple iterations and averaging the results on our low-end device, we estimate that one such operation takes 48.8284μs to complete. Checking if the instance is redundant (same as last logged instance) and eliminating it results in an overhead of approx. 30μs. For $T = 10$ and $\mathcal{W}_{vote} = 30$, there would be an average overhead of $10 \times 30 \times (48.8284\mu\text{s} + 30\mu\text{s}) =$ **23.648ms** in one classification cycle for context switches, logging and redundant instances elimination.

Calculation of frequency components in a segment through Discrete Cosine Transform (DCT) takes 320μs on average for the 10×32 matrix of feature values in one segment ($30 \times 320\mu\text{s} =$ **9.6ms** per classification cycle), while calculation of accumulated variation of these frequency components takes on average approx. 570.63μs per segment ($30 \times 570.63\mu\text{s} =$ **17.119ms** per classification cycle) on our low-end Android device. The testing phase of J48 classifier consists of simple numerical comparisons as the rules in the decision tree are fired. We estimate that this testing process introduces a delay of 5μs per segment ($30 \times 5\mu\text{s} =$ **0.15ms** per classification cycle) for the classification of the activity during that time segment as benign or malicious. The voting process takes approx. **0.03ms** for making a decision and raising an alarm if needed.

Therefore, in a classification cycle occurring over a period of 3000ms, the combined estimated overhead of all components of the framework on average is approx. $23.648\text{ms} + 9.6\text{ms} + 17.119\text{ms} + 0.15\text{ms} + 0.03\text{ms} =$ **50.547ms**. This corresponds to an average processing overhead of **1.685**%. A low overhead of 1.685% makes the proposed framework an ideal candidate for malware detection on resource constrained mobile device.

5.2.3 System performance degradation:

Although, there may be only one application running in the foreground, a typical Android based smartphone has system services/background applications running at the same time. Therefore, the actual performance degradation in the presence of the framework may be significantly higher. Moreover, different applications have different execution patterns. Some applications are CPU intensive, while others tend to use I/O resources more often. Similarly, the memory requirements of different

TABLE 3
System Performance Degradation Analysis on Android

Application	Baseline Exec. Time (s)	Exec. Time with TstructDroid (s)	Overhead (%)
Zip (Archive)	127.2	131.5	3.27
Zip (Best Compress)	67.86	69.94	2.97
Zip (Best Speed)	50.39	52.01	3.11
Zip (Deflated)	132.12	141.81	6.83
Zip (Filtered)	147.59	154.27	4.33
File Copy	190.6	196.95	3.22
File Search	131.68	134.91	2.4
Average			**3.73**

applications differ significantly resulting in variable frequency of page faults. Therefore, the actual degradation in the performance may be different for different types of applications.

We have performed experiments to measure the performance degradation experienced by an application when the framework is in full operation. We have created an ensemble of custom Android applications each of which performs different operations. Some of these operations are CPU-intensive, some are I/O-bound, while some others are a combination of both behaviors. The algorithms for these operations have variable memory requirements as well. These operations include five different algorithms for file compression, a file (folder) copy operation and text search within file contents. Four different folders containing different files (total folder size: 425 MB, 500 MB, 850 MB and 1275 MB) have been used as input to these applications. After performing the experiments on different inputs and for multiple iterations, average performance degradation results are reported in Table 3. As expected, the performance overhead varies with the type of operation, and the average performance overhead is comparatively higher (**3.73**%). However, an average overhead of 3.73% still makes the proposed framework an ideal candidate for malware detection on resource constrained mobile device.

5.3 Evasion Analysis and Mitigation Techniques:

We now discuss a couple of ways in which a malware can attempt to evade getting detected by our framework. We also describe the available techniques to mitigate such evasion tactics.

The first strategy that a malware can employ is to mimic the execution behavior of a benign process. To do this, a malware must know the values of benign sets of features and their variation pattern. However, most of these features depend heavily on the configuration of a device (such as physical memory, cache, storage and paging mechanisms etc.), typical use and other processes & services affecting the kernel state. Therefore, we expect that estimating such a pattern for a particular device in a particular state can be daunting for a malware application. Our experiments include a significant number of *Trojan* malware applications (applications that appear benign and masquerade as common useful applications), which are efficiently detected by the framework, therefore, attempting to hide malicious execution pattern inside benign patterns is not very useful for evasion purpose.

