Hey, You, Get Off of My Market: Detecting Malicious Apps in Official and Alternative Android Markets

Yajin Zhou Zhi Wang Wu Zhou Xuxian Jiang
Department of Computer Science
North Carolina State University
{yajin_zhou,zhi_wang,wzhou2}@ncsu.edu jiang@cs.ncsu.edu

Abstract

In this paper, we present a systematic study for the detection of malicious applications (or apps) on popular Android Markets. To this end, we first propose a permission-based behavioral footprinting scheme to detect new samples of known Android malware families. Then we apply a heuristics-based filtering scheme to identify certain inherent behaviors of unknown malicious families. We implemented both schemes in a system called DroidRanger*. The experiments with* $204,040$ *apps collected from five different Android Markets in May-June 2011 reveal* 211 *malicious ones:* 32 *from the official Android Market (*0.02% *infection rate) and* 179 *from alternative marketplaces (infection rates ranging from* 0.20% *to* 0.47%*). Among those malicious apps, our system also uncovered two zero-day malware (in* 40 *apps): one from the official Android Market and the other from alternative marketplaces. The results show that current marketplaces are functional and relatively healthy. However, there is also a clear need for a rigorous policing process, especially for non-regulated alternative marketplaces.*

1 Introduction

Smartphones are becoming increasingly ubiquitous. A recent report from Gartner [12] shows that there are over 100 millions of smartphones sold in the first quarter of 2011, an increase of 85% over the last year. The popularity is also partially propelled with the large collection of feature-rich smartphone applications (or apps) in various marketplaces. For example, on May 10, 2011, Google announced [14] that the official Android Market reached the $200,000$ app milestone. Moreover, in addition to the official marketplaces (from platform providers such as Google and Apple), there also exist a number of third-party or alternative ones (e.g., Amazon AppStore for Android [4]), which further boost the popularity. These centralized marketplaces or app stores streamline the process of browsing, downloading and installing a variety of apps - therein facilitating the use of smartphones.

Unfortunately, such popularity also attracts the attention of malware authors. A few reports [19, 20, 25] already showed the presence of malicious apps in these marketplaces. For instance, `DroidDream` [19] and `DroidDreamLight` [25] were detected from the official Android Market in March and May 2011, respectively. While these reports offer detailed analysis about the spotted (individual) malware, they do not provide a systematic view of the overall health of existing Android Markets. Recently, Enck *et al.* [36] studied $1,100$ free apps from the official Android Market and attempted to understand a broad range of security-related metrics among them (as an indication of the overall app security). However, the study is limited in only examining a small number of apps, i.e., top 50 free apps from the 22 app categories.

In this paper, we present a systematic study to better understand the overall health of existing Android Markets, including both *official* and *unofficial* (third-party) ones. In particular, our goal here is to detect malicious apps on these marketplaces. To do that, we use a crawler to collect all possible (free) apps we can obtain from five representative marketplaces within a two-month period, i.e, May and June 2011. In total, we have collected $204,040$ apps. To detect malware infection among them, we take the following approach. (1) First, in order to detect the infection from known malware, we propose a scalable and efficient scheme called *permission-based behavioral footprinting*. Instead of taking a traditional approach with content invariants as malware signature, our scheme takes a scalable design by initially filtering out these apps based on the inherent Android permissions required by the malware for its wrongdoings and then matching them with malware-specific behavioral footprints (that characterize the malware behavior – Section 2.1). (2) Second, in order to detect unknown malware, we propose a *heuristics-based filtering scheme* that defines suspicious

behaviors from possibly malicious apps and then uses them to detect suspect apps (Section 2.2). Example heuristics include the suspicious attempts to dynamically fetch and execute code from a remote untrusted website. For each detected suspect app, we further dynamically monitor its runtime execution to confirm whether it is truly malicious or not. If the app is malicious and does not appear in our (malware) database, we consider it zero-day and then generate the corresponding permission-based behavioral footprint in a feedback loop to detect other samples in the collection.

We have implemented both schemes in a system called *DroidRanger* and used the system to detect infected apps in the collected $204,040$ samples from five different marketplaces. Among the collected apps, $153,002$ of them come from the official Android Market and the rest $51,038$ apps come from four other (alternative) marketplaces. To evaluate the effectiveness of our system, we have initially generated behavioral footprints from 10 known Android malware families[1]. Based on these footprints, DroidRanger successfully detected 171 infected apps among our collection: 21 infected apps in the official Android Market and 150 in four other alternative marketplaces. Moreover, we have implemented two additional heuristics to capture suspicious dynamic code loading behavior, either remotely or locally. These two heuristics effectively lead to the discovery of two sophisticated zero-day malware with 40 samples in our collection (Section 3)[2].

Overall, by combining both schemes to detect known and zero-day Android malware, DroidRanger reported 211 infected apps in our collection. If we calculate the infection rate, about 0.02% of apps from the official Android Market are infected while 0.20% to 0.47% of apps from alternative marketplaces are infected. As a result, the infection rate in alternative marketplaces is an order of magnitude higher than the official marketplace. Considering the recent trend in observing an unprecedent growth of mobile malware, we believe these results pose a clear call for better security protection in both official and unofficial marketplaces.

In summary, this paper makes the following contributions:

- To the best of our knowledge, DroidRanger is the first systematic study on the overall health of both official and unofficial Android Markets with the unique focus on the detection of malicious apps.

- To perform the study, we have collected $204,040$ Android apps in May and June 2011 from five popular Android Markets. To allow for scalable and efficient detection of both known and unknown malicious apps, we have accordingly proposed two different schemes, *permission-based behavioral footprinting* and *heuristics-based filtering*. We believe this is one of the most extensive study ever performed to understand the security of existing Android Markets.

- We have implemented our techniques in DroidRanger. When applied to the collected apps, our system successfully detected 211 malicious apps. Among them, our heuristics-based filtering scheme leads to the discovery of two sophisticated zero-day malware with 40 samples – 11 of them appear in the official Android Market. Our reporting of them to the respective marketplaces has immediately resulted in their removal.

The rest of this paper is organized as follows: We first describe the overall system design of DroidRanger in Section 2. We then present the system prototype and related evaluation in Section 3. After that, we discuss possible limitations and future improvements in Section 4. Finally, we present related work in Section 5 and conclude our work in Section 6.

2 Design

To systematically detect malicious apps in existing Android Markets, we have three key design goals: *accuracy*, *scalability*, and *efficiency*. Accuracy is a natural requirement to effectively detect malicious apps in current marketplaces with low false positives and negatives. Scalability and efficiency are challenging as we need to accommodate the large number of apps that need to be scanned. Specifically, with our current collection of more than $200,000$ apps, if it takes 6 seconds to examine a single app, a full scanning of the collection for known malware will require more than two weeks to complete. In other words, the traditional approach of performing deep scanning (with content invariants as signatures) for each app may not be desired in this study. Therefore, in our design, we choose to take advantage of the unique information embodied in each app (i.e., various Android permissions in the app-specific manifest file) and develop a scheme called *permission-based behavioral footprinting* for scalable and efficient malware detection. For unknown malware, we choose to devise a few heuristics that define suspicious behaviors from possibly malicious apps and then use them to effectively identify suspect apps for runtime monitoring.

