Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale

Kai Chen, *Chinese Academy of Sciences and Indiana University;* Peng Wang, Yeonjoon Lee, Xiaofeng Wang, and Nan Zhang, *Indiana University;* Heqing Huang, *The Pennsylvania State University;* Wei Zou, *Chinese Academy of Sciences;* Peng Liu, *The Pennsylvania State University*

https://www.usenix.org/conference/usenixsecurity15/technical-sessions/presentation/chen-kai

This paper is included in the Proceedings of the 24th USENIX Security Symposium

August 12–14, 2015 • Washington, D.C.

ISBN 978-1-931971-232

Finding Unknown Malice in 10 Seconds: Mass Vetting for New Threats at the Google-Play Scale

Kai Chen[‡,†], Peng Wang[†], Yeonjoon Lee[†], XiaoFeng Wang[†], Nan Zhang[†], Heqing Huang[§], Wei Zou[‡] and Peng Liu[§]
{chenkai, zouwei}@iie.ac.cn, {pw7, yl52, xw7, nz3}@indiana.edu, hhuang@cse.psu.edu, pliu@ist.psu.edu
[†]*Indiana University, Bloomington*
[‡]*State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences*
[§]*College of IST, Penn State University*

Abstract

An app market's vetting process is expected to be scalable and effective. However, today's vetting mechanisms are slow and less capable of catching new threats. In our research, we found that a more powerful solution can be found by exploiting the way Android malware is constructed and disseminated, which is typically through repackaging legitimate apps with similar malicious components. As a result, such attack payloads often stand out from those of the same repackaging origin and also show up in the apps not supposed to relate to each other.

Based upon this observation, we developed a new technique, called *MassVet*, for vetting apps at a massive scale, without knowing what malware looks like and how it behaves. Unlike existing detection mechanisms, which often utilize heavyweight program analysis techniques, our approach simply compares a submitted app with all those already on a market, focusing on the difference between those sharing a similar UI structure (indicating a possible repackaging relation), and the commonality among those seemingly unrelated. Once public libraries and other legitimate code reuse are removed, such diff/common program components become highly suspicious. In our research, we built this "DiffCom" analysis on top of an efficient similarity comparison algorithm, which maps the salient features of an app's UI structure or a method's control-flow graph to a value for a fast comparison. We implemented MassVet over a stream processing engine and evaluated it nearly 1.2 million apps from 33 app markets around the world, the scale of Google Play. Our study shows that the technique can vet an app within 10 seconds at a low false detection rate. Also, it outperformed all 54 scanners in VirusTotal (NOD32, Symantec, McAfee, etc.) in terms of detection coverage, capturing over a hundred thousand malicious apps, including over 20 likely zero-day malware and those installed millions of times. A close look at these apps brings to light intriguing new observations: e.g., Google's detection strategy and malware authors' countermoves that cause the mysterious disappearance and reappearance of some Google Play apps.

1 Introduction

The phenomenal growth of Android devices brings in a vibrant application ecosystem. Millions of applications (*app* for short) have been installed by Android users around the world from various *app markets*. Prominent examples include Google Play, Amazon Appstore, Samsung Galaxy Apps, and tens of smaller third-party markets. With this prosperity, the ecosystem is tainted by the rampancy of Android malware, which masquerades as a useful program, often through repackaging a legitimate app, to wreak havoc, e.g., intercepting one's messages, stealing personal data, sending premium SMS messages, etc. Countering this menace primarily relies on the effort from the app markets, since they are at a unique position to stop the spread of malware in the first place. Accomplishing this mission, however, is by no means trivial, as highlighted by a recent report [8] that 99% of mobile malware runs on Android devices.

Challenges in app vetting. More specifically, the protection today's app market puts in place is a *vetting process*, which screens uploaded apps by analyzing their code and operations for suspicious activities. Particularly, Google Play operates *Bouncer* [24], a security service that statically scans an app for known malicious code and then executes it within a simulated environment on Google's cloud to detect hidden malicious behavior. The problem here is that the static approach does not work on new threats (i.e., *zero-day* malware), while the dynamic one can be circumvented by an app capable of fingerprinting the testing environment, as discovered by a prior study [30]. Also the dynamic analysis can be heavyweight, which makes it hard to explore all execution paths of an app.

New designs of vetting techniques have recently been proposed by the research community [57, 28] for capturing new apps associated with known suspicious behavior, such as dynamic loading of binary code from a remote untrusted website [57], operations related to component hijacking [28], Intent injection [12], etc. All these approaches involve a heavyweight information-flow analysis and require a set of heuristics that characterize the known threats. They often need a dynamic analysis in addition to the static inspection performed on app code [57] and further human interventions to annotate the code or even participate in the analysis [14]. Moreover, emulators that most dynamic analysis tools employ can be detected and evaded by malware [23]. Also importantly, none of them has been put to a market-scale test to understand their effectiveness, nor has their performance been clearly measured.

Catching unknown malice. Actually, a vast majority of Android malware are repackaged apps [56], whose authors typically attach the same attack payload to different legitimate apps. In this way, not only do they hide their malicious program logic behind the useful functionalities of these apps, but they can also automate the repackaging process to quickly produce and distribute a large number of Trojans[1]. On the other hand, this practice makes such malware stand out from other repackaged apps, which typically incorporate nothing but advertising libraries [2]. Also as a result of the approach, similar code (typically in terms of Java methods) shows up in unrelated apps that are not supposed to share anything except popular libraries.

These observations present a new opportunity to catch malicious repackaged apps, the mainstay of Android malware, without using any heuristics to model their behavior. What we can do is to simply compare the code of related apps (an app and its repackaged versions, or those repackaged from the same app) to check their *different* part, and unrelated apps (those of different origins, signed by different parties) to inspect their *common* part to identify suspicious code segments (at the method level). These segments, once found to be inexplicable (e.g., not common libraries), are almost certain to be malicious, as discovered in our study (Section 4.2). This *DiffCom* analysis is well suited for finding previously unknown malicious behavior and also can be done efficiently, without resorting to any heavyweight information-flow technique.

Mass vetting at scale. Based on this simple idea, we developed a novel, highly-scalable vetting mechanism for detecting repackaged Android malware on one market or cross markets. We call the approach *mass vetting* or simply *MassVet*, as it does not use malware signatures and any models of expected malicious operations, and instead, solely relies on the features of existing apps on a market to vet new ones uploaded there. More specifically, to inspect a new app, MassVet runs a highly efficient DiffCom analysis on it against the whole market. Any existing app related to the new one (i.e., sharing the same repackaging origin) is quickly identified from the structural similarity of their user interfaces (aka., *views*), which are known to be largely preserved during repackaging (Section 2). Then, a *differential analysis* happens to those sharing the similar view structure (indicating a repackaging relation between them) when a match has been found. Also, an *intersection analysis* is performed to compare the new app against those with different view structures and signed by different certificates. The code components of interest discovered in this way, either the common (or similar) methods (through the intersection analysis) or different ones (by the differential analysis), are further inspected to remove common code reuses (libraries, sample code, etc.) and collect evidence for their security risks (dependence on other code, resource-access API calls, etc.), before a red flag is raised.

Supporting this mass vetting mechanism are a suite of techniques for high-performance view/code comparisons. Particularly, innovations are made to achieve a scalable analysis of different apps' user interfaces (Section 3.2). The idea is to project a set of salient features of an app's view graph (i.e., the interconnections between its user interfaces), such as types of widgets and events, to a single dimension, using a unique index to represent the app's location within the dimension and the similarity of its interface structure to those of others. In our research, we calculated this index as a geometric center of a view graph, called *v-core*. The v-cores of all the apps on the market are sorted to enable a binary search during the vetting of a new app, which makes this step highly scalable. The high-level idea here was applied to application clone detection [7], a technique that has been utilized in our research (mapping the features of a Java method to an index, called *m-core* in our research) for finding common methods across different apps (Section 3.3). It is important to note that for the view-graph comparison, new tricks need to be played to handle the structural changes caused by repackaging, e.g., when advertisement interfaces are added (Section 3.2).

Our findings. We implemented MassVet on a cloud platform, nearly 1.2 million real-world apps collected from 33 app markets around the world. Our experimental study demonstrates that MassVet vetted apps within ten seconds, with a low false positive rate. Most importantly, from the 1.2 million apps, our approach discovered 127,429 malware: among them at least 20 are likely zero-day and 34,026 were missed by the majority of the malware scanners run by *VirusTotal*, a website that syn-

[1]Those Trojans are typically signed by different keys to avoid blocking of a specific signer.

dicates 54 different antivirus products [43]. Our study further shows that MassVet achieved a better detection coverage than *any* individual scanner within VirusTotal, such as Kaspersky, Symantec, McAfee, etc. Other highlights of our findings include the discovery of malicious apps in leading app markets (30,552 from Google Play), and Google's strategies to remove malware and malware authors' countermoves, which cause mysterious disappearance and reappearance of apps on the Play Store.

Contributions. The contributions of the paper are summarized as follows:

• *New techniques*. We developed a novel mass vetting approach that detects new threats using nothing but the code of the apps already on a market. An innovative differential-intersection analysis (i.e., DiffCom) is designed to exploit the unique features of repackaging malware, catching the malicious apps even when their behavior has not been profiled *a priori*. This analysis is made scalable by its simple, static nature and the feature projection techniques that enable a cloud-based, fast search for view/code differences and similarities. Note that when the v-core and m-core datasets (only 100 GB for 1.2 million apps) are shared among multiple markets, MassVet can help one market to detect malicious submissions using the apps hosted by all these markets.