A much more serious scenario is that of a *Rootkit*. Rootkits infect a system's kernel and thus get full access to the system. The typical use of rootkit exploits is to hide activity of other malware installed on the system. Moreover, the rootkits are not run in limited permissions environment like other types of malware and thus are able to monitor and modify other applications and sensitive resources on the smartphone device. A rootkit can either monitor another benign process on the system and make the malware application mimic similar behavior, or it can simply fool the framework in to reading the process control block of another benign process when an attempt is made to read process control block of the malware. It can even simply stop the framework from working. However, the problem of rootkits is well known in the OS security community for years now. Moreover, having a rootkit installed on the system means that the system is already compromised. It is possible to mitigate this threat by using our proposed framework in conjunction with one of the several available rootkit detection frameworks for Android (and other smartphones) [18] [19] [20].

6 Related Work

Dynamic malware detection techniques intend to detect malicious programs during or after the program execution by leveraging the runtime information. Such techniques may involve monitoring execution patterns of programs, performing the taint-analysis and estimating their impacts on the OS. As a result, they are able to withstand code obfuscation or polymorphism techniques. A number of dynamic malware detection frameworks have been proposed in literature. In this section, we describe some of the latest proposed frameworks and their shortcomings.

Dini et al. have presented a multilevel anomaly detection technique for detecting Android malware (*MADAM*) [15]. The proposed framework operates in kernel and user space simultaneously and is capable of detecting previously unseen, zero-day malicious applications. A rich feature set is derived due to a multilevel view of the system events in both spaces. The K-nearest neighbors (KNN) algorithm is then applied during classification process. The operation of the framework can be divided into training, learning and operative phases. The machine learning classifier can adapt to new changes by incorporating new feature vectors in training and learning set at run-time. An average detection rate of 93% along with an average false positive rate of 5% is reported upon evaluation on a small dataset.

A hybrid framework for automatic malware detection *Smartdroid* is proposed in [21] which monitors user interface interactions. In the static analysis, a *static path selector* builds activity control graphs and function call

graphs. Function call graphs are updated for indirect & event driven API calls by employing the technique given by Woodpacker [22]. The dynamic analysis is performed by making a modification in source code of Android framework while a Restrictor component is added to limit the new activities that are created after UI interaction. This framework is suitable only for online analysis due to its requirement of changes in the Android framework and the high analysis time overhead. During the analysis, some of the complicated indirect UI-based trigger conditions were also missed.

TaintDroid [23] is an information flow tracking tool for Android smartphones that gives dynamic taint tracking capability to the system. This framework can track multiple sources of private data by applying labels to the sensitive data. These labels work on four levels of tracking namely variable-level, method-level, file-level and message-level. An alarm is raised if the labeled data leaves the system through an un-trusted third party application. Authors have evaluated TaintDroid extensively over thirty most commonly used android applications and zero false positive is reported. System overhead is large due to labels storage adjacent to sensitive data and their propagation. An average of 14% processing overhead in executing Java instructions and an average of 4.4% memory overhead is observed. The major limitation of TaintDroid is that it only looks for explicit information flow; therefore, it is still possible to circumvent taint propagation through implicit flow of information.

In [24], the authors have proposed *DroidScope* which is a multilevel semantic analysis tool that performs dynamic profiling and information tracking to detect malicious behavior and privacy leaks in Android based smartphone applications. The tool runs in a virtual environment and logs instruction traces, API calls (at OS level and Dalvik VM level) and uses taint analysis to discover leakage of sensitive information. The tool has been tested on only two real world malware samples.

Android Application SandBox framework is proposed in [25] for malicious software detection on Android. This framework first performs static analysis of user applications. The applications are then transmitted to a remote server that executes them in the sandbox and performs clustering and analysis of the generated logs to detect malicious patterns. The dataset used for testing Sandbox is small and analysis of the time and memory overhead is also not included. Another framework that uses similar concept of decoupled security is an API based malware detection system *Paranoid Android* [26].

Burguera et.al have developed a dynamic framework called *Crowdroid* which recognizes Trojan-like malware. It takes into account the fact that genuine and trojan affected applications differ in types and number of system calls during the execution of an action that requires user interaction. Authors have reported results on a small dataset. The false-alarm rate is significantly high (20%). The authors have not discussed the robustness of their features set [27].