Figure 1 shows an overview of our approach. In essence, DroidRanger leverages a crawler to collect Android apps from existing Android Markets and saves them into a local repository. For each collected app, DroidRanger extracts fundamental properties associated with each app (e.g., the requested permissions and author information) and orga-

[1]The list of known Android malware is shown in Table 4.

[2]To better protect users, we have promptly provided samples of these zero-day malware to leading mobile anti-virus software vendors. They are now detectable by most mobile anti-virus software.

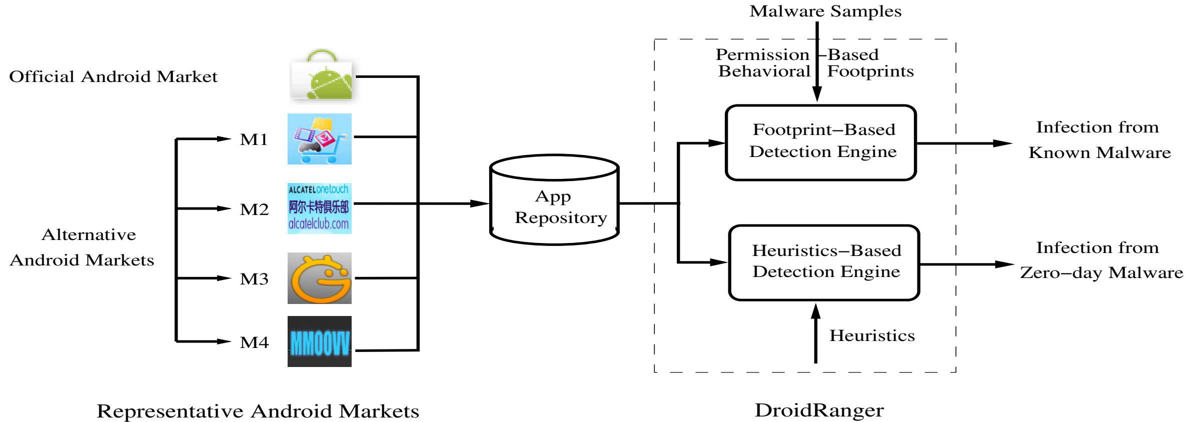

Figure 1: The overall architecture of DroidRanger

nizes them along with the app itself in a central database for efficient indexing and lookup.

After that, in order to detect potentially malicious apps, we take an approach with two different detection engines. The first detection engine (or footprint-based detection engine) is tailored to detect known malware. Specifically, each known malware will be first pre-processed or distilled into a so-called permission-based behavioral footprint. Each footprint essentially contains necessary Android permissions requested by the malware and succinctly summarizes the wrongdoings. These footprints are essentially the key to meet the scalability and efficiency requirements. The second detection engine (or heuristics-based detection engine) aims to uncover malware that has not been reported before. In particular, it recognizes suspicious behaviors from possibly malicious apps and detects certain Android features that may be misused. By doing so, we can identify suspicious apps and each will then be executed and monitored to verify whether it indeed exhibits any malicious behavior at runtime. If so, the app will then be manually confirmed and the associated behavioral footprint will be extracted and included in the first detection engine. In the following, we examine each detection engine in detail.

2.1 Detecting Known Android Malware

To meet the scalability and accuracy requirements, our first detection engine is divided into two steps: *permission-based filtering* and *behavioral footprint matching*. The first step aims to quickly exclude unrelated apps to obtain a more focused set of apps, which will then be used in the second step for detailed malware behavior matching.

Permission-based filtering As mentioned earlier, a significant challenge in designing DroidRanger is how to handle hundreds of thousands of apps in a scalable and efficient way. To address it, we observe that each Android app has to explicitly declare the permissions it requires in its manifest file. Thus, we could use essential permissions for the malware's functionalities to filter out unrelated apps (that do not declare these permissions). By doing so, we can possibly significantly reduce the number of apps that need to be processed in the second step. Therefore, our system can remain efficient even though our second step may involve expensive or time-consuming methods for deep scanning or thorough analysis.

Table 1: The number of remaining apps after filtering the repository with `RECEIVE_SMS`/`SEND_SMS` permissions

Permission	RECEIVE_SMS	SEND_SMS	RECEIVE_SMS & SEND_SMS
Apps	$5,214$	$8,235$	$3,204$
Percentage	2.85%	4.50%	1.75%

For illustration, we use two functionalities that are often misused by Android malware: sending SMS messages and monitoring SMS messages. For instance, the `Zsone` malware [24] is designed to send SMS messages to certain premium numbers, which will cause financial loss to the infected users. To hide these SMS messages from users, the malware also monitors received SMS messages and removes billing-related notifications. In Android, to use these two functionalities, an app needs to request `SEND_SMS` and `RECEIVE_SMS` permissions, respectively. Therefore, to detect this type of malware, we can first scan the app repository with these two permissions. In Table 1, we show the number of apps after we apply the filtering based on these two permissions in our collection: $5,214$ apps request the `RECEIVE_SMS` permission, which means 97.15% of apps do not need to be considered further for this type of malware. Similarly, filtering the repository with `SEND_SMS` removes 95.50% of apps from consideration. Combining these two permissions together, only 1.75% apps remain. Our experience (Section 3.1) indicates that permission-based filtering

is generally effective, even though the final results may depend on the popularity of the permissions of interest. In one case of our study, only 0.66% apps are needed to be processed in our second step.

We stress that in the selection of permissions for filtering, it is important to choose only essential ones. Including non-essential permissions will unnecessarily exclude apps, which leads to false negatives. For instance, considering a trojanized app with bot functionalities, we may select the permissions required for its command and control channel, but not those required by optional payloads (that could be easily changed from one variant to another). In our study, we have experienced one case with respect to the the `Pjapps` [7] malware. In particular, the malware requires the `INTERNET` permission to support the communication with the remote bot server and the `RECEIVE_SMS` permission to intercept or monitor incoming SMS messages. However, some variants may have the payload to add bookmarks to the built-in web browser (requiring `WRITE_HISTORY_BOOKMARKS`) and others do not. As such, we should only choose `INTERNET` and `RECEIVE_SMS` as the essential ones, not `WRITE_HISTORY_BOOKMARKS`.

Behavioral footprint matching Our first-step screening is effective in significantly reducing the number of apps for examination. However, it may leave thousands of apps, which means manual analysis is still not feasible. In our second step, we initially attempt to deploy and run off-the-shelf mobile anti-virus software [16]. However, the results are not satisfactory. In fact, an example run of a leading mobile anti-virus software missed about 23.52% of infected apps from our study (Section 3.1)[3].

As our solution, instead of taking the traditional approach of extracting content invariants as malware signatures, we choose to take a behavior-based approach. Specifically, we manually analyze and distill essential malware behaviors into their behavioral footprints. Note that because the first step has significantly reduced the number of apps for further processing, we can afford in the second step semantic-rich, but potentially time-consuming methods to express, summarize, and match malware behaviors. In particular, our behavioral footprinting scheme accommodates multiple dimensions to describe malware behaviors.