• *New discoveries*. We implemented MassVet and evaluated it using nearly 1.2 million apps, a scale unparalleled in any prior study on Android malware detection, up to our knowledge, and on a par with that of Google Play, the largest app market in the world with 1.3 million apps [39]. Our system captured tens of thousands of malware, including those slipping under the radar of most or all existing scanners, achieved a higher detection coverage than all popular malware scanners within VirusTotal and vetted new apps within ten seconds. Some malware have over millions of installs. 5,000 malware were installed over 10,000 times each, impacting hundreds of millions of mobile devices. A measurement study further sheds light on such important issues as how effective Google Play is in screening submissions, how malware authors hide and distribute their attack payloads, etc.

2 Background

Android App markets. Publishing an app on a market needs to go through an approval process. A submission will be inspected for purposes such as quality control, censorship, and also security protection. Since 2012, Google Play has been under the protection of Bouncer. This mechanism apparently contributes to the reduction of malware on the Play store, about 0.1% of all apps there as discovered by F-Secure [15]. On the other hand, this security vetting mechanism was successfully circumvented by an app that fingerprints its simulator and strategically adjusts its behavior [33]. Compared with the Android official market, how third-party markets review submitted apps is less clear. The picture painted by F-Secure, however, is quite dark: notable markets like Mumayi, AnZhi, Baidu, etc. were all found riddled with malware infiltrations [16].

Attempts to enhance the current secure vetting mechanisms mainly resort to conventional malware detection techniques. Most of these approaches, such as VetDroid [52], rely on tracking information flows within an app and the malicious behavior modeling for detecting malware. In the case that what the malware will do is less clear to the market, these approaches no longer help. Further, analyzing information flows requires semantically interpreting each instruction and carefully-designed techniques to avoid false positives, which are often heavyweight. This casts doubt on the feasibility of applying these techniques to a large-scale app vetting.

Repackaging. App repackaging is a process that modifies an app developed by another party and already released on markets to add in some new functionalities before redistributing the new app to the Android users. According to Trend Micro (July 15, 2014), nearly 80% of the top 50 free apps on Google Play have repackaged versions [49]. Even the Play store itself is reported to host 1.2% repackaged apps [58]. This ratio becomes 5% to 13% for third-party markets, according to a prior study [55]. These bogus apps are built for two purposes: either for getting advertisement revenues or for distributing malware [7]. For example, one can wrap AngryBird with ad libraries, including his own adverting ID to benefit from its advertising revenue. Malware authors also found that leveraging those popular legitimate apps is the most effective and convenient avenue to distribute their attack payloads: repackaging saves them a lot of effort to build the useful functionalities of a Trojan and the process can also be automated using the tools like smali/baksmali [36]; more importantly, they can free-ride the popularity of these apps to quickly infect a large number of victims. Indeed, research shows that the vast majority of Android malware is repackaged apps, about 86% according to a study [56]. A prominent feature shared by all these repackaged apps, malicious or not, is that they tend to keep the original user interfaces intact, so as to impersonate popular legitimate apps.

Scope and assumptions. MassVet is designed to detect repackaged Android malware. We do not consider the situation that the malware author makes his malicious payload an inseparable part of the repackaged app, which needs much more effort to understand the legitimate app than he does today. Also, MassVet can handle typical code obfuscation used in most Android apps (Section 3). However, we assume that the code has not been

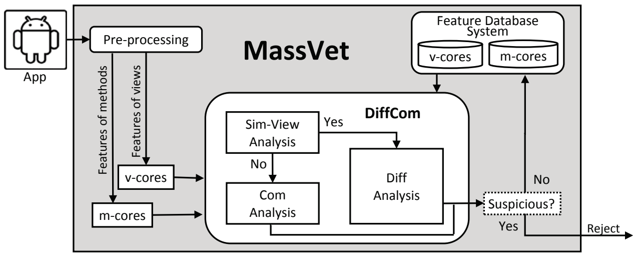

Figure 1: The Architecture of MassVet.

obfuscated to the extent that even disassembly cannot go through. When this happens, not only our approach but also most of other static analyses will fail. Finally, we assume that the app market under protection accommodates a large number of highly-diverse apps, so that for the malicious repackaged app uploaded, on the market there will be either another app sharing its repackaging origin or the one incorporating the same attack payload. To make this more likely to happen, different markets can share the feature datasets of their apps (i.e., v-cores and m-cores) with each other. Note that such datasets are very compact, only 100 GB for 1.2 million apps.

3 MassVet: Design and Implementation

3.1 Overview

Design and architecture. To detect unknown malware at a large scale, we come up with a design illustrated in Figure 1. It includes three key components: a preprocessing module, a feature database system and a DiffCom module. The preprocessing module automatically analyzes a submitted app, which includes extracting the features of its view structure and methods, and then summarizing them into the app's v-cores and m-cores respectively. The DiffCom component then works on these features, searching for them within the app market's v-core and m-core databases. Matches found there are used to identify suspicious different or common methods, which are further screened to remove false positives.

How it works. Here we use an example to walk through the work flow of the system. MassVet first processes all the apps on a market to create a v-core database for view structures and an m-core database for Java methods (Section 3.4). Both databases are sorted to support a binary search and are used for vetting new apps submitted to the market. Consider a repackaged AngryBird. Once uploaded to the market, it is first automatically disassembled at the preprocessing stage into a `smali` representation, from which its interface structures and methods are identified. Their features (for views, user interfaces, types of widgets and events, and for methods, control flow graphs and bytecode) are mapped to a set of v-cores (Section 3.2) and m-cores (Section 3.3) through calculating the geometric centers of the view graphs and control-flow graphs respectively. The app's v-cores are first used to query the database through a binary search. Once a match is found, which happens when there exists another app with a similar AngryBird user interface structure, the repackaged app is compared with the app already on the market at the method level to identify their difference. Such different methods (*diff* for short) are then automatically analyzed to ensure that they are not ads libraries and indeed suspicious, and if so, are reported to the market (Section 3.2). When the search on the v-core database comes back with nothing[2], MassVet continues to look for the AngryBird's m-cores in the method database. If a similar method has been found, our approach tries to confirm that indeed the app including the method is unrelated to the submitted AngryBird and it is not a legitimate code reuse (Section 3.3). In this case, MassVet reports that a suspicious app is found. All these steps are fully automated, without human intervention.

3.2 Fast User-Interface Analysis

As discussed before, the way MassVet vets an app depends on whether it is *related* to any other app already on the market. Such a relation is established in our research through a quick inspection of apps' user interfaces (UI) to identify those with similar view structures. When such apps are not "officially" connected, e.g., produced by the same party, the chance is that they are of the same repackaging origin, and therefore their diffs become interesting for malicious code detection. This interface-based relation identification is an alternative to code-based identification: a malicious repackaged app can be obfuscated and junk code can be easily added to make it look very different from the original version in terms of the similarity between their code (e.g., percentage of similar methods shared between them). On the other hand, a significant change to the user interface needs more effort and most importantly affects user experience, making it more difficult for the adversary to free ride the popularity of the original app. Therefore, most repackaged apps preserve their original UI structures, as found by the prior research [50]. In our research, we further discovered that many repackaged apps incorporate a large amount of new code, even more than that in their original versions, but still keep those apps' UI structures largely intact.

The idea of using view structures to detect repackaged apps has been preliminarily explored in prior research [50], which utilizes subgraph isomorphism algorithms to measure the similarity between two apps. However, the approach is less effective for the apps with relatively simple user-interface structures, and most impor-

[2]The market can also choose to perform both differential and interaction analyses for all new apps (Section 3.3).

tantly, agonizingly slow: it took 11 seconds to compare a pair of apps [50], which would need 165 days to analyze one submission against all 1.3 million apps on the Google Play store.

Following we elaborate our new solution designed for an accurate and high performance app-view analysis.

Feature extraction. An app's user interface consists of a set of views. Each view contains one or more visual widgets such as Button, ListView, TextView, etc. These UI components respond to users' input events (e.g., tapping and swiping) with the operations specified by the app developer. Such responses may cause visible changes to the current view or transitions to other views. This interconnection structure, together with the layouts and functionalities of individual views, was found to be sufficiently unique for characterizing each app [50].

In our research, we model such a UI structure as a *view graph*, which is a directed weighted graph including all views within an app and the navigation relations (that is, the transition from one view to another) among them. On such a graph, each node is a view, with the number of its *active* widgets (those with proper event-response operations) as its *weight*, and the arcs connecting the nodes describe the navigation (triggered by the input events) relations among them. According to the types of the events (e.g., `onClick`, `onFocusChange`, `onTouch`, etc.), edges can be differentiated from each other.

Such a view graph can effectively describe an app with a reasonably complicated UI structure. However, it becomes less effective for the small apps with only a couple of views and a rather straightforward connection structure. To address this issue, we enrich the view graph with additional features, including other UIs and the types of widgets that show up in a view. Specifically, in addition to view, which is displayed through invocation of an Android *Activity*, the UIs such as `AlertDialog` are also treated as nodes for the graph. Custom dialogs can be handled by analyzing class inheritance. Further, each type of widgets is given a unique value, with a sole purpose of differentiating it from other types. In this way, we can calculate a UI node's weight by adding together the values associated with the widgets it carries to make a small view graph more distinctive from others. An example is illustrated in Figure 2.