Andromaly is another IDS which monitors both the system and user behaviors by observing several parameters, spanning from sensors activities to CPU usage [28]. The authors have developed four custom malicious applications to evaluate the ability to detect anomalies. They have created four different training/testing scenarios and have reported competitive results. Andromaly degrades performance of smartphone by 10% with the malware detection time of 5 sec. Also, it is not evaluated over real-world Android malware dataset.

Confused deputy attacks — (deputing tasks to a more privileged application through publicly defined interfaces such as Intents) — allows an application escalate its privileges indirectly. The authors of [29] have proposed Quire framework that attempts to solve this problem by restricting the inappropriate use of application's permissions through its public interface and providing a trusted communication mechanism between applications using remote procedure calls.

Some of the common shortcomings for dynamic malware detection approaches in published literature include the following: (1) Significant processing overheads, (2) Evasion through mimicry attacks, (3) High false alarm rates, and (4) Lack of testing on real-world malware dataset.

7 Conclusion

As the smartphone devices have become a basic necessity and their use has become ubiquitous in recent years, the malware attacks on smartphone platforms have escalated sharply. As an increasing number of smartphone users tend to use their devices for storing privacy-sensitive information and performing financial transactions, there is a dire need to protect the smartphone users from the rising threat of malware attacks. In this paper, we have presented a realtime malware detection framework for Android platform that performs dynamic analysis of smartphone applications and detects the malicious activities through in-execution monitoring of process control blocks in Android kernel. Using information theoretic analysis, time-series feature logging, segmentation and frequency component analysis of data, and machine learning classifier, this framework is able to detect real world malware applications for Android while producing very low false alarms. We have used a realworld dataset of 110 benign and 110 malware Android applications. The experiments show that our framework is able to detect the zero-day malware with over 98% accuracy and less than 1% false alarm rates. Moreover, the system performance degradation caused by or framework is only 3.73% on average for a low-end Android smartphone, making it ideal for deployment on resource constrained mobile devices.

References

[1] Gartner, "Gartner says worldwide smartphone sales soared in fourth quarter of 2011 with 47 percent growth,"

http://www.gartner.com/it/page.jsp?id=2017015 [last-viewed-July-2-2012], 2012.

[2] Mashable-Tech, "There will be more smartphones than humans on the planet by years end," http://mashable.com/2012/02/14/more-smartphones-than-humans/, 2012, accessed: Nov 15, 2012.

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

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

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

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

[7] Kindsight-Inc., "Mobile malware statistics 3q 2012," *Kindsight Security Lab: Malware Report Q3 2012*, 2012.

[8] S. HILL, "Top 3 android security apps, do they protect you?" *http://www.digitaltrends.com/mobile/top-android-security-apps/(last-viewed-on-December-10-2012)*, 2012.

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

[10] O. Sukwong, H. Kim, and J. Hoe, "Commercial antivirus software effectiveness: An empirical study," *Computer*, pp. 63–70, 2010.

[11] J. Oberheide and F. Jahanian, "When mobile is harder than fixed (and vice versa): demystifying security challenges in mobile environments," in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*. ACM, 2010, pp. 43–48.

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

[13] M. A. Akbar, F. Shahzad, and M. Farooq, "The droid knight: a silent guardian for the android kernel, hunting for rogue smartphone malware applications," in *23rd Virus Bulletin International Conference (VB2013)(Inpress)*. VB, 2013.

[14] F. Shahzad, M. A. Akbar, S. Khan, and M. Farooq, "Tstructdroid: Realtime malware detection using in-execution dynamic analysis of kernel process control blocks on android," nexGIN RC, FAST-National University, Islamabad, Pakistan, Technical Report TR-nexGINRC-2013-02, 2013.

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

[16] D. Ehringer, "The dalvik virtual machine architecture," *Techn. report (March 2010)*, 2010.

[17] I. Witten and E. Frank, *Data mining: Practical machine learning tools and techniques, second edition*. Morgan Kaufmann, 2005.

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

[19] J. Joy and A. John, "A host based kernel level rootkit detection mechanism using clustering technique," *Trends in Computer Science, Engineering and Information Technology*, pp. 564–570, 2011.