- First, the app-specific manifest file provides semantic-rich information about the app and the information is readily accessible for our purposes. For example, if an app needs to listen to system-wide broadcast messages, the broadcast receivers can be statically contained in the manifest file or dynamically registered in the code. We can then express the following rule to match any app that monitors incoming SMS messages: an app contains a receiver that listens to `android.provider.Telephony.SMS_RECEIVED`.
- Second, the app bytecode contains a wealth of semantic information, which can also be used in our behavioral footprints. Specifically, in DroidRanger, we choose to focus on those APIs defined in the Android framework. Particularly, we can express what APIs are called, and their sequences in a single rule. Moreover, by leveraging the call graph, we can associate API calls to a specific component in the rule. As an example, by extending the previous rule with a call to the `abortBroadcast` function, we can obtain a rule to capture any app that monitors incoming SMS messages and may potentially intercept received SMS messages. Furthermore, we can also utilize a data flow analysis algorithm [36] to detect function parameters with static or fixed inputs. This rule can be used to express malware behavior such as sending SMS messages to premium numbers.
- Third, we can also express malware behaviors based on the structural layout of the app. As an Android app is essentially a compressed archive, we can decompress it to reveal its internal tree structure and then correspondingly express rules such as what packages are used by the app, what kind of class hierarchies they have, and where a specific resource is located.

To illustrate, we again use the `Zsone` [24] example, an SMS Trojan that sends short messages to premium numbers and removes billing-related notification messages from respective service providers. We can accordingly generate the following behavioral footprints to describe `Zsone`: (1) An app contains a receiver that listens to `android.provider.Telephony.SMS_RECEIVED` and calls `abortBroadcast`; (2) An app sends SMS messages to certain specific premium numbers, including “10621900”, “106691819”, “10665123085”, “1066185829” and etc.; (3) An app intercepts SMS messages from certain numbers, including “10086”, “10000”, “10010”, and “1066134.” This behavioral footprint can then be efficiently applied to detect Zsone-infected apps in our collection. In practice, we found this scheme is rather effective and efficient, which leads to the discovery of 9 instances of Zsone-infected apps from the official Android Market in our collection (Section 3.1).

2.2 Detecting Unknown Android Malware

Our first detection engine is designed to capture known malware. Next, we present the second detection engine to uncover unknown malware. Similar to the first one, our second detection engine also works in two steps: *heuristics-based filtering* and *dynamic execution monitoring*. The first

[3] One possible reason is that the mobile anti-virus software uses content invariants as malware signatures, which could lead to a high false negative rate when code is changed or obfuscated.

step applies certain heuristics to identify potentially suspicious apps, which will then be confirmed in the second step.

Heuristics-based filtering Unlike the first detection engine that can have known malware samples as templates for detection, our second step has to rely on certain heuristics to uncover zero-day malware. Although we can potentially employ different types of heuristics, we in this work focus on certain Android features that may be misused to load new code, either Java binary code or native machine code. The reasoning is straightforward: dynamic loading of new (untrusted) code can always open the doors for misuse and it becomes hard to predict what kind of code will be executed next.

Accordingly, the first heuristic is related to the dynamic loading of Java binary code from a remote untrusted website. Typically, an Android app is self-contained in that its `classes.dex` file contains all the Dalvik bytecode for execution. However, the Dalvik VM provides the `DexClassLoader` class that can be used by an app to load classes from external files such as `.apk` or `.jar` (the standard archive of Java applications). If the dynamically-loaded code is fetched from a remote server, it becomes extremely hard to predict the app's behavior and thus indicates a potential threat to user privacy and security. Our measurement on the collected apps shows that `DexClassLoader` is used by 1055 apps (about 0.58%), and the vast majority of them are related to advertisement libraries, instead of the app itself. For example, one particular advertisement library, i.e., AdTOUCH [2], accounts for 40% of the uses of `DexClassLoader`. By considering the use of this feature by the app itself as suspicious, we further put the related apps under scrutiny, which essentially leads to the discovery of the zero-day Plankton spyware (Section 3.2).

Our second heuristic is about the dynamic loading of native code locally. Most of the Android apps are programmed in Java. However, for performance or compatibility reasons, about 4.52% Android apps we studied still use native code. Note that though each app may run with a separate UID, the OS (Linux) kernel's system call interface is directly exposed to the native code, making it possible for malicious apps to exploit vulnerabilities in OS kernel and privileged daemons to "root" the system. In Android, the default directory to store the native code is in the app-specific `lib/armeabi` subdirectory. As such, we consider it unusual if an app attempts to keep (or rather hide) native code in a directory other than the default one. Table 2 shows that about 4.52% of the total apps contain native code. In addition, 0.17% save native code in the `assets` directory and 0.11% put native code in the `res/raw` directory. This heuristic leads to the discovery of the zero-day `DroidKungFu` malware (Section 3.2).

Dynamic execution monitoring Our heuristics effectively identify new untrusted code running in the app. Our next step deploys a dynamic execution monitor to inspect its runtime behaviors, particularly those triggered by the new code. In particular, for the dynamically loaded Java code, our dynamic monitor records any calls to the Android framework APIs (particularly these related to Android permissions) and their arguments. The arguments can provide rich semantic information about an app's behavior. For example, a call to the `SmsManger.sendTextMessage` function reveals both the destination number (is it a premium number?) and the content of the SMS message body (is it a message to subscribe a service that will cost money?). For the dynamically-loaded native code, our dynamic monitor collects system calls made by the native code. The collection is possible with a kernel module that hooks the system call table in the (Linux) kernel. In our prototype, we do not log all the system calls but instead focus on those used by existing Android root exploits and/or made with the root privilege (with the assumption that malware will use them to subvert the system integrity after gaining the root privilege). For instance, we are interested in the `sys_mount` system call because it can be used to remount the (normally read-only) Android system partition for modification.

Table 2: The number (and percentage of) apps with native code not in the default location

Apps with Native Code	Native Code in Assets Directory	Native Code in Res Directory
8272 (4.52%)	313 (0.17%)	195 (0.11%)

After collecting the logs, we then analyze them to look for signs of suspicious runtime behaviors (e.g., sending SMS messages to premium numbers, executing certain system calls with root privilege and etc.). If so, we will further manually validate whether the app is indeed a zero-day malware. If yes, we will then extract the corresponding behavioral footprint and include it in the first detection engine to detect other samples infected by this malware.

3 Evaluation

In this section, we present our evaluation results by applying our system to the collected apps from existing Android Markets. Specifically, we crawled five representative marketplaces, including the official Android Market (maintained by Google), eoeMarket [10] , alcatelclub [3], gfan [13], and mmoovv [17] [4]. The collection was made in a two-month period during May and June 2011 and the number of apps from each marketplace is shown in Table 3. In total, we collected $204,040$ free apps. Among them, $\sim 75\%$ of apps ($153,002$) are collected from the official Android Market and the rest 25% ($51,038$) come from the four al-

[4]For simplicity, we use M1, M2, M3, and M4 to denote these four alternative marketplaces in the rest of this paper.