Note that we avoid using text labels on UI elements or other attributes like size or color. All the features selected here, including UIs, types of widgets and events that cause transitions among UIs, are less manipulable: in the absence of serious effort, any significant change to them (e.g., adding junk widgets, modifying the widget types, altering the transitions among views) will perceivably affect user experience, making it more difficult for the adversary to use them to impersonate popular apps.

To construct the view graph for a submitted app, the preprocessing module automatically analyzes its code to recover all UI-related interprocess communication (IPC), the channel through which an Android app invokes user interfaces. Such IPC calls include `startActivity` and `startActivityForResult`. For each call, our approach locates it within a UI and further identifies the UI it triggers. Specifically, the program location of the IPC is examined to determine whether it is inside a UI-related class v. Its parameter is parsed to find out the class it calls (v'). In this case, nodes are created on the view graph for both classes (UIs) and an edge is added to link v to v'. Also, the type of the edge is determined by the event handler at which the IPC is located: for example, when the call is found inside the `onClick` function for a button, we know that this widget is used to cause a view transition. All such widgets are identified from each class for determining the weight of its node.

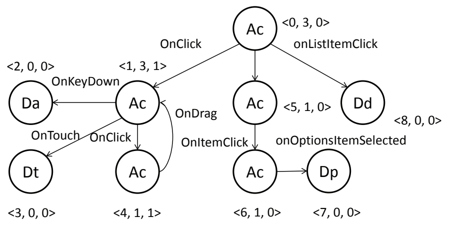

Figure 2: A View-graph example.
Ac: Activity; Da: AlertDialog; Dt: TimePickerDialog
Dp: ProgressDialog; Dd: DatePickerDialog

Design for scale. Once a view graph is recovered from an app, we want to quickly compare it across a market (or markets) to identify those related to the app. This operation needs to be of high-performance, capable of processing over one million apps within seconds. To this end, we applied a recently proposed similarity comparison algorithm, called *Centroids* [7], to the view-graph analysis. Centroid maps the features of a program's control-flow graph (CFG) into a value, which is calculated as the geometric center of the program. This value has a *monotonicity* property: that is, whenever a minor change happens to the CFG, the value changes accordingly at a small scale, reflecting the level of the difference made to the program. This property localizes the global comparison to a small number of "neighbors" to achieve high scalability without losing accuracy. The approach was used for the method comparison in our research (Section 3.3). However, it cannot be directly adopted for analyzing the UI structure, as the view graph is quite different from the CFG. Also, an app's graph is often fragmented due to the unusual ways to trigger some of its modules: e.g., most advertisement views are invoked through callbacks using the APIs of their library; as a result, their graph becomes separated from that of the main program. Here we describe how we address these issues.

Given a set of subgraphs for an app UI, $G_{i=1\cdots n}$, our preprocess module analyzes them one by one to calculate their individual geometric centers, i.e., v-cores. For a subgraph G_i, the first thing that needs to be done is to convert the features of each of its nodes (i.e., view) into a three-dimensional vector $\vec{c} = \{\alpha, \beta, \gamma\}$. Here α is a *sequence number* assigned to each node in G_i, which is done through an ordered deep-first traversal of G_i: starting from its main view, we select nodes to visit in the order of the sizes of their subtrees, and use their individual weights to break a tie; each node traversed in this way gets the number based upon its order of being visited. If two subtrees have the same size, we select the one according to their node types. In this way, we ensure that the assignment of sequence numbers is unique, which only depends on the structure of the directed weighted graph. The second element, β, in the vector is the out degree of the node: that is, the number of UIs the node can lead to. Finally, γ is the number of "transition loops" the current node is involved: i.e., the structure that from the node, there exists a navigation path that by visiting each node on the path only once, the user is able to navigate back to the current view. Figure 2 presents an example that show how such a vector is constructed.

After every node k on G_i has been given a vector $\vec{c}_k$, we can calculate its geometric center, i.e., v-core vc_i, as follows:

$$vc_i = \frac{\sum_{e(p,q)\in G_i}(w_p\vec{c}_p + w_q\vec{c}_q)}{\sum_{e(p,q)\in G_i}(w_p + w_q)}$$

where $e(p,q)$ denotes an edge in G_i from node p to q and w_p is the weight of node p. With the monotonicity of v-cores, we can sort them for a large number of apps to support a binary search. In this way, the subgraph G_i can be quickly compared with millions of graphs to identify similar ones. Specifically, given another graph G_t with a v-core vc_t, we consider that it matches G_i if $|vc_i - vc_t| \leq \tau$, where τ is a threshold. Further, given two apps sharing a subset of their view-graphs $G_{i(l=1\cdots m)}$, we consider that these two apps are similar in their UI structure when the following happens to at least one app: $\sum_l |G_{i(l)}| / \sum_i |G_i| \geq \theta$: that is, most of the app's view structures also appear in the other app (with θ being a threshold). This ensures that even when the adversary adds many new views to an app (e.g., through fake advertisements), the relation between the repackaged app and the original one can still be identified.

In our research, such thresholds were determined from a training process using 50,000 randomly selected apps (Section 3.3). We set different thresholds and measured the corresponding false positive/negative rates. For false positives, we randomly sampled 50 app pairs detected by our approach under each threshold and manually checked their relations. For false negatives, we utilized 100 app pairs known to have repackaging relations as the ground truth to find out the number of pairs our approach identified with different thresholds. The study shows that when $\tau = 0$ and $\theta = 0.8$, we got both a low false positive rate (4%) and a low false negative rate (6%). Among these 50,000 apps, we found that 26,317 app pairs had repackaging relations, involving 3,742 apps in total.

Effectiveness of the view-graph analysis. Compared with existing code-based approaches [7], the view-graph analysis turns out to be more effective at detecting apps of the same repackaging origin. Specifically, we randomly selected 10,000 app pairs (involving 17,964 apps) from those repackaged from the same programs, as discovered from 1.2 million apps we collected (Section 4.1). Many of these repackaging pairs involve the apps whose code significantly differ from each other. Particularly, in 14% of these pairs, *two apps were found to have less than 50% of their individual code in common*. This could be caused by a large library (often malicious) added to an app during repackaging or junk code inserted for the purpose of obfuscation. Since these apps look so different based upon their code, their repackaging relations cannot be easily determined by program analysis. However, they were all caught by our approach, simply because the apps' view-graphs were almost identical.

3.3 DiffCom Analysis at Scale

For an app going through the mass vetting process, the view-graph analysis first determines whether it is related to any app already on the market. If so, these two apps will be further compared to identify their diffs for a malware analysis. Otherwise, the app is checked against the whole market at the method level, in an attempt to find the program component it shares with other apps. The diffs and common component are further inspected to remove common code reuse (libraries, sample code, etc.) and collect evidence for their security risks. This "difference-commonality" analysis is performed by the DiffCom module. We also present the brick and mortar for efficient code-similarity analyzer and discuss the evasion of DiffCom.

The brick and mortar. To vet apps at the market scale, DiffCom needs a highly efficient code-similarity analyzer. In our research, we chose Centroids [7] as this building block. As discussed before, this approach projects the CFG of a program to its geometric center, in a way similar to the view-graph analysis. More specifically, the algorithm takes a basic program block as a node, which includes a sequence of consecutive statements with only a single input and output. The weight of the block is the number of the statements it contains. For each node on the CFG, a sequence number is assigned, together with the counts of the branches it connects and the number of loops it is involved in. These parameters are used to calculate the geometric center of the program.

To prepare for mass vetting, our approach first goes through all the apps on a market and breaks them into methods. After removing common libraries, the preprocessing module analyzes their code, calculates the geometric centers for individual methods (i.e., the m-cores) and then sorts them before storing the results in the database. During the vetting process, if a submitted app is found to share the view graph with another app, their diffs are quickly identified by comparing the m-cores of their individual methods. When the app needs to go through the intersection step, its methods are used for a binary search on the m-core database, which quickly discovers those also included in existing apps. Here we elaborate how these operations are performed. Their overhead is measured in Section 4.2.

Analyzing diffs. Whenever an app is found to relate to another one from their common view graph, we want to inspect the difference part of their code to identify suspicious activities. The rationale is that repackaged apps are the mainstay of Android malware, and the malicious payloads are often injected automatically (using tools like smali/baksmali) without any significant changes to the code of the original app, which can therefore be located by looking at the diffs between the apps of the same repackaging origin. Such diffs are quickly identified by comparing these two apps' m-cores: given two ordered sequences of m-cores L and L', the diff between the apps at the method level is found by merging these two lists according to the orders of their elements and then removing those matching their counterparts on the other list; this can be done within $\min(|L|,|L'|)$ steps.

However, similarity of apps' UIs does not always indicate a repackaging relation between them. The problem happens to the apps produced by the same party, individual developers or an organization. In this case, it is understandable that the same libraries and UIs could be reused among their different products. It is even possible that one app is actually an updated version of the other. Also, among different developers, open UI SDKs such as Appcelerator [3] and templates like Envatomarket [13] are popular, which could cause the view structures of unrelated apps to look similar. Further, even when the apps are indeed repackaged, the difference between them could be just advertisement (ad) libraries instead of malicious payloads. A challenge here is how to identify these situations and avoid bringing in false alarms.

