You Can Promote, But You Can't Hide: Large-Scale Abused App Detection in Mobile App Stores

Zhen Xie*, Sencun Zhu*†, Qing Li‡, Wenjing Wang‡

*Dept. of Electrical Engineering and Computer Science, Pennsylvania State University
†Dept. of College of Information Sciences and Technology, Pennsylvania State University
‡Blue Coat Systems, Inc.
Email: {zhenxie, szhu}@cse.psu.edu, {qing.li, wenjing.wang}@bluecoat.com

ABSTRACT

Instead of improving their apps' quality, some developers hire a group of users (called *collusive attackers*) to post positive ratings and reviews irrespective of the actual app quality. In this work, we aim to expose the apps whose ratings have been manipulated (or *abused*) by collusive attackers. Specifically, we model the relations of raters and apps as biclique communities and propose four attack signatures to identify malicious communities, where the raters are collusive attackers and the apps are abused apps. We further design a linear-time search algorithm to enumerate such communities in an app store. Our system was implemented and initially run against Apple App Store of China on July 17, 2013. In 33 hours, our system examined 2,188 apps, with the information of millions of reviews and reviewers downloaded on the fly. It reported 108 abused apps, among which 104 apps were confirmed to be abused. In a later time, we ran our tool against Apple App Stores of China, United Kingdom, and United States in a much larger scale. The evaluation results show that among the apps examined by our tool, abused apps account for 0.94%, 0.92%, and 0.57% out of all the analyzed apps, respectively in June 2013. In our latest checking on Oct. 15, 2015, these ratios decrease to 0.44%, 0.70%, and 0.42%, respectively. Our algorithm can greatly narrow down the suspect list from all apps (e.g., below 1% as shown in our paper). App store vendors may then use other information to do further verification.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

Keywords

App Store; Abused Apps; Collusion Attack; Temporal Biclique Community;

1. INTRODUCTION

ACSAC '16, December 05-09, 2016, Los Angeles, CA, USA

DOI: http://dx.doi.org/10.1145/2991079.2991099

With the fast development of smart phones and tablets, application markets (e.g., Android, iOS) have already grown to be a huge marketplace. In each market (also called *app store*), there are hundreds of thousands apps (e.g., 1,500,000+ apps in iTunes by June 8, 2015 [**?**]) and tens of billions of downloads (e.g., 100 billion downloads in iTunes). Meanwhile, the competition to attract customers becomes more furious than ever. Driven by monetary reward, some developers try to take a shortcut, which is to promote their apps using illegal methods [23]. Due to the large number of ratings and reviews per app (e.g., 5,000 on average for iOS apps [20]), individual or small-group (e.g., friends) rating promotion does not work effectively in app stores. Consequently, many sites like BestReviewApp have emerged in recent years with their business focusing on manipulating app ratings and reviews. In this work, we call the raters hired by a rating manipulation company *collusive attackers*.

For the healthiness of app ecosystems (especially not to discourage honest developers), paid ratings and reviews are forbidden by app store vendors [4][12] and FTC (Federal Trade Commission) [11]. In spite of this, many apps were still found to be promoted with paid ratings and reviews [7] [22][23]. As such, catching the abused apps and the collusive attackers is an urgent task. However, because of the massive number of accounts (e.g., 800 million users in Apple App store by April 23, 2014 [**?**]), ad-hoc or manual detection mechanisms do not work well. Besides, reviews in mobile app stores are often very short. Existing detection mechanisms based on review content analysis for traditional stores like Amazon are not applicable in mobile app stores.

The goal of our work is to design an effective and efficient abused app detection mechanism, which involves two major problems. First, *how to identify abused apps and related collusive attackers accurately*? To answer this question, we establish four basic attack signatures, which reflect the effect of attacks (e.g., abnormal change of average ratings) and the collaboration of attackers (e.g., burstiness of biased ratings). Such collusive features can be barely generated by biased raters, individual attackers, or small groups. Second, *how to discover abused apps and collusive attackers efficiently?* Given the number of apps and accounts in a market, the connections between apps and collusive attackers are too huge to trace collusive attackers. We observe that a special mutual dependency relationship exists between the suspicious levels of rater groups and the apps they have rated – the more suspicious a rater group, the more suspicious its rated apps, and vice versa. Hence, we model the relations of raters and apps as biclique communities and leverage these attack

signatures and the mutual dependency to identify malicious ones, where the raters are collusive attackers and the apps are abused apps. Further, we design a linear-time search algorithm to iteratively enumerate adjacent communities one by one.

Based on our algorithm, we implement a tool to detect abused apps in an app store. In the initial study, we ran it against Apple App Store of China on July 17, 2013. In 33 hours, our tool examined on the fly 2,188 apps with 4,841,720 reviews and 1,622,552 reviewers. It reported 108 abused apps, among which 104 apps were finally confirmed to be abused. Therefore, the detection precision of our algorithm is about 96.3%. We also applied our tool to Apple Stores of China, United Kingdom (UK), and United States of America (USA) after Apple Inc. cleared up its iTunes store in August, 2013. Our study shows that the clearance largely purified Apple Store of China by reducing abused apps from 4.75% to 0.94%. The ratio of abused apps in Apple Store of UK is 0.92% and that of USA is 0.57%. In our latest checking on Oct. 15, 2015, the ratios of abused apps in China, UK, USA decrease to 0.44%, 0.70%, and 0.42%, respectively. Performance evaluation also shows that the average time our algorithm takes to inspect one community is 167.3 ms, if not counting the time for downloading app metadata information.