Table 3: Statistics of collected apps from existing Android Markets

	Official Android Market	Alternative Android Markets			
		M1	M2	M3	M4
Number of Apps	153,002 (74.98%)	17,229 (8.44%)	14,943 (7.33%)	10,385 (5.09%)	8,481 (4.16%)
Total Apps	153,002 (74.98%)	51,038 (25.02%)			
	204,040				

Table 4: Ten known malware families used in this study (†: first reported in the official Android Market)

Malware	Reported Time	MD5	Comments
Geinimi [20]	12/2010	fa6e905d4be3b09541106e1cb4052943 b391d30ec53a08c20552203f58ca712c	Trojan with bot-like capabilities
ADRD [21]	02/2011	b3b38812cc01aa24df68bac4de517e23 a3fec4efca307adab4888d20d786a2a4	Trojan with bot-like capabilities
Pjapps [7]	02/2011	cec4c470a6dbac597064c6f566101028 c05d4ff1a80f18ba9d8a86afd88bc05d	Trojan with bot-like capabilities
Bgserv [6]	03/2011	ea97576befac2b142144ce30c2326ed6 1d696b87e665498b878bf92ce25440db	Trojan with bot-like capabilities
DroidDream† [19]	03/2011	63f26345ba76ef5e033ef6e5ccecd30d 763a1ab4e4a21373a24a921227a6f7a4	Root exploits with Exploid, Rageagainstthecage
zHash† [23]	03/2011	f3b6eb64adef922096ff8b8618f3c1a9 5beae5543a3f085080c26de48a811da6	Root exploit with Exploid
BaseBridge [5]	05/2011	b2d359952bce1823d29e182dacac159c d9814e6ec16be5b6174b518c186647b5	Root exploit with Rageagainstthecage
DroidDreamLight† [25]	05/2011	3ae28cbf5a92e8e7a06db4b9ab0a55ab 5e4fd0a680ffc8d84d16b8c0e5404140	Trojan with information stealing capabilities
Zsone† [24]	05/2011	d204007a13994cb1ad68d8d2b89cc9a8 a673481ff15028e05f6fe778a773e0f9	Trojan that sends premium-rate SMS messages
jSMSHider [22]	06/2011	a3c0aacb35c86b4468e85bfb9e226955 389b416fb0f505d661716b8da02f92a2	Trojan that targets custom firmware devices

ternative markets (with each having about tens of thousands of apps). Due to the fact that an app may be included in multiple markets, there are total $182,823$ distinct apps[5].

3.1 Detecting New Samples of Known Malware

To evaluate the effectiveness of our first detection engine, we obtained and analyzed known Android malware samples from 10 different families. Table 4 lists these ten malware families, which represent most Android malware reported in the first half of year 2011. For each family, we then extract the corresponding behavioral footprints from two samples (their MD5 values are also included in the table). Among these malware, three of them embed at least one of the following root exploits, i.e., Rageagainstthecage [18], Zimperlich [26], or Exploid [11]. Specifically, Rageagainstthecage takes advantage of a vulnerability in adbd, a privileged daemon running as root; Zimperlich exploits a similar bug but in the privileged zygote daemon; while Exploid makes use of the vulnerable *udev* (*init*) daemon (CVE-2009-1185) in Android. Among these exploits, Exploid needs the triggering of certain udev events such as turning on or off the WIFI interface, which requires the CHANGE_WIFI_STATE permission. Accordingly, our detection engine recognizes such need and uses it to prune the collected apps for Exploid-specific root exploits.

Effectiveness of permission-based filtering Based on the known Android malware samples, our first step is to extract the essential permissions required by the malware to perform its functionalities. In Table 5, we list the essential permissions for each malware family and also report the number of apps that survived the permission-based filter. From the table, the proposed permission-based filtering is rather effective for most of these malware families. Specifically, eight of them have less than 6% apps left after applying the essential permissions. Among the two exceptions, DroidDreamLight requires INTERNET and READ_PHONE_STATE permissions which are common in existing apps. In this particular case, we

[5] In this study, we consider apps with different SHA1 values are distinct.

Table 5: Essential permissions of 10 known malware families

Malware	Essential Permissions	Apps
`ADRD`	INTERNET, ACCESS_NETWORK_STATE RECEIVE_BOOT_COMPLETED	$10,379$ (5.68%)
`Bgserv`	INTERNET, RECEIVE_SMS, SEND_SMS	$2,880$ (1.58%)
`DroidDream`	CHANGE_WIFI_STATE	$4,096$ (2.24%)
`DroidDreamLight`	INTERNET, READ_PHONE_STATE	$71,095$ (38.89%)
`Geinimi`	INTERNET, SEND_SMS	$7,620$ (4.17%)
`jSMSHider`	INSTALL_PACKAGES	$1,210$ (0.66%)
`BaseBridge`	NATIVE CODE	$8,272$ (4.52%)
`Pjapps`	INTERNET, RECEIVE_SMS	$4,637$ (2.54%)
`Zsone`	RECEIVE_SMS, SEND_SMS	$3,204$ (1.75%)
`zHash`	CHANGE_WIFI_STATE	$4,096$ (2.24%)

Table 6: The number of infected apps by 10 known malware families on 5 studied marketplaces

Malware	Official Android Market	Alternative Marketplaces				Total	Distinct
		M1	M2	M3	M4		
`ADRD`	0	1	1	4	3	9	8
`BaseBridge`	0	2	2	0	2	6	4
`Bgserv`	0	0	0	0	1	1	1
`DroidDream`	0	6	6	0	0	12	6
`DroidDreamLight`	12	0	0	0	0	12	12
`Geinimi`	0	26	26	2	10	64	37
`jSMSHider`	0	3	3	0	6	12	9
`Pjapps`	0	12	9	14	8	43	31
`zHash`	0	1	1	0	1	3	2
`Zsone`	9	0	0	0	0	9	9
Total	21	51	48	20	31	171	119

also observe it needs to register for a broadcast receiver for `android.intent.action.PHONE_STATE`. Therefore, we also leverage it as a pre-condition for our behavioral footprint matching, which significantly reduces to 0.64% of apps (or 1175) for the second step. The another exception, i.e., `BaseBridge`, does not require any permission, but the fact that contains the native code to launch the `Rageagainstthecage` immediately reduces to the 4.52% of apps (with native code) in our collection.

It is worth mentioning the essential permission of a particular malware, i.e., `jSMSHider`. It is special in that it requires `INSTALL_PACKAGES`, a `signatureOrSystem` permission. This type of permission by definition can only be granted to an app either signed by the same signing key as the system firmware or installed in the privileged `/system/app` directory. In other words, it will never be granted to third-party apps. `jSMSHider` is unique by targeting the popular third-party custom firmware whose signing key is (inappropriately) publicly accessible. In other words, by signing the infected app with it, `jSMSHider` is able to gain this `signatureOrSystem` permission to install a package. As a result, we selected `INSTALL_PACKAGES` as the essential permission for the `jSMSHider` malware.