To address these issues, MassVet first cleans up a submitted app's methods, removing ad and other libraries, before vetting the app against the market. Specifically, we maintain a white list of legitimate ad libraries based on [6], which includes popular mobile ad platforms such as MobWin, Admob, etc. To identify less known ones, we analyzed a training set of 50,000 apps randomly sampled from three app markets, with half of them from Google Play. From these apps, our analysis discovered 34,886 methods shared by at least 27,057 apps signed by different parties. For each of these methods, we further scanned its hosting apps using VirusTotal. If none of them were found to be malicious, we placed the method on the white list. In a similar way, popular view graphs among these apps were identified and the libraries associated with these views are white-listed to avoid detecting false repackaging relations during the view-graph analysis. Also, other common libraries such as Admob were also removed during this process, which we elaborate later. Given the significant size of the training set (50,000 randomly selected apps), most if not all legitimate libraries are almost certain to be identified. This is particularly true for those associated with advertising, as they need certain popularity to remain profitable. On the other hand, it is possible that the approach may let some zero-day malware fall through the cracks. In our research, we further randomly selected 50 ad-related methods on the list and searched for them on the Web, and confirmed that all of them were indeed legitimate. With this false-negative risk, still our approach achieved a high detection coverage, higher than any scanner integrated in VirusTotal (Section 4.2).

When it comes to the apps produced by the same party, the code they share is less popular and therefore may not be identified by the approach. The simplest solution here is to look at similar apps' signatures: those signed by the same party are not considered to be suspicious because they do have a good reason to be related. This simple treatment works most of time, since legitimate app vendors typically sign their products using the same certificate. However, there are situations when two legitimate apps are signed by different certificates but actually come from the same source. When this happens, the diffs of the apps will be reported and investigated as suspicious code. To avoid the false alarm, we took a close look at the legitimate diffs, which are characterized by their intensive connections with other part of the app. They are invoked by other methods and in the meantime call the common part of the code between the apps. On the other hand, the malicious payload packaged to a legitimate app tends to stand alone, and can only be triggered from a few (typically just one) program locations and rarely call the components within the original program.

In our research, we leveraged this observation to differentiate suspicious diffs from those likely to be legitimate. For each diff detected, the DiffCom analyzer looks for the calls it makes toward the rest of the program and inspects the `smali` code of the app to identify the references to the methods within the diff. These methods will go through a further analysis only when such interactions are very limited, typically just an inward invocation, without any outbound call. Note that current malware authors do not make their code more connected to the legitimate app they repackage, mainly because more effort is needed to understand the code of the app and carefully construct the attack. A further study is needed to understand the additional cost required to build more sophisticated malware to evade our detection.

For the diff found in this way, DiffCom takes further measures to determine its risk. A simple approach used in our implementation is to check the presence of API calls (either Android framework APIs or those associated with popular libraries) related to the operations considered to be dangerous. Examples include `getSimSerialNumber`, `sendTextMessage` and `getLastKnownLocation`. The findings here indicate that the diff code indeed has the means to cause damage to the mobile user's information assets, though how exactly this can happen is not specified. This is different from existing behavior-based detection [27], which looks for much more specific operation sequences such as "reading from the contact list and then sending it to the Internet". Such a treatment helps suppress false alarms and still preserves the generality of our design, which aims at detecting unknown malicious activities.

Analyzing intersections. When no apparent connection has been found between an app and those already on the market, the vetting process needs to go through an intersection analysis. This also happens when DiffCom is configured to perform the analysis on the app that has not been found to be malicious at the differential step. Identification of common methods a newly submitted app carries is rather straightforward: each method of the app is mapped to its m-core, which is used to search against the m-core database. As discussed before, this can be done through a binary search. Once a match is found, DiffCom further inspects it, removing legitimate connections between the apps, and reports the finding to the market.

Again, the main challenge here is to determine whether two apps are indeed unrelated. A simple signature check removes most of such connections but not all. The "stand-alone" test, which checks whether a set of methods intensively interact with the rest of an app, does not work for the intersection test. The problem here is that the common methods between two repackaged apps may not be the complete picture of a malicious payload, making them different from the diff identified in the differential-analysis step: different malware authors often use some common toolkits in their attack payloads, which show up in the intersection between their apps; these modules still include heavy interactions with other components of the malware that are not found inside the intersection. As a result, this feature, which works well on diffs, cannot help to capture suspicious common code among apps.

An alternative solution here is to look at how the seemingly unrelated apps are actually connected. As discussed before, what causes the problem is the developers or organizations that reuse code internally (e.g., a proprietary SDK) but sign the apps using different certificates. Once such a relation is also identified, we will be more confident about whether two apps sharing code are independent from each other. In this case, the common code becomes suspicious after all public libraries (e.g., those on the list used in the prior research [6]) and code templates have been removed. Here we describe a simple technique for detecting such a hidden relation.

From our training dataset, we found that most code reused legitimately in this situation involves user interfaces: the developers tend to leverage existing view designs to quickly build up new apps. With this practice, even though two apps may not appear similar enough in terms of their complete UI structures (therefore they are considered to be "unrelated" by the view-graph analysis), a close look at the subgraphs of their views may reveal that they actually share a significant portion of their views and even subgraphs. Specifically, from the 50,000 apps in our training set, after removing public libraries, we found 30,286 sharing at least 30% of their views with other apps, 16,500 sharing 50% and 8,683 containing no less than 80% common views. By randomly sampling these apps (10 each time) and analyzing them manually, we confirmed that when the portion goes above 50%, almost all the apps and their counterparts are either from the same developers or organizations, or having the same repackaging origins. Also, once the shared views become 80% or more, almost always the apps are involved in repackaging. Based upon this observation, we run an additional correlation check on a pair of apps with common code: DiffCom compares their individual subgraphs again and if a significant portion (50%) is found to be similar, they are considered related and therefore their intersection will not be reported to the market.

After the correlation check, all the apps going through the intersection analysis are very likely to be unrelated. Therefore, legitimate code shared between them, if any, is almost always public libraries or templates. As described before, we removed such common code through white-listing popular libraries and further complemented the list with those discovered from the training set: methods in at least 2,363 apps were considered legitimate public resources if all these apps were cleared by VirusTotal. Such code was further sampled and manually analyzed in our study to ensure that it indeed did not involve any suspicious activities. With all such libraries removed, the shared code, particularly the method with dangerous APIs (e.g., `getSimSerialNumber`), is reported as possible malicious payload.

Evading MassVet. To evade MassVet, the adversary could try to obfuscate app code, which can be tolerated to some degree by the similarity comparison algorithm we use [7]. For example, common obfuscation techniques, such as variable/method renaming, do not affect centroids. Also the commonality analysis can only be defeated when the adversary is willing to significantly alter the attack payload (e.g., a malicious SDK) each time when he uses it for repackaging. This is not supported by existing automatic tools like ADAM [53] and DroidChameleon [32], which always convert the same code to the same obfuscated form. Further, a deep obfuscation will arouse suspicion when the app is compared with its repackaging origin that has not been obfuscated. The adversary may also attempt to obfuscate the app's view graphs. This, however, is harder than obfuscating code, as key elements in the graph, like event functions `OnClick`, `OnDrag`, etc., are hardcoded within the Android framework and cannot be modified. Also adding junk views can be more difficult than it appears to be: the adversary cannot simply throw in views disconnected from existing sub-graphs, as they will not affect how MassVet determines whether two view-graphs match (Section 3.2); otherwise, he may connect such views to existing sub-graphs (potentially allowing them to be visited from the existing UI), which requires understanding a legitimate app's UI structures to avoid affecting user experience.

We further analyzed the effectiveness of existing obfuscation techniques against our view-graph approach over 100 randomly selected Google-Play apps. Popular obfuscation tools such as DexGuard [37] and ProGuard [38] only work on Java bytecode, not the Dalvik bytecode of these commercial apps. In our research, we utilized ADAM [53] and DroidChameleon [32], which are designed for Dalvik bytecode, and are highly effective according to prior studies [53, 32]. Supposedly they can also work on view-related code within those apps. However after running them on the apps, we found that their v-cores, compared with those before the obfuscation, did not change at all. This demonstrates that such obfuscation is not effective on our view-graph approach.

On the other hand, we acknowledge that a new obfuscation tool could be built to defeat MassVet, particularly its view-graph search and the Com step. The cost for doing this, however, is less clear and needs further effort to understand (Section 5).

3.4 System Building

In our research, we implemented a prototype of MassVet using C and Python, nearly 1.2 million apps collected from 33 markets, including over 400,000 from Google Play (Section 4.1). Before these apps can be used to vet new submissions, they need to be inspected to detect malicious code already there. Analyzing apps of this scale and utilizing them for a real-time vetting require carefully designed techniques and a proper infrastructure support, which we elaborate in this section.

System bootstrapping and malware detection. To bootstrap our system, a market first needs to go through all its existing apps using our techniques in an efficient way. The APKs of these apps are decompiled into `smali` (using the tool baksmali [36][3]) to extract their view graphs and individual methods, which are further converted into v-cores and m-cores respectively. We use NetworkX [29] to handle graphs and find loops. Then these features (i.e., cores) are sorted and indexed before stored into their individual databases. In our implementation, such data are preserved using the Sqlite database system, which is characterized by its small size and efficiency. For all these apps, 1.5 GB was generated for v-cores and 97 GB for m-cores.

The next step is to scan all these 1.2 million apps for malicious content. A straightforward approach is to inspect them one by one through the binary search. This will take tens of millions of steps for comparisons and analysis. Our implementation includes an efficient alternative. Specifically, on the v-core database, our system goes through the whole sequence from one end (the smallest element) to the other end, evaluating the elements along the way to form *equivalent groups*: all those with identical v-cores are assigned to the same group[4].