[20] R. Brodbeck, "Covert android rootkit detection: Evaluating linux kernel level rootkits on the android operating system," Defense Technical Information Center, Tech. Rep., 2012.

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

[22] W. Zhou, Y. Zhou, X. Jiang, and P. Ning, "Detecting repackaged smartphone applications in third-party android marketplaces," in *Proceedings of the second ACM conference on Data and Application Security and Privacy*. ACM, 2012, pp. 317–326.

[23] W. Enck, P. Gilbert, B. Chun, L. Cox, J. Jung, P. McDaniel, and A. Sheth, "Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones," in *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, 2010.

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

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

[26] G. Portokalidis, P. Homburg, K. Anagnostakis, and H. Bos, "Paranoid android: Versatile protection for smartphones," in *Proceedings of the 26th Annual Computer Security Applications Conference*. ACM, 2010, pp. 347–356.

[27] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani, "Crowdroid: behavior-based malware detection system for android," in *Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices*. ACM, 2011, pp. 15–26.

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

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

TABLE 4
List of Collected Benign and malware Applications

No	Benign Apps	No	Benign Apps	No	Malicious Apps	No	Malicious Apps
	Games	55	baum.ImageFaker-8		**Trojans**	58	gonein604
1	vszombies	56	picsart.studio-35	1	DDS.STiNiTER	59	gonein605
2	papertoss	57	shinycore.picsayfree	2	M.space.Sexypic	60	Dougalek.-gamatome
3	candydroid.breakpoint		**Misc.**	3	N.ZitMo.Sec.Prim1	61	Dougalek.sir-gamatome
4	DragRacing	58	scmonkeys.pickup	4	N.ZitMo.Sec.Prim2	62	Dougalek.himatubu-
5	doodle.retaurant	59	wpaper.bkgrnd.hd	5	N.ZitMo.Sec.Prim3	63	Dougalek.out-tubebe
6	droidhen.car3d	60	android.bestapps-35	6	N.ZitMo.Sec.Prim4	64	Dougalek.tube-gatome
7	forestman	61	evernote.skitch097	7	N.ZitMo.Sec.Prim5	65	Dougalek.jp.waraeru-
8	ectricsheep.edj	62	bbt.downloadall	8	LuckyCat.Service	66	Plankton.zoric.celebrity
9	pagossoft.trialdemo	63	jcwp.S4305	9	Qicsomos.rrieriq	67	Plankton.butterflies
10	bmx-b+B39oy+B36	64	lifequotes	10	com.zft	68	Plankton.chompsms
11	BubbleShoot-4	65	penguinfree-16	11	FakeInstagram-are.app	69	Plankton.SkatingMadnessP
12	catvsdogfree	66	jp.fujivol-15	12	FakeTimer.btm.ser	70	Plankton.Puzzle.Tales
13	icenta.sudoku.ui	67	shant.app.jokes	13	FakeTimer.ctm.ser	71	Plankton.Livewallpaper
14	junerking.pinball	68	krzysiek.afc.iqt-7	14	FakeTimer.dtm.ser	72	FindAndCall.co.egv
15	unblockmefree	69	simple.a-22	15	FakeTimer.k.ser	73	BatteryDoctor
16	kmagic.solitaire	70	jraf.android.nolock	16	FakeTimer.mtm.ser	74	Fake-token.generator
17	motoxmayhem1lite	71	.wordsearchfree.arff	17	PJApps.LivePrints	75	why.whackAMole
18	oldwang1.darts	72	cing.spades-ads.arff	18	SMSZombie.gmdcd	76	Steek-ieldBadCompany2
19	polarit.thunderlite	73	d.app.dialertab.arff	19	SMSZombie.dh	77	Steek-10D78.BloonsTD4
20	reverie.bubble	74	le.android.talk.arff	20	SMSZombie.livepicker	78	Steek-llOfDutyZombies