Effectiveness of behavioral footprint matching After pruning the apps with essential permissions, our second step examines the resulting apps for ones that match known malware behaviors. Our experiments show that the detection of known malware (of 10 families) among these $204,040$ apps only took about four and a half hours to complete. Our scanning results are shown in Table 6. In particular, we detected 171 malicious apps in our collection with 21 of them from the official Android Market and 150 from the other four alternative marketplaces. Similarly, due to the fact that one sample may be submitted to multiple marketplaces, 119 of these 171 infected apps are unique.

Based on the detection results in Table 6, it is evident that the malware infection in alternative marketplaces is more serious than the infection in official Android Market. Specifically, the number (150) of infected apps in these four alternative marketplaces is more than 7 times of that in the official marketplace (21) even though the total number of apps we collected from the former is only one third of the latter. Moreover, we also found that the `Geinimi` malware, which was publicly reported in January, 2011, is still

Table 7: The missed known malware families by Lookout Security & Antivirus software (T, D, and M represent the total, detected, and missed number of samples, respectively.)

	ADRD			Bgserv			jSMSHider			BaseBridge			Pjapps		
	T	D	M	T	D	M	T	D	M	T	D	M	T	D	M
version 6.3 (r8472)	8	3	5	1	0	1	9	6	3	4	1	3	31	15	16
version 6.11 (26cf47e)	8	3	5	1	0	1	9	9	0	4	4	0	31	31	0

Table 8: Two zero-day malware families detected by DroidRanger

Malware	Official Android Market	Alternative Markets				Total	Distinct
		M1	M2	M3	M4		
`Plankton`	11	0	0	0	0	11	11
`DroidKungFu`	0	9	10	1	9	29	18
Total	11	9	10	1	9	40	29

spreading in alternative marketplaces. This is in sharp contrast to the timely removal of malware from the official Android Market.

In this study, we also attempted to measure the effectiveness of existing mobile anti-virus software. Especially, we downloaded and installed the free version of Lookout Security & Antivirus, one of the leading mobile security software from the official Android Market. We first tested it in the first week of August 2011 and then tested it again in the first week of November[6] to scan the samples reported by our system. In our experiments, we installed those malware samples one by one on a Nexus One phone (running Android version 2.3.3) with the Lookout Security & Antivirus software. The scanning results are shown in Table 7.

From the table, none of these two versions can detect all the samples. In particular, among these 119 distinct malware samples, the earlier version (6.3) reports 91 as malware and 28 sample as safe, leading to a high false negative rate 23.52%. To verify that these 28 samples are indeed malicious, we have manually examined these samples and confirmed each single of them. The recent version (6.11) performs better by reporting 113 as malware and 6 samples as safe with a reduced false negative rate 5.04%. We believe the improvement of scanning results is due to the active update of malware signatures.

False negative measurement Our previous experiments demonstrate the effectiveness of our scheme (especially in terms of false positives) for known malware detection. Next, we aim to measure the false negatives of our system. To do that, we first download (in the first week of August, 2011) the malware samples from the *contagio dump* [8], which seems to be the only publicly available repository of mobile malware samples. Within the downloaded data set, there are 27 samples falling in the 10 known families which we have the related behavioral footprints. Because our scheme requires known malware samples for behavioral footprint extraction, we take cautions in eliminating those duplicate samples in the *contagio dump* with the same SHA1 values used in footprint extraction. As a result, there are 24 distinct samples in total for our measurement. Though the data set is not sufficient large, this seems to be the only available source for our study.

Based on these 24 samples, our system detected 23 of them, implying a false negative rate of 4.2%. A detailed analysis of the missing sample shows that it is the payload (*com.android.providers.downloadsmanager*) of the `DroidDream` malware, not the malware itself. This payload is also a standalone app and we do not have the corresponding behavioral footprint, which explains why our system missed it. Meanwhile, from this experiment, we also find that the malware repository has miscategorized one particular sample *PMSW_V1.8_.apk* (md5: $5895bcd066abf6100a37a25c0c1290a5$). The sample, considered to be a `DroidDream` malware, is actually an `ADRD` malware.

[6]The version numbers are version 6.3 (release 8472) and version 6.11 (26cf47e) respectively.

3.2 Detecting Zero-day Malware

Next, we present the evaluation results for zero-day malware detection. As mentioned earlier, we have developed heuristics to identify suspicious apps that dynamically load untrusted code for execution. These heuristics effectively lead to the discovery of two zero-day malware: `Plankton` and `DroidKungFu`. Table 8 shows the number of detected zero-day malware samples from each marketplace. In this section, we present in more detail on how we discovered these two malware and what are their malicious behaviors.

Plankton In the second detection engine, our first heuristic is to capture dynamic loading of untrusted code from remote websites. With that, we perform a query on the collected apps and found $1,055$ apps that invoked the

```
public enum Commands
{
    ...
    static
    {
        ACTIVATION          = new Commands("ACTIVATION", 1, "Activation", "/activate");
        HOMEPAGE            = new Commands("HOMEPAGE", 2, "Homepage", "/homepage");
        COMMANDS_STATUS     = new Commands("COMMANDS_STATUS", 3, "CommandsStatus", "/commandstatus");
        BOOKMARKS           = new Commands("BOOKMARKS", 4, "Bookmarks", "/bookmarks");
        SHORTCUTS           = new Commands("SHORTCUTS", 5, "Shortcuts", "/shortcuts");
        HISTORY             = new Commands("HISTORY", 6, "History", "/history");
        TERMINATE           = new Commands("TERMINATE", 7, "Terminate", "/terminate");
        STATUS              = new Commands("STATUS", 8, "Status", "/status");
        DUMP_LOG            = new Commands("DUMP_LOG", 9, "DumpLog", "/dumplog");
        UNEXP_EXCEPTION     = new Commands("UNEXP_EXCEPTION", 10, "UnexpectedException", "/unexpectedexception");
        UPGRADE             = new Commands("UPGRADE", 11, "Upgrade", "/installation");
        INSTALLATION        = new Commands("INSTALLATION", 12, "Installation", "/installation");

        arrayOfCommands[0]  = COMMANDS;
        arrayOfCommands[1]  = ACTIVATION;
        arrayOfCommands[2]  = HOMEPAGE;
        arrayOfCommands[3]  = COMMANDS_STATUS;
        arrayOfCommands[4]  = BOOKMARKS;
        arrayOfCommands[5]  = SHORTCUTS;
        arrayOfCommands[6]  = HISTORY;
        arrayOfCommands[7]  = TERMINATE;
        arrayOfCommands[8]  = STATUS;
        arrayOfCommands[9]  = DUMP_LOG;
        arrayOfCommands[10] = UNEXPECTED_EXCEPTION;
        arrayOfCommands[11] = UPGRADE;
        arrayOfCommands[12] = INSTALLATION;
    }
}
```

Figure 2: The list of commands supported in `Plankton`

`DexClassLoader` support for Java class loading. A further inspection shows that most of such uses are actually by advertisement libraries. After a simple white-listing of advertisement libraries, we run the remaining 240 apps with a dynamic execution monitor (Section 2.2).