All the subgraphs within the same group match each other. However, assembling them together to determine the similarity between two apps turns out to be tricky. This is because the UI of each app is typically broken into around 20 subgraphs distributed across the whole ordered v-core sequence. As such, any attempt to make the comparison between two apps requires to go through all equivalent groups. The fastest way to do that is to maintain a table for the number of view subgraphs shared between any two apps. However, this simple approach requires a huge table, half of 1.2 million by 1.2 million in the worst case, which cannot be completely loaded into the memory. In our implementation, a trade-off is made to save space by only inspecting 20,000 apps against the rest 1.2 million each time, which requires going through the equivalent groups for 60 times and uses about 100 GB memory at each round.

The inspection on m-cores is much simpler and does not need to compare one app against all others. This is because all we care about here are just the common methods that already show up within individual equiva-

[3]Very few apps (0.01%) cannot be decompiled in our dataset due to the limitation of the tool.

[4]In our implementation, we set the threshold τ to zero, which can certainly be adjusted to tolerate some minor variations among similar methods.

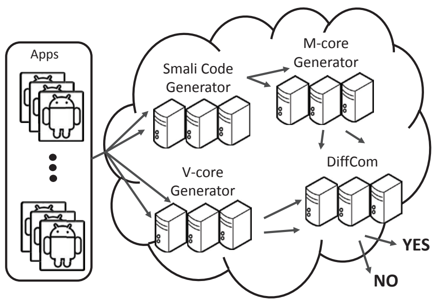

Figure 3: Cloud framework for MassVet.

lent groups. Those methods are then further analyzed to detect suspicious ones.

Cloud support. To support a high-performance vetting of apps, MassVet is designed to work on the cloud, running on top of a stream processing framework (Figure 3). Specifically, our implementation was built on *Storm* [40], an open-source stream-processing engine that also powers leading web services such as WebMD, Alibaba, Yelp, etc. Storm supports a large-scale analysis of a data stream by a set of worker units that connect to each other, forming a topology. In our implementation, the work flow of the whole vetting process is converted into such a topology: a submitted app is first disassembled to extract view graphs and methods, which are checked against the white list to remove legitimate libraries and templates; then, the app's v-cores and m-cores are calculated, and a binary search on the v-core database is performed; depending on the results of the search, the differential analysis is first run, which can be followed by the intersection analysis. Each operation here is delegated to a worker unit on the topology and all the data associated with the app are in a single stream. The Storm engine is designed to support concurrently processing multiple streams, which enables a market to efficiently vet a large number of submissions.

4 Evaluation and Measurement

4.1 Setting the Stage

App collection. We collected 1.2 million apps from 33 Android markets, including over 400,000 from Google Play, 596,437 from 28 app stores in China, 61,866 from European stores and 27,047 from other US stores as elaborated in Table 5 in Appendix. We removed duplicated apps according to their MD5. All the apps we downloaded from Google Play have complete meta data (upload date, size, number of downloads, developer, etc.), while all those from third-party markets do not.

The apps from Google Play were selected from 42 categories, including Entertainment, Tools, Social, Communication, etc. From each category, we first went for its top 500 popular ones (in terms of number of installs) and then randomly picked up 1000 to 30,000 across the whole category. For each third-party market, we just randomly collected a set of apps (Table 5) (190 to 108,736, depending on market sizes). Our collection includes high-profiled apps such as Facebook, Skype, Yelp, Pinterest, WeChat, etc. and those less popular ones. Their sizes range from less than 1 MB to more than 100 MB.

Validation. For the suspicious apps reported by our prototype, we validated them through VirusTotal and manual evaluations. Virustotal is the most powerful public malware detection system, which is a collection of 54 anti-malware products, including the most high-profile commercial scanners. It also provides the scanning service on mobile apps [44]. VirusTotal has two modes, complete scanning (which we call *new scan*) and using cached results (called *cached scan*). The latter is fast, capable of going through 200 apps every minute, but only covers those that have been scanned before. For the programs it has never seen or uploaded only recently, the outcome is just "unknown". The former determines whether an app is malicious by running on it all 54 scanners integrated within VirusTotal. The result is more up-to-date but the operation is much slower, taking 5 minutes for each app.

To validate tens of thousands suspicious cases detected from the 1.2 million apps (Section 4.2), we first performed the cached scan to confirm that most of our findings were indeed malicious. The apps reported to be "unknown" were further randomly sampled for a new scan. For all the apps that VirusTotal did not find malicious, we further picked up a few samples for a manual analysis. Particularly, for all suspicious apps identified by the intersection analysis, we clustered them according to their shared code. Within each cluster, whenever we observed that most members were confirmed malware, we concluded that highly likely the remaining apps there are also suspicious, even if they were not flagged by VirusTotal. The common code of these apps were further inspected for suspicious activities such as information leaks. A similar approach was employed to understand the diff code extracted during the differential analysis. We manually identified the activities performed by the code and labeled it as suspicious when they could lead to damages to the user's information assets.

4.2 Effectiveness and Performance

Malware found and coverage. From our collection, MassVet reported 127,429 suspicious apps (10.93%). 10,202 of them were caught by "Diff" and the rest were discovered by "Com". These suspicious apps are from different markets: 30,552 from Google Play and 96,877 from the third-party markets, as illustrated in Table 5. We first validated these findings by uploading them to VirusTotal for a cached scan (i.e., quickly checking the apps against the checksums cached by VirusTotal),

AV Name	# of Detection	% Percentage
Ours (MassVet)	197	70.11
ESET-NOD32	171	60.85
VIPRE	136	48.40
NANO-Antivirus	120	42.70
AVware	87	30.96
Avira	79	28.11
Fortinet	71	25.27
AntiVir	60	21.35
Ikarus	60	21.35
TrendMicro-HouseCall	59	21.00
F-Prot	47	16.73
Sophos	46	16.37
McAfee	45	16.01

Table 1: The coverages of other leading AV scanners.

which came back with 91,648 confirmed cases (72%), 17,061 presumably false positives (13.38%, that is, the apps whose checksums were in the cache but not found to be malicious when they were scanned) and 13,492 unknown (10.59%, that is, the apps whose checksums were not in VirusTotal's cache). We further randomly selected 2,486 samples from the unknown set and 1,045 from the "false-positive" set, and submitted to VirusTotal again for a new scan (i.e., running all the scanners, with the most up-to-date malware signatures, on the apps). It turned out that 2,340 (94.12%) of unknown cases and 349 (33.40%) of "false positives" are actually malicious apps, according to the new analysis. This gives us a false detection rate (FDR: false positives vs. all detected) of 9.46% and a false positive rate (FPR: false positives vs. all apps analyzed) of 1%, solely based upon VirusTotal's scan results. Note that the Com step found more malware than Diff, as Diff relies on the presence of two apps of same repackaging origins in the dataset, while Com only looks for common attack payloads shared among apps. It turns out that many malicious-apps utilize same malicious SDKs, which make them easier to catch.

We further randomly sampled 40 false positives reported by the new scan for a manual validation and found that 20 of them actually are highly suspicious. Specifically, three of them load and execute suspicious code dynamically; one takes pictures stealthily; one performs sensitive operation to modify the booting sequence of other apps; seven of them get sensitive user information such as SIM card SN number and telephone number/ID; several aggressive adware turn out to add phishing plugins and app installation bars without the user's consent. The presence of these activities makes us believe that very likely they are actually zero-day malware. We have reported all of them to four Antivirus software vendors such as Norton and F-Secure for a further analysis. If all these cases are confirmed, then the FDR of MassVet could further be reduced to 4.73%.

To understand the coverage of our approach, we randomly sampled 2,700 apps from Google Play and scanned them using MassVet and the 54 scanners within VirusTotal. All together, VirusTotal detected 281 apps and among them our approach got 197 apps. The coverage of MassVet, with regard to the collective result of all 54 scanners, is 70.1%, better than what could be achieved by any individual scanner integrated within VirusTotal, including such top-of-the-line antivirus systems as NOD32 (60.8%), Trend (21.0%), Symantec (5.3%) and McAfee (16%). Most importantly, MassVet caught at least 11% malware those scanners missed. The details of the study are presented in Table 1 (top 12).

# Apps	Pre-Processing analysis	v-core database search	differential	m-core database search (Intersection)	sum
10	5.84	0.15	0.33	1.80	8.12
50	5.85	0.15	0.34	1.99	8.33
100	5.85	0.14	0.35	2.23	8.57
200	5.88	0.16	0.35	3.13	9.52
500	5.88	0.16	0.35	3.56	9.95

Table 2: Performance: "Apps" here refers to the number of concurrently submitted apps.

Vetting delay. We measured the performance of our technique, on a server with 260 GB memory, 40 cores at 2.8 GHz and 28 TB hard drives. Running on top of the Storm stream processor, our prototype was tested against 1 to 500 concurrently submitted apps. The average delay we observed is 9 seconds, from the submission of the app to the completion of the whole process on it. This vetting operation was performed against all 1.2 million apps.

Table 2 further shows the breakdown of the vetting time at different vetting stages, including preprocessing (v-core and m-core generation), search across the v-core database, the differential analysis, search over the m-core database and the intersection analysis. Overall, we show that MassVet is indeed capable of scaling to the level of real-world markets to provide a real-time vetting service.

4.3 Measurement and Findings

Over the 127,429 malicious apps detected in our study, we performed a measurement study that brings to light a few interesting observations important for understanding the seriousness of the malware threat to the Android ecosystem, as elaborated below.