21	fallingball	75	m.android.email.arff	21	SMSZombie.hxmv696	79	Steek-RioCityofSaints
22	toiletpaper	76	m.awesome.facts.arff	22	SMSZombie.xqxmn18	80	Steek-i-T10D78.FIFA12
23	susichain.activity	77	mots.WordSearch.arff	23	SMSZombie.zqbb1221	81	Steek-WestCoastHustle
24	rovio.angrybirds	78	newbenignfilelist	24	SMSZombie.lzll	82	Steek-GlobalWarRiot
25	bubble8	79	ngtech.rushhour.arff	25	SMSZombie.-albumsho-	83	Steek-.JetpackJoyride
26	tapglider-5	80	om.skype.raider.arff	26	Vdloader.Ninja-Chicken	84	Steek-D78.MaddenNFL12
27	osaris.turboflydemo	81	pp.controlpanel.arff	27	Vdloader.LiveWPcube	85	Steek-0D78.TouchGrind
28	lightracer-21	82	pp.weatherclock.arff	28	Loozfon.lin.ero	86	Steek-T10D78.RopenFly
29	aifactory.chess.free	83	roid.pulsepaper.arff	29	Loozfon.ap.ken	87	Steek-8.NinJumpDeluxe
	Image Viewer	84	s.category.quiz.arff	30	AngryBirds.rovio.ads	88	Steek-0D78.WorldOfGoo
30	imageshrinklite	85	world.DoYouKnow.arff	31	FakeToken.generator	89	Steek-8.ZombieHighway
31	roidapp.photogrid		**Android Apps**	32	Counterclank.ladies3	90	Steek-Ipad2App
32	perfectviewer	86	android.browser	33	MMarketPay.mediawoz		**BackDoors**
33	thinkdroid.amlite	87	android.youtube	34	MMarketPay.mediawoz2	91	DroidKungFu.cuttherope
34	ImageViewer	88	android.apps.maps	35	Geinimi.JewelBears	91	DroidKungFu-pl.byq
35	androidcomics.acv	89	android.deskclock	36	Geinimi-A.gamevil	93	DroidKungFu.myvpn
36	imageviewerminor	90	anndroid.calander	37	Geinimi.Shoppers.sgg.sp		**Root Exploits**
37	jsimagefinder	91	android.calendar	38	Geinimi.Kosenkov.Pr	94	ITFunz.supertools
	Sketch Tools	92	android.settings	39	Geinimi.Chinese.maps	95	psufou.su
38	bejoy.minipaint	93	p.voicerecorder	40	Gamex.SD-Booster	95	Z4root:three
39	bejoy.sketchmovie1	94	anroid.app.camera	41	Gamex.SD-Booster2	97	Lotoor - App2card
40	PicassoMirror	95	app.fmradio	42	Arspam AlSalah	98	Root-smart
41	drawinggame2		**Notes**	43	Steek.thouch.lite		**Bots**
42	imadain.sketch.tab	96	gss.app.notepad-19	44	Steek.Ipad2App	99	Android-CI4.updater
43	sketchit	97	houmiak.desknote	45	Dialer.VoiceChangeIL		**Misc.**
44	sketchbookexpress	98	systems.super-note	46	FakeAngry.screenofflock2	100	ustwo.mouthoff
45	android.sketch2	99	markspace.fliqnotes	47	FakeAngry.screenofflock	101	android.shotgun
	Text Editor	100	com.mbile.notes	48	DroidDreamLight	102	cn.yufu16.merry
46	nsweringMachine	101	movinapp.quicknote	49	FakeInstaller.skyscanner	103	cts.JewelsMania
47	simpletextwidget-5	102	quicknotes.views	50	FingerPrint.screensvr	104	miniarmy.engine
48	pandora.jota-73	103	notepad.color.note	51	Advancedfm-2150	105	instantheartrate
49	BookSite-5	104	suishouxie.freenote	52	Advancedfm-2200	106	andoid.computerlab
50	simplenotepad	105	threebanana.notes	53	Zimto2012.service	107	oregame.drakula
51	paulmach.textedit-15	106	inkpad.notepad.notes	54	Droid-DreamLight	108	HamsterSuper.game
	Image Editor	107	.noteeverything		**SpyWare/Adware**	109	per.mobi.eraser-3600
52	photoeditorulitimate	108	my.handrite	55	gonein60	110	zyhamster-Super
53	photo.editor	109	d.demo.notepad3	56	gonein602		
54	jellybus.fxfree-18	110	nl.jacobras.notes-12	57	gonein603		