With a dynamic monitor, we can not only intercept the code-loading behavior, but also obtain the *jar/apk* files that are being actually loaded. In this particular case, when running a suspect app named `Angry Birds Cheater` (`com.crazyapps.angry.birds.cheater.trainer.helper`) in our dynamic monitor, our logs show the attempt to load one jar file named *"plankton_v0.0.4.jar,"* which is downloaded from a remote website. A more in-depth analysis shows that before downloading this jar file, the app transports the list of permissions granted to the app over to a remote server. This would presumably allow the remote server to customize the dynamically loaded binary based on the granted app's permissions. For the downloaded *plankton_v0.0.4.jar* file, it turns out that it contains a number of bot-related functionalities or commands that can be remotely invoked. In Figure 2, we show the list of commands supported by this spyware. Specifically, the `/bookmarks` command manipulates the browser bookmarks, and the `/history` command leaks the user browsing history. It also has a `/dumplog` command that can dump the runtime logs from `logcat`. Earlier reports [15] show that highly sensitive private information may exist as plain text in the runtime log.

After we identified the Plankton malware, we generated its behavioral footprint and applied it to detect other incidents in our collection. As a result, we detect 10 more `Plankton` samples; all of them could be downloaded at that time from the official Android Market. We then reported these 11 apps to Google, which promptly removed them from the official Android Market on the same day.

DroidKungFu Our second heuristic aims to detect suspicious apps that load native code in an unusual way (e.g., from non-standard directories). In fact, among 8,272 apps that contain native code, 508 of them keep native code in non-standard directories. As such, we monitored the runtime behaviors of these apps with our dynamic monitor. When analyzing the log, we find some particular apps that attempted to remount the system partition (with the `sys_mount` syscall) to make it writable. This is highly suspicious because remounting system partition can only be executed by users with the root privilege. For third-party apps, this almost certainly means that the app has successfully launched a root exploit.

Therefore, we further analyze these apps, which lead to the discovery of `DroidKungFu` malware. The malware contains both `Rageagainstthecage` and `Exploid` root exploits in an encrypted form. When `DroidKungFu` runs, it will first decrypt and launch the root exploits. If successful, the malware will essentially elevate its privilege to root so that it can arbitrarily access and modify any resources on the phone, including the installation and removal of another app. Our study shows that one particular app installed by `DroidKungFu` pretends to be the legitimate Google Search

Table 9: The total number of malware samples detected by DroidRanger

Malware	Official Android Market	Alternative Markets				Total	Distinct
		M1	M2	M3	M4		
Total (Known)	21	51	48	20	31	171	119
Total (Zero-day)	11	9	10	1	9	40	29
Total	32 (0.02%)	60 (0.35%)	58 (0.39%)	21 (0.20%)	40 (0.47%)	211	148

app by showing an identical icon. This app actually acts as a bot client that will connect to a remote server to retrieve and execute commands.

3.3 Summary of Detected Malware

Overall, our system detected 211 malicious or infected apps among the collected $204,040$ apps from five studied marketplaces. The detailed breakdown is shown in Table 9: among these 211 apps, 32 of them are from the official Android Market ($\sim 0.02\%$ infection rate) and 179 come from the other four alternative marketplaces (0.20%-0.47% infection rates). Due to the relative lack of regulation in alternative marketplaces, their infection rate is more than an order of magnitude higher than the official Android Market. Considering the recent trend in observing an unprecedent growth of mobile malware, we believe a rigorous vetting process needs to be in place for both official and unofficial marketplaces.

4 Discussion

Our scanning results show a relatively low malware infection rate on studied Android Markets (0.02% on official Android Market and 0.20% to 0.47% on other alternative marketplaces). Such infection rate is certainly less than that of malicious web contents reported earlier in [46]. However, due to the centralized role they played in the smartphone app ecosystem, such infection rate, though low, can still compromise a tremendous number of smartphones and cause lots of damages. Using the `DroidDream` malware as the example, it is reported that this particular malware infected more than $260,000$ users within 48 hours before Google removed them from the Android Market [1]. Also, due to the relative lack of regulation in alternative marketplaces, even though a malware may be removed from the official Android Market, it can continue to exist in alternative marketplaces. With the same `DroidDream` example, we found its presence in unofficial marketplaces three months later after they were removed from the official Android Market. Moreover, we notice that some malware have embedded the root exploits which allow them to obtain full access on compromised devices. In fact, among the 10 known malware families, four of them, i.e., `BaseBridge`, `DroidDream`, `DroidKungFu`, and `zHash`, have at least one root exploit. Last but not least, zero-day malware exist in both official and unofficial marketplaces and they may be able to bypass most existing, if not all, up-to-date mobile anti-virus software.

With that, there is a clear need to instantiate a rigorous vetting or policing process in both official and unofficial Android Markets, which is currently lacking. In the same spirit, we argue the current model of passively waiting for the feedback or ratings from users and the community may not work as smoothly as we expect. In fact, to foster a hygienic mobile app ecosystem, there is a need for a vetting process that includes DroidRanger-like systems to examine uploaded apps and better protect mobile users.

From another perspective, our current prototype is still limited in the coverage of apps and marketplaces. For example, though the collected pool of apps may be useful to demonstrate the effectiveness of our current prototype, they are all free apps. A recent report from Androlib [9] shows that 36.2% of existing apps are paid ones. We have the reason to believe that paid apps could provide unique differences that may warrant necessary re-design or adaption of our system. Also, our current study only focuses on five different Android Markets. Though our findings may not be directly applicable to other marketplaces including iOS-related app stores, the principles or key methodologies used in this study can still be applied.

Finally, we point out that our current study only explored two basic heuristics to uncover zero-day malware. There exist many other heuristics that could be equally effective. For example, a heuristic can be developed to capture apps that involve background sending of unauthorized SMS messages to certain premium-rated numbers. Another heuristic can also be designed to detect bot-like behavior that is remotely controlled by SMS messages. The exploration of these heuristics to capture additional zero-day malware remains an interesting future work.

5 Related Work

Smartphone platform security The first area of related work includes recent systems [27, 28, 31, 32, 34, 35,

36, 37, 38, 39, 40, 42, 44, 47, 48, 54] that either reveal potential security risks or improve the overall security on smartphone platforms. For example, TaintDroid [35] and PiOS [34] are two representative systems that demonstrate potential privacy threats from third-party apps in both Android and iOS platforms. Comdroid [31] and the subsequent work [39] analyze the vulnerability in inter-app communication in Android apps and find a number of exploitable vulnerabilities. In particular, a notion of permission re-delegation has been proposed to describe associated risks when public interfaces of permission-guarded operations are exposed. Woodpecker [41] similarly exposes capability leaks on stock Android phones by analyzing preloaded apps in the phone firmware. Stowaway [38] examines 940 apps and finds that about one-third apps are over-privileged, i.e., they are requesting more permissions than they need. Our work has a different focus on detecting malicious apps in current Android Markets.