Landscape. The malware we found are distributed across the world: over 35,473 from North America, 4,852 from Europe and 87,104 from Asia. In terms of the portion of malicious code within all apps, Chinese app markets take the lead with 12.90%, which is followed by US, with 8.28%. This observation points to a possible lack of regulations and proper security protection in many Chinese markets, compared with those in other countries. Even among the apps downloaded from Google Play, over 7.61% are malicious, which is different from a prior report of only 0.1% malware discovered there [15]. Note that most of the malware here has been confirmed by VirusTotal. This indicates that indeed the portion of the apps with suspicious activities on leading app stores could be higher than previously thought. De-

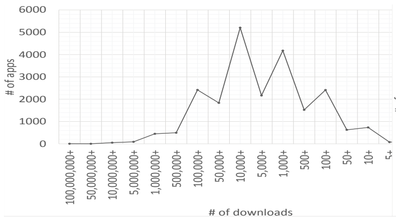

Figure 4: The distribution of downloads for malicious or suspicious apps in Google Play.

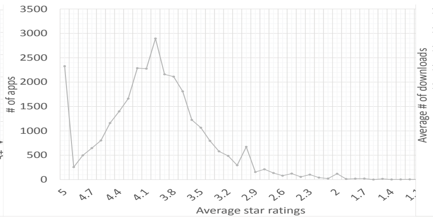

Figure 5: The distribution of rating for malicious or suspicious apps in Google Play.

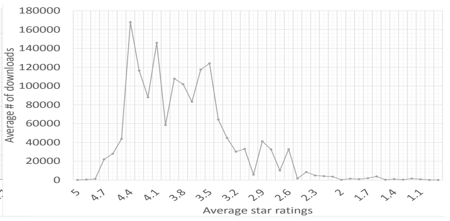

Figure 6: The distribution of average number of downloads for malicious or suspicious apps in Google Play.

tailed numbers of malicious apps are shown in Appendix (Table 5).

We observed that most scanners react slowly to the emergence of new malware. For all 91,648 malicious apps confirmed by VirusTotal, only 4.1% were alarmed by at least 25 out of 54 scanners it hosts. The results are present in Figure 7. This finding also demonstrates the capability of MassVet to capture new malicious content missed by most commercial scanners.

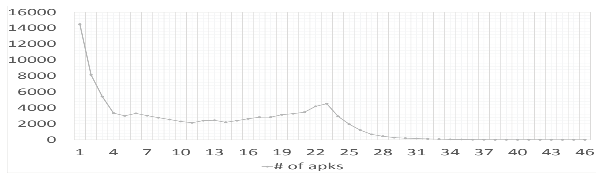

Figure 7: Number of malware detected by VirusTotal.

The impacts of those malicious apps are significant. Over 5,000 such apps have already been installed over 10,000 times each (Figure 4). Also, there are a few extremely popular ones, with the install count reaching 1 million or even more. Also, the Google-Play ratings of the suspicious APKs are high (most of them ranging from 3.6 to 4.6, Figure 5), with each being downloaded for many times (100,000 to 250,000) on average (Figure 6). This suggests that hundreds of millions of mobile devices might have already been infected.

Existing defense and disappeared apps. Apparently, Google Play indeed makes effort to mitigate the malware threat. However, our measurement study also shows the challenge of this mission. As Figure 8 illustrates, most malware we discovered were uploaded in the past 14 months. Also the more recently an app shows up, the more likely it is problematic. This indicates that Google Play continuously inspects the apps it hosts to remove the suspicious ones. For the apps that have already been there for a while, the chance is that they are quite legitimate, with only 4.5% found to be malicious. On the other hand, the newly released apps are much less trustworthy, with 10.69% of them being suspicious. Also, these malicious apps have a pretty long shelf time, as Google needs up to 14 months to remove most of them. Among the malware we discovered, 3 apps uploaded in Dec. 2010 are still there in Google Play.

Interestingly, 40 days after uploading 3,711 apps (those we asked VirusTotal to run *new scan* upon, as mentioned earlier) to VirusTotal, we found that 250 of them disappeared from Google Play. 90 days later, another 129 apps disappeared. Among the 379 disappeared apps, 54 apps (14%) were detected by VirusTotal. Apparently, Google does not run VirusTotal for its vetting but pays close attention to the new malware it finds.

We further identified 2,265 developers of the 3,711 suspicious apps, using the apps' meta data, and monitored all their apps in the follow-up 15 weeks (November 2014 to February 2015). Within this period, we observed that additional 204 apps under these developers disappeared, all of which were detected by MassVet, *due to the suspicious methods they shared with the malware we caught before that period*. The interesting part is that we did not scan these apps within VirusTotal, which indicates that it is likely that Google Play also looked into their malicious components and utilized them to check all other apps under the same developers. However, apparently, Google did not do this across the whole marketplace, because we found that other apps carrying those methods were still there on Google Play. If these apps were missed due to the cost for scanning all the apps on the Play Store, MassVet might actually be useful here: our prototype is able to compare a method across all 1.2 million apps within 0.1 second.

Another interesting finding is that we saw that some of these developers uploaded the same or similar malicious apps again after they were removed. Actually, among the 2,125 reappeared apps, 604 confirmed malware (28.4%) showed up in the Play Store *unchanged*, with the same MD5 and same names. Further, those developers also published 829 apps with the same malicious code (as that of the malware) but under different names. The fact that the apps with known malicious payloads still got slipped in suggests that Google might not pay adequate attention to even known malware.

Repackaging malware and malicious payload. Among the small set of repackaging malware captured by the differential analysis, most are from third-party stores (92.35%). Interestingly, rarely did we observe that malware authors repackaged Google Play apps

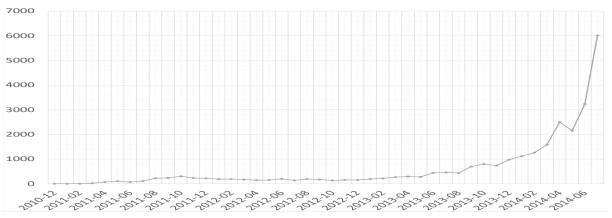

Figure 8: Number of malicious apps overtime.

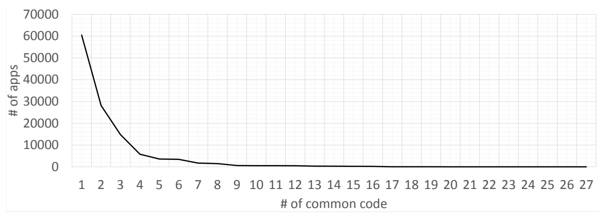

Figure 9: The distribution of common code across malware.

Signature	# of malicious apps
c673c8a5f021a5bdc5c036ee30541dde	1644
a2993eaecf1e3c2bcad4769cb79f1556	1258
3be7d6ee0dca7e8d76ec68cf0ccd3a4a	615
f8956f66b67be5490ba6ac24b5c26997	559
86c2331f1d3bb4af2e88f485ca5a4b3d	469

Table 3: Top 5 signatures used in apps.

and distributed them to the third-party stores in China. Instead, malware repackaging appears to be quite localized, mostly between the app stores in the same region or even on the same store. A possible explanation could be the effort that malware authors need to make on the original app so that it works for a new audience, which is certainly higher than simply repackaging the popular one in the local markets.

Figure 9 illustrates the distribution of common code across malware, as discovered from the intersection analysis. A relatively small set of methods have been reused by a large number of malicious apps. The leading one has been utilized by 9,438 Google-Play malware and by 144 suspicious apps in the third-party markets. This method turns out to be part of the library ("com/startapp") extensively used by malware. Over 98% of the apps integrating this library were flagged as malicious by VirusTotal and the rest were also found to be suspicious through our manual validation. This method sends users' fine-grained location information to a suspicious website. Similarly, all other popular methods are apparently also part of malware-building toolkits. Examples include "guohead", "purchasesdk" and "SDKUtils". The malware integrating such libraries are signed by thousands of different parties. An observation is that the use of these malicious SDKs is pretty regional: in Chinese markets, "purchasesdk" is popular, while "startapp" is widely used in the US markets. We also noticed that a number of libraries have been obfuscated. A close look at these attack SDKs shows that they are used for getting sensitive information like phone numbers, downloading files and loading code dynamically.

Signatures and identities. For each confirmed malicious app, we took a look at its "signature", that is, the public key on its X.509 certificate for verifying the integrity of the app. Some signatures have been utilized by more than 1,000 malware each: apparently, some malware authors have produced a large number of malicious apps and successfully disseminated them across different markets (Table 3). Further, when we checked the meta data for the malware discovered on Google Play, we found that a few signatures have been associated with many *identities* (e.g., the `creator` field in the metadata). Particularly, one signature has been linked to 604 identities, which indicates that the adversary might have created many accounts to distribute his app (Table 4).

Case studies. Among the suspicious apps MassVet reported are a set of APKs not even VirusTotal found to be malicious. We analyzed 40 samples randomly chosen from this set and concluded that 20 of them were indeed problematic through manual analysis, likely to be zero-day malware. We have reported them to 4 malware companies (F-Secure, Norton, Kaspersky, Trend Micro) for further validations. The behaviors of these apps include installing apps without user's consent, collecting user's private data (e.g., take screen shots of other apps) even though such information does not serve apps' stated functionalities, loading and executing native binary for command and control.

These apps use various techniques to evade detection. For example, some hide the suspicious functionality for weeks before starting to run it. "Durak card game" is such an game, which has been downloaded over 5,000,000 times. It was on Google Play before BBC reported it on February 4th 2015 [25]. So far, only two scanners hosted by VirusTotal can detect it. This malware disguises as warning messages when the user unlock her Android smartphone. It waits for several weeks before performing malicious activities. Its advertisements also do not show up until at least one reboot. Although Google removes "Durak card game", other apps with similar functionalities are still on the Play Store now. We also found that some malicious apps conceal their program logic inside native binaries. Some even encrypt the binaries and dynamically decrypt them for execution. Further some utilize Java reflection and other obfuscation techniques to cover their malicious code.