On the defensive side, ScanDroid [40] extracts app-specific security specifications and applies data flow analysis for their consistency in the app code. Kirin [37] aims to block the installation of potential unsafe apps if they exhibit certain dangerous permission combination. Apex [47], MockDroid [28], TISSA [54] and AppFence [42] revise the current Android framework or runtime to better provide fine-grained controls of resources accessed by third-party untrusted apps. Saint [48] is proposed to protect the exposed interfaces (accessible to other apps) according to security policies defined by the app authors. Quire [32] addresses the confused deputy attacks in Android by proposing a set of extensions. L4Android [44] and Cells [27] isolate smartphone OSes for different usage environments in different virtual machines (VMs).

Among the most related, Enck *et al.* [36] conducts a systematic study of $1,100$ top free Android apps from the official Android Market to better understand generic security characteristics (e.g., pervasive use/misuse of personal identifiers and deep penetration of advertising and analytics networks). The study itself does not lead to any malware detection. In comparison, our study has an exclusive focus on detecting malware infection in both official and unofficial marketplaces. Based on the collected $204,040$ apps from five different marketplaces, our system detects 211 infected apps and uncovers two zero-day malware, including one from the official Android Market.

Malware detection on mobile devices The second area of related work includes systems [29, 30, 33, 43, 45, 49, 52] to detect malware on mobile devices. For example, Bose *et al.* [29] presents a framework to detect malware on mobile handsets by observing the logical order of an app's actions and matching with "spatial-temporal" representation of known malware behaviors. pBMDS [52] uses a probabilistic-based approach that correlates a user's input with system call events to detect anomalous behaviors in mobile phones. Kim *et al.* [43] focuses on energy-greeding malware on mobile devices. VirusMeter [45] proposes to detect malware based on abnormal power consumption caused by malware. Crowdroid [30] collects system calls of running apps on mobile devices and applies clustering algorithms to differentiate between benign and malicious apps. Dixon *et al.* [33] proposes a system to detect malicious code by correlating power consumption pattern with the user's location. Felt *et al.* [49] surveys the current situation of mobile malware on three popular smartphone platforms (iOS, Android and Symbian). DroidMOSS [53] detects repackaged apps in third-party Android Markets. DroidRanger is different from these systems in not detecting mobile malware on mobile devices (under resource constraints such as limited battery or CPU). Instead, it performs offline analysis to detect malware in current Android Markets. Accordingly, it needs to address different challenges by accommodating a large number of apps (Section 2).

Other systematic security studies The third area of related work includes prior efforts [46, 50, 51] to systematically study malicious web contents. For example, HoneyMonkey [51] identifies malicious websites by deploying and running client-side honeypots with different levels of vulnerable web browsers. Moshchuk *et al.* [46] performs a large-scale, systematic study of malicious objects on the web, which crawls a large number of webpages and executables from Internet and uses the off-the-shelf anti-spyware software to identify malicious ones. Provos *et al.* [50] collects billions of URLs in a period of ten months and finds over 3 million malicious URLs which can launch drive-by download attacks. Instead of detecting malicious web contents, our system focuses on detecting malicious apps, which pose different requirements and challenges in the system design. From another perspective, our system does share a similar spirit by involving a large-scale crawling of apps from existing marketplaces for malicious app detection.

6 Conclusion

In the paper, we have presented a systematic study to detect malicious apps on both official and unofficial Android Markets. Our study involves a comprehensive crawling of five representative Android Markets in May and June 2011, which results in a collection of $204,040$ Android apps. To scalably and efficiently detect potential infection from both known and unknown malware, we have accordingly proposed two different schemes, i.e., permission-based behavioral footprinting and heuristics-based filtering. We have implemented both schemes in DroidRanger and the evaluation results of successfully detecting 211 malicious apps and uncovering two zero-day malware in both official and

unofficial marketplaces demonstrate the feasibility and effectiveness of our approach. From another perspective, our results also call for the need of a rigorous vetting process to better police both official and unofficial marketplaces.

Acknowledgements The authors would like to thank the anonymous reviewers for their insightful comments that helped improve the presentation of this paper. Special thanks go to Simon Zou, Jack Chiang, and Yang Cao at NQ Mobile for their constructive suggestions and feedbacks. This work was supported in part by the US Army Research Office (ARO) under grant W911NF-08-1-0105 managed by NCSU Secure Open Systems Initiative (SOSI) and the US National Science Foundation (NSF) under Grants 0855297, 0855036, 0910767, and 0952640. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the ARO and the NSF.

References

[1] 260,000 Android users infected with malware. http://www.infosecurity-magazine.com/view/16526/260000-android-users-infected-with-malware/.

[2] AdTOUCH. http://www.adtouchnetwork.com/adtouch/sdk/SDK.html.

[3] Alcatelclub. http://www.alcatelclub.com/.

[4] Amazon Appstore. http://www.amazon.com/mobile-apps/b?ie=UTF8&node=2350149011.

[5] Android.Basebridge Technical Details. http://www.symantec.com/security_response/writeup.jsp?docid=2011-060915-4938-99&tabid=2.

[6] Android.Bgserv Found on Fake Google Security Patch. http://www.symantec.com/connect/blogs/androidbgserv-found-fake-google-security-patch.

[7] Android.Pjapps Technical Details. http://www.symantec.com/security_response/writeup.jsp?docid=2011-022303-3344-99&tabid=2.

[8] Contagio mobile malware mini dump. http://contagiominidump.blogspot.com/.

[9] Distribution of Free and Paid Apps in Android Market. http://www.androlib.com/appstatsfreepaid.aspx.

[10] eoeMarket. http://www.eoemarket.com/.

[11] Exploid Root Exploit. http://c-skills.blogspot.com/2010/07/exploid-works-on-droid-x.html.

[12] Gartner Says 428 Million Mobile Communication Devices Sold Worldwide in First Quarter 2011. http://www.gartner.com/it/page.jsp?id=1689814.

[13] Gfan. http://www.gfan.com/.

[14] Google I/O 2011. http://www.google.com/events/io/2011/.

[15] HTC IME Sends All Info to Logcat. http://code.google.com/p/cyanogenmod/issues/detail?id=445.

[16] Lookout Mobile Security. https://www.mylookout.com/.

[17] Mmoovv. http://android.mmoovv.com/web/index.html.

[18] Rageagainstthecage Root Exploit. http://c-skills.blogspot.com/2010/08/droid2.html.

[19] Security Alert: DroidDream Malware Found in Official Android Market. http://blog.mylookout.com/2011/03/security-alert-malware-found-in-official-android-market-droiddream/.

[20] Security Alert: Geinimi, Sophisticated New Android Trojan Found in Wild. http://blog.mylookout.com/2010/12/geinimi_trojan/.

[21] Security Alert: HongTouTou, New Android Trojan, Found in China. http://blog.mylookout.com/2011/02/security-alert-hongtoutou-new-android-trojan-found-in-china/.