Signature	# of different identities
02d98ddfbcd202b13c49330182129e05	604
a2993eaecf1e3c2bcad4769cb79f1556	447
82fd3091310ce901a889676eb4531f1e	321
9187c187a43b469fa1f995833080e7c3	294
c0520c6e71446f9ebdf8047705b7bda9	145

Table 4: Top 5 signatures used by different identities.

5 Discussion

As discussed before, MassVet aims at repackaging malware, the mainstay of potentially harmful mobile apps: this is because malware authors typically cannot afford to spend a lot of time and money to build a popular app just for spreading malware, only to be forced to do this all over again once it gets caught. Our technique exploits a weakness of their business model, which relies on repackaging popular apps with a similar attack payload to keep the cost for malware distribution low. With the fundamentality of the issue and the effectiveness of the technique on such malware, our current implementation, however, is still limited, particularly when it comes to the defense against evasion.

Specifically, though simply adding the junk views connected to an existing app's view graph can affect user experience and therefore may not work well (Section 3.3), a more effective alternative is to obfuscate the links between views (calls like `StartActivity`). However, this treatment renders an app's UI structure less clear to our analyzer, which is highly suspicious, as the vast majority of apps' view graphs can be directly extracted. What we could do is to perform a dynamic analysis on such an app, using the tools like *Monkey* to explore the connections between different views. Note that the overall performance impact here can still be limited, simply because most apps submitted to an app store are legitimate and their UI structures can be statically determined.

Further, to evade the commonality analysis, the adversary could obfuscate the malicious methods. As discussed earlier (Section 3.3), this attempt itself is nontrivial, as the m-cores of those methods can only be moved significantly away from their original values through substantial changes to their CFGs each time when a legitimate app is repackaged. This can be done by adding a large amount of junk code on the CFGs. Our current implementation does not detect such an attack, since it is still no there in real-world malware code. On the other hand, further studies are certainly needed to better understand and mitigate such a threat.

Critical to the success of our DiffCom analysis is removal of legitimate libraries. As an example, we could utilize a crawler to periodically gather shared libraries and code templates from the web to update our whitelists. Further, a set of high-profile legitimate apps can be analyzed to identify the shared code missed by the crawler. What can also be leveraged here is a few unique resources in the possession of the app market. For example, it knows the account from which the apps are uploaded, even though they are signed by different certificates. It is likely that legitimate organizations are only maintaining one account and even when they do have multiple ones, they will not conceal the relations among them. Using such information, the market can find out whether two apps are actually related to identify the internal libraries they share. In general, given the fact that MassVet uses a large number of existing apps (most of which are legitimate) to vet a small set of submissions, it is at the right position to identify and remove most if not all legitimate shared code.

6 Related Work

Malicious app detection. App vetting largely relies on the techniques for detecting Android malware. Most existing approaches identify malicious apps either based upon how they look like (i.e., content-based signature) [20, 27, 21, 45, 51, 57, 19, 54, 17, 22, 4] or how they act (behavior-based signature) [11, 31, 48, 47, 42, 18, 34]. Those approaches typically rely on heavyweight static or dynamic analysis techniques, and cannot detect the unknown malware whose behavior has not been modeled a priori. MassVet is designed to address these issues by leveraging unique properties of repackaging malware. Most related to our work is PiggyApp [54], which utilizes the features (permissions, APIs, etc.) identified from a major component shared between two apps to find other apps also including this component, then clusters the rest part of these apps' code, called *piggybacked payloads*, and samples from individual clusters to manually determine whether the payloads there are indeed malicious. In contrast, MassVet *automatically* detects malware through inspecting the code diff among apps with a similar UI structure and the common methods shared between those unrelated. When it comes to the scale of our study, ANDRUBIS [26, 46] dynamically examined the operations of over 1 million apps in four years. Different from ANDRUBIS, which is an off-line analyzer for recovering detailed behavior of individual malicious apps, MassVet is meant to be a fast online scanner for identifying malware without knowing its behavior. It went through 1.2 million of apps within a short period of time.

Repackaging and code reuse detection. Related to our work is repackaging and code reuse detection [55, 21, 1, 9, 10, 41, 35, 5]. Most relevant to MassVet is the Centroids similarity comparison [7], which is also proposed for detecting code reuse. Although it is a building block for our technique, the approach itself does *not* detect malicious apps. Significant effort was made in our research to build view-graph and code analysis on top of it to achieve an accurate malware scan. Also, to defeat code obfuscation, a recent proposal leverages the similarity between repackaged apps' UIs to detect their relations [50]. However, it is too slow, requiring 11 seconds to process a pair of apps. In our research, we come up with a more effective UI comparison technique, through mapping the features of view graphs to their geometric centers, as Centroids does. This significantly improves the performance of the UI-based approach, enabling it to help vet a large number of apps in real time.

7 Conclusion

We present MassVet, an innovative malware detection technique that compares a submitted app with all other apps on a market, focusing on its diffs with those having a similar UI structure and intersections with others. Our implementation was used to analyze nearly 1.2 million apps, a scale on par with that of Google Play, and discovered 127,429 malicious apps, with 20 likely to be zero-day. The approach also achieves a higher coverage than leading anti-malware products in the market.

Acknowledgement

We thank our shepherd Adam Doupé and anonymous reviewers for their valuable comments. We also thank Dr. Sencun Zhu and Dr. Fangfang Zhang for sharing their ViewDroid code, which allows us to understand how their system works, and VirusTotal for the help in validating over 100,000 apps discovered in our study. IU authors were supported in part by the NSF 1117106, 1223477 and 1223495. Kai Chen was supported in part by NSFC 61100226, 61170281 and strategic priority research program of CAS (XDA06030600). Peng Liu was supported by NSF CCF-1320605 and ARO W911NF-09-1-0525 (MURI).

References

[1] ANDROGUARD. Reverse engineering, malware and goodware analysis of android applications ... and more. http://code.google.com/p/androguard/, 2013.

[2] APPBRAIN. Ad networks - android library statistics — appbrain.com. http://www.appbrain.com/stats/libraries/ad. (Visited on 11/11/2014).

[3] APPCELERATOR. 6 steps to great mobile apps. http://www.appcelerator.com/. 2014.

[4] ARZT, S., RASTHOFER, S., FRITZ, C., BODDEN, E., BARTEL, A., KLEIN, J., LE TRAON, Y., OCTEAU, D., AND MCDANIEL, P. Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps. In *PLDI* (2014), ACM, p. 29.

[5] BAYER, U., COMPARETTI, P. M., HLAUSCHEK, C., KRUEGEL, C., AND KIRDA, E. Scalable, behavior-based malware clustering. In *NDSS* (2009), vol. 9, Citeseer, pp. 8–11.

[6] CHEN, K. A list of shared libraries and ad libraries used in android apps. http://sites.psu.edu/kaichen/ 2014/02/20/a-list-of-shared-libraries-and-ad-libraries-used-in-android-apps.

[7] CHEN, K., LIU, P., AND ZHANG, Y. Achieving accuracy and scalability simultaneously in detecting application clones on android markets. In *ICSE* (2014).

[8] CISCO. "cisco 2014 annual security report,". http://www.cisco.com/web/offer/gist_ty2_asset/ Cisco_2014_ASR.pdf, 2014.

[9] CRUSSELL, J., GIBLER, C., AND CHEN, H. Attack of the clones: Detecting cloned applications on android markets. *ESORICS* (2012), 37–54.

[10] CRUSSELL, J., GIBLER, C., AND CHEN, H. Scalable semantics-based detection of similar android applications. In *ESORICS* (2013).

[11] ENCK, W., GILBERT, P., CHUN, B.-G., COX, L. P., JUNG, J., MCDANIEL, P., AND SHETH, A. Taintdroid: An information-flow tracking system for realtime privacy monitoring on smartphones. In *OSDI* (2010), vol. 10, pp. 1–6.

[12] ENCK, W., OCTEAU, D., MCDANIEL, P., AND CHAUDHURI, S. A study of android application security. In *USENIX security symposium* (2011), vol. 2, p. 2.

[13] ENVATOMARKET. Android notification templates library. http://codecanyon.net/item/android-notification-templates-library/5292884. 2014.

[14] ERNST, M. D., JUST, R., MILLSTEIN, S., DIETL, W. M., PERNSTEINER, S., ROESNER, F., KOSCHER, K., BARROS, P., BHORASKAR, R., HAN, S., ET AL. Collaborative verification of information flow for a high-assurance app store.

[15] F-SECURE. F-secure : Internet security for all devices. http://f-secure.com, 2014.

[16] F-SECURE. Threat report h2 2013. Tech. rep., f-secure, http://www.f-secure.com/documents/996508/1030743/ Threat_Report_H2_2013.pdf, 2014.

[17] FENG, Y., ANAND, S., DILLIG, I., AND AIKEN, A. Apposcopy: Semantics-based detection of android malware through static analysis. In *SIGSOFT FSE* (2014).

[18] GILBERT, P., CHUN, B.-G., COX, L. P., AND JUNG, J. Vision: automated security validation of mobile apps at app markets. In *Proceedings of the second international workshop on Mobile cloud computing and services* (2011), ACM, pp. 21–26.