[22] Security Alert: Malware Found Targeting Custom ROMs (jSMSHider). http://blog.mylookout.com/2011/06/security-alert-malware-found-targeting-custom-roms-jsmshider/.

[23] Security Alert: zHash, A Binary that can Root Android Phones, Found in Chinese App Markets and Android Market. http://blog.mylookout.com/2011/03/security-alert-zhash-a-binary-that-can-root-android-phones-found-in-chinese-app-markets-and-android-market/.

[24] Security Alert: Zsone Trojan found in Android Market. http://blog.mylookout.com/2011/05/security-alert-zsone-trojan-found-in-android-market/.

[25] Update: Security Alert: DroidDreamLight, New Malware from the Developers of DroidDream. http://blog.mylookout.com/2011/05/security-alert-droiddreamlight-new-malware-from-the-developers-of-droiddream/.

[26] Zimperlich sources. http://c-skills.blogspot.com/2011/02/zimperlich-sources.html.

[27] J. Andrus, C. Dall, A. Van't Hof, O. Laadan, and J. Nieh. Cells: A Virtual Mobile Smartphone Architecture. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, SOSP '11, 2011.

[28] A. R. Beresford, A. Rice, N. Skehin, and R. Sohan. MockDroid: Trading Privacy for Application Functionality on Smartphones. In *Proceedings of the 12th International Workshop on Mobile Computing System and Applications*, HotMobile '11, 2011.

[29] A. Bose, X. Hu, K. G. Shin, and T. Park. Behavioral Detection of Malware on Mobile Handsets. In *Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, 2008.

[30] I. Burguera, U. Zurutuza, and S. Nadjm-Tehrani. Crowdroid: Behavior-Based Malware Detection System for Android. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM'11, 2011.

[31] E. Chin, A. P. Felt, K. Greenwood, and D. Wagner. Analyzing Inter-Application Communication in Android. In *Proceedings of the 9th Annual Symposium on Network and Distributed System Security*, MobiSys 2011, 2011.

[32] M. Dietz, S. Shekhar, Y. Pisetsky, A. Shu, and D. S. Wallach. QUIRE: Lightweight Provenance for Smart Phone Operating Systems. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[33] B. Dixon, Y. Jiang, A. Jaiantilal, and S. Mishra. Location Based Power Analysis to Detect Malicious Code in Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM'11, 2011.

[34] M. Egele, C. Kruegel, E. Kirda, and G. Vigna. PiOS: Detecting Privacy Leaks in iOS Applications. In *Proceedings of the 18th Annual Symposium on Network and Distributed System Security*, NDSS '11, 2011.

[35] W. Enck, P. Gilbert, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth. TaintDroid: An Information-Flow Tracking System for Realtime Privacy Monitoring on Smartphones. In *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation*, USENIX OSDI '10, 2010.

[36] W. Enck, D. Octeau, P. McDaniel, and S. Chaudhuri. A Study of Android Application Security. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[37] W. Enck, M. Ongtang, and P. McDaniel. On Lightweight Mobile Phone Application Certification. In *Proceedings of the 16th ACM Conference on Computer and Communications Security*, CCS '09, 2009.

[38] A. P. Felt, E. Chin, S. Hanna, D. Song, and D. Wagner. Android Permissions Demystied. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[39] A. P. Felt, H. J. Wang, A. Moshchuk, S. Hanna, and E. Chin. Permission Re-Delegation: Attacks and Defenses. In *Proceedings of the 20th USENIX Security Symposium*, USENIX Security '11, 2011.

[40] A. Fuchs, A. Chaudhuri, and J. Foster. SCanDroid: Automated Security Certification of Android Applications. http://www.cs.umd.edu/ avik/projects/scandroidascaa.

[41] M. Grace, Y. Zhou, Z. Wang, and X. Jiang. Systematic Detection of Capability Leaks in Stock Android Smartphones. In *Proceedings of the 19th Annual Symposium on Network and Distributed System Security*, NDSS '12, 2012.

[42] P. Hornyack, S. Han, J. Jung, S. Schechter, and D. Wetherall. These Aren't the Droids You're Looking For: Retrofitting Android to Protect Data from Imperious Applications. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, 2011.

[43] H. Kim, J. Smith, and K. G. Shin. Detecting Energy-Greedy Anomalies and Mobile Malware Variants. In *Proceeding of the 6th International Conference on Mobile Systems, Applications, and Services*, MobiSys '08, 2008.

[44] M. Lange, S. Liebergeld, A. Lackorzynski, A. Warg, and M. Peter. L4Android: A Generic Operating System Framework for Secure Smartphones. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM'11, 2011.

[45] L. Liu, G. Yan, X. Zhang, and S. Chen. VirusMeter: Preventing Your Cellphone from Spies. In *Proceedings of the 12th International Symposium on Recent Advances in Intrusion Detection*, RAID'09, 2009.

[46] A. Moshchuk, T. Bragin, S. D. Gribble, and H. M. Levy. A Crawler-based Study of Spyware on the Web. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security*, NDSS '06, 2006.

[47] M. Nauman, S. Khan, and X. Zhang. Apex: Extending Android Permission Model and Enforcement with User-Defined Runtime Constraints. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, ASIACCS '10, 2010.

[48] M. Ongtang, S. McLaughlin, W. Enck, and P. McDaniel. Semantically Rich Application-Centric Security in Android. In *Proceedings of the 2009 Annual Computer Security Applications Conference*, ACSAC '09, 2009.

[49] A. Porter Felt, M. Finifter, E. Chin, S. Hanna, and D. Wagner. A Survey of Mobile Malware In The Wild. In *Proceedings of the 1st Workshop on Security and Privacy in Smartphones and Mobile Devices*, CCS-SPSM'11, 2011.

[50] N. Provo, P. Mavrommatis, M. A. Rajab, and F. Monrose. All Your iFRAMEs Point to Us. In *Proceedings of the 17th USENIX Security Symposium*, USENIX Security '08, 2008.

[51] Y.-M. Wang, D. Beck, X. Jiang, R. Roussev, C. Verbowski, S. Chen, and S. King. Automated Web Patrol with Strider HoneyMonkeys: Finding Web Sites That Exploit Browser Vulnerabilities. In *Proceedings of the 13th Annual Symposium on Network and Distributed System Security*, NDSS '06, 2006.

[52] L. Xie, X. Zhang, J.-P. Seifert, and S. Zhu. pBMDS: A Behavior-based Malware Detection System for Cellphone Devices. In *Proceedings of the 3rd ACM conference on Wireless Network Security*, WiSec '10, 2010.

[53] W. Zhou, Y. Zhou, X. Jiang, and P. Ning. DroidMOSS: Detecting Repackaged Smartphone Applications in Third-Party Android Marketplaces. In *Proceedings of the 2nd ACM Conference on Data and Application Security and Privacy*, CODASPY'12, 2012.

[54] Y. Zhou, X. Zhang, X. Jiang, and V. W. Freeh. Taming Information-Stealing Smartphone Applications (on Android). In *Proceeding of the 4th International Conference on Trust and Trustworthy Computing*, TRUST '11, 2011.