[19] GRACE, M., ZHOU, Y., ZHANG, Q., ZOU, S., AND JIANG, X. Riskranker: scalable and accurate zero-day android malware detection. In *Proceedings of the 10th international conference on Mobile systems, applications, and services* (2012), ACM, pp. 281–294.

[20] GRIFFIN, K., SCHNEIDER, S., HU, X., AND CHIUEH, T.-C. Automatic generation of string signatures for malware detection. In *Recent Advances in Intrusion Detection* (2009), Springer, pp. 101–120.

[21] HANNA, S., HUANG, L., WU, E., LI, S., CHEN, C., AND SONG, D. Juxtapp: A scalable system for detecting code reuse among android applications. In *DIMVA* (2012).

[22] HUANG, H., CHEN, K., REN, C., LIU, P., ZHU, S., AND WU, D. Towards discovering and understanding unexpected hazards in tailoring antivirus software for android. In *AsiaCCS* (2015), ACM, pp. 7–18.

[23] JING, Y., ZHAO, Z., AHN, G.-J., AND HU, H. Morpheus: automatically generating heuristics to detect android emulators. In *Proceedings of the 30th Annual Computer Security Applications Conference* (2014), ACM, pp. 216–225.

[24] KASSNER, M. Google play: Android's bouncer can be pwned. http://www.techrepublic.com/blog/it-security/-google-play-androids-bouncer-can-be-pwned/, 2012.

[25] KELION, L. Android adware 'infects millions' of phones and tablets. http://www.bbc.com/news/technology-31129797, 2015.

[26] LINDORFER, M., NEUGSCHWANDTNER, M., WEICHSELBAUM, L., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis-1,000,000 apps later: A view on current android malware behaviors. In *Proceedings of the the 3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security (BADGERS)* (2014).

[27] LINDORFER, M., VOLANIS, S., SISTO, A., NEUGSCHWANDTNER, M., ATHANASOPOULOS, E., MAGGI, F., PLATZER, C., ZANERO, S., AND IOANNIDIS, S. Andradar: Fast discovery of android applications in alternative markets. In *DIMVA* (2014).

[28] LU, L., LI, Z., WU, Z., LEE, W., AND JIANG, G. Chex: statically vetting android apps for component hijacking vulnerabilities. In *Proceedings of the 2012 ACM conference on Computer and communications security* (2012), ACM, pp. 229–240.

[29] NETWORKX. Python package for creating and manipulating graphs and networks. https://pypi.python.org/pypi/networkx/1.9.1, 2015.

[30] OBERHEIDE, J., AND MILLER, C. Dissecting the android bouncer. *SummerCon2012, New York* (2012).

[31] RASTOGI, V., CHEN, Y., AND ENCK, W. Appsplayground: Automatic security analysis of smartphone applications. In *Proceedings of the third ACM conference on Data and application security and privacy* (2013), ACM, pp. 209–220.

[32] RASTOGI, V., CHEN, Y., AND JIANG, X. Catch me if you can: Evaluating android anti-malware against transformation attacks. *Information Forensics and Security, IEEE Transactions on 9*, 1 (2014), 99–108.

[33] READING, I. . D. Google play exploits bypass malware checks. http://www.darkreading.com/risk-management/google-play-exploits-bypass-malware-checks/d/d-id/1104730?, 6 2012.

[34] REINA, A., FATTORI, A., AND CAVALLARO, L. A system call-centric analysis and stimulation technique to automatically reconstruct android malware behaviors. *EuroSec, April* (2013).

[35] REN, C., CHEN, K., AND LIU, P. Droidmarking: resilient software watermarking for impeding android application repackaging. In *ASE* (2014), ACM, pp. 635–646.

[36] SMALI. An assembler/disassembler for android's dex format. http://code.google.com/p/smali/, 2013.

[37] SQUARE, G. Dexguard. https://www.saikoa.com/dexguard, 2015.

[38] SQUARE, G. Proguard. https://www.saikoa.com/proguard, 2015.

[39] STATISTA. Statista : The statistics portal. http://www.statista.com/, 2014.

[40] STORM, A. Storm, distributed and fault-tolerant realtime computation. https://storm.apache.org/.

[41] VIDAS, T., AND CHRISTIN, N. Sweetening android lemon markets: measuring and combating malware in application marketplaces. In *Proceedings of the third ACM conference on Data and application security and privacy* (2013), ACM, pp. 197–208.

[42] VIDAS, T., TAN, J., NAHATA, J., TAN, C. L., CHRISTIN, N., AND TAGUE, P. A5: Automated analysis of adversarial android applications. In *Proceedings of the 4th ACM Workshop on Security and Privacy in Smartphones & Mobile Devices* (2014), ACM, pp. 39–50.

[43] VIRUSTOTAL. Virustotal - free online virus, malware and url scanner. https://www.virustotal.com/, 2014.

[44] VIRUSTOTAL. Virustotal for android. https://www.virustotal.com/en/documentation/mobile-applications/, 2015.

[45] WALENSTEIN, A., AND LAKHOTIA, A. *The software similarity problem in malware analysis*. Internat. Begegnungs-und Forschungszentrum für Informatik, 2007.

[46] WEICHSELBAUM, L., NEUGSCHWANDTNER, M., LINDORFER, M., FRATANTONIO, Y., VAN DER VEEN, V., AND PLATZER, C. Andrubis: Android malware under the magnifying glass. *Vienna University of Technology, Tech. Rep. TRISECLAB-0414-001* (2014).

[47] WU, C., ZHOU, Y., PATEL, K., LIANG, Z., AND JIANG, X. Airbag: Boosting smartphone resistance to malware infection. In *NDSS* (2014).

[48] YAN, L. K., AND YIN, H. Droidscope: seamlessly reconstructing the os and dalvik semantic views for dynamic android malware analysis. In *In USENIX rSecurity 12'*.

[49] YAN, P. A look at repackaged apps and their effect on the mobile threat landscape. http://blog.trendmicro.com/trendlabs-security-intelligence/a-look-into-repackaged-apps-and-its-role-in-the-mobile-threat-landscape/, 7 2014. Visited on 11/10/2014.

[50] ZHANG, F., HUANG, H., ZHU, S., WU, D., AND LIU, P. Viewdroid: Towards obfuscation-resilient mobile application repackaging detection. In *Proceedings of the 7th ACM Conference on Security and Privacy in Wireless and Mobile Networks (WiSec 2014). ACM* (2014).

[51] ZHANG, Q., AND REEVES, D. S. Metaaware: Identifying metamorphic malware. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual* (2007), IEEE, pp. 411–420.

[52] ZHANG, Y., YANG, M., XU, B., YANG, Z., GU, G., NING, P., WANG, X. S., AND ZANG, B. Vetting undesirable behaviors in android apps with permission use analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (2013), ACM, pp. 611–622.

[53] ZHENG, M., LEE, P. P., AND LUI, J. C. Adam: An automatic and extensible platform to stress test android anti-virus systems. In *Detection of Intrusions and Malware, and Vulnerability Assessment* (2013), pp. 82–101.

[54] ZHOU, W., ZHOU, Y., GRACE, M., JIANG, X., AND ZOU, S. Fast, scalable detection of piggybacked mobile applications. In *CODASPY* (2013).

[55] ZHOU, W., ZHOU, Y., JIANG, X., AND NING, P. Detecting repackaged smartphone applications in third-party android marketplaces. In *Proceedings of the second ACM conference on Data and Application Security and Privacy* (2012), ACM, pp. 317–326.

[56] ZHOU, Y., AND JIANG, X. Dissecting android malware: Characterization and evolution. In *Security and Privacy (SP), 2012 IEEE Symposium on* (2012), IEEE, pp. 95–109.

[57] ZHOU, Y., WANG, Z., ZHOU, W., AND JIANG, X. Hey, you, get off of my market: Detecting malicious apps in official and alternative android markets. In *NDSS* (2012).

[58] ZORZ, Z. 1.2info. http://www.net-security.org/secworld.php?id=15976, 11 2013. (Visited on 11/10/2014).

8 Appendix

Appstore	# of malicious apps	# of total apps studied	Percentage	Country
Anzhi	17921	46055	38.91	China
Yidong	1088	3026	35.96	China
yy138	828	2950	28.07	China
Anfen	365	1572	23.22	China
Slideme	3285	15367	21.38	US
AndroidLeyuan	997	6053	16.47	China
gfun	17779	108736	16.35	China
16apk	4008	25714	15.59	China
Pandaapp	1577	10679	14.77	US
Lenovo	9799	68839	14.23	China
Haozhuo	1100	8052	13.66	China
Dangle	2992	22183	13.49	China
3533_world	1331	9886	13.46	China
Appchina	8396	62449	13.44	China
Wangyi	85	663	12.82	China
Youyi	408	3628	11.25	China
Nduo	20	190	10.53	China
Sogou	2414	23774	10.15	China
Huawei	148	1466	10.1	China
Yingyongbao	272	2812	9.67	China
AndroidRuanjian	198	2308	8.58	China
Anji	3467	41607	8.33	China
AndroidMarket	1997	24332	8.21	China
Opera	4852	61866	7.84	Europe
Mumayi	6129	79594	7.7	China
Google	30552	401549	7.61	US
Xiaomi	832	12139	6.85	China
others	2377	38648	6.15	China
Amazon	59	1001	5.89	US
Baidu	831	21122	3.93	China
7xiazi	898	26195	3.43	China
Liqu	394	26392	1.49	China
Gezila	30	5000	0.6	China

Table 5: App Collection & Malware in Different Markets.