In summary, our work offer the following contributions.

- Problem Formulation: We offered a new problem formulation to a real-world problem existing in most mobile app stores.
- Algorithm & Tool: We presented a linear algorithm to narrow down the suspect list from all apps (e.g., below 1% as shown in our paper), which could be further verified by app store vendors.
- New Discoveries: Based on the abused apps reported by our algorithm, we revealed the status and structures formed by abused apps and collusive attackers in the current app stores. We also uncovered some attack behaviors including attack launch time, attack length, and attack review length. These new discoveries could lead to the design of new detection algorithms.

1.1 Scope of Research

In reality, there could be various ways of promotion. Some developers could ask several friends/relatives to give high ratings to their apps. This kind of promotion works better for a newly released app, but the impact of such ratings would be quickly diluted by ratings from benign users. Some others might buy positive ratings from sites like Appqibu, itunestop, SafeRankPro, BuyAppStoreReviews, BestReviewApp, which have emerged in recent years with their business focusing on manipulating app ratings and reviews. If only buying a few ratings, the value of these ratings will also be diluted quickly.

In this work, *we do not deal with small-scale promotion.* Instead, our design goal is to catch abused apps promoted by a relatively large number of reviews, e.g., over 100. Such attacks are launched by commercial attackers. Moreover, due to the complication of establishing the ground truth, instead of exposing all abused apps precisely, *our goal is to greatly narrow down the suspect list from all apps* (e.g., below 1% as shown in our paper). App store vendors may then use other information like purchasing history, geographical locations to do further verification.

2. PROBLEM FORMULATION

2.1 Preliminaries

Naturally, all the apps and their raters form a bipartite graph, denoted by $W(V \bigcup G, E)$, where V denotes the set of apps, G the set of their raters, and E the set of edges. Each edge $(a, u) \in E$ indicates user u rated app a. For example, as shown in Fig. 1, 12 apps, 27 raters and their ratings form a bipartite graph, where $V = \{I \sim XII\}$ and $G = \{1 \sim 27\}$.

To make profit quickly, some developers might get a large group of raters to promote their apps. We define a *collusion group*, $G^c \subseteq G$, as a group of raters who leave false ratings in order to manipulate apps' overall rating scores. In practice, all the members of a collusion group (e.g., all the accounts controlled by an app rating manipulation company) need not participate in all rating attacks. Therefore, to be more generic, we model the structure of a collusion group as a set of adjacent subgroups (denoted by $G^s \subseteq G^c$) where each subgroup works as a unit to attack an app. To be considered as a subgroup, *it must have the minimum size M_s and have co-rated at least M_a apps.* Here both M_s and M_a are system parameters we define.

To be considered as *adjacent*, two subgroups must share no less than M_{so} overlapped raters, where M_{so} is another system parameter we define. If two subgroups (e.g., G_i^s, G_j^s) have at least one *path* (i.e., they can reach each other through a series of *adjacent* subgroups), they will be considered belonging to the same collusion group. The *path* set between G_i^s and G_j^s is denoted by $Path(G_i^s, G_j^s, M_{so})$. Otherwise, we classify them into two different collusion groups and our algorithm will check them separately.

DEFINITION 1. *A* collusion group, *$G^c[M_s, M_{so}]$ is a group consisting of* adjacent *subgroups of minimum size M_s whose members collaborate in manipulating apps' overall ratings. That is, $\forall i, j,\ G_i^s \subseteq G^c[M_s, M_{so}],\ G_j^s \subseteq G^c[M_s, M_{so}]$,*

$$|G_i^s| \geq M_s, |G_j^s| \geq M_s \tag{1}$$

$$Path(G_i^s, G_j^s, M_{so}) \neq \emptyset \tag{2}$$

Accordingly, we further define *abused apps* as apps whose ratings have been manipulated by raters who are part of a collusion group. From this definition, we exclude the cases where rating manipulations are performed by individuals who do not form a collusion group.

Let $A(G_i^s)$ denote the set of apps rated by subgroup G_i^s, that is, $A(G_i^s) = \{a | \text{app } a \text{ rated by } G_i^s\}$. Each subgroup G_i^s and $A(G_i^s)$ form a biclique where every member has rated every app. For example, in Fig. 1, raters $3 \sim 12$ form a subgroup. With app II, they together form a biclique. Further, if taking all the co-rated apps into consideration, they will form a *maximal* biclique that is not a subset of any other biclique (e.g., in Fig. 1, raters $3 \sim 12$ and apps $II \sim IV$).

When launching an attack to an app (e.g., a), for effectiveness a subgroup often posts all its ratings in a short time [1]. Hence, the reviews and ratings would carry some *temporal* features. Considering the posting date of each rating, we next define the concept of *temporal maximal biclique* (TMB). Let E^s denote the edge set of a maximal biclique

[1]Posting an abnormally large number of ratings in one or two days would easily expose the manipulated apps, so the promotion market often requires hired reviewers to rate apps in a week or two, according to [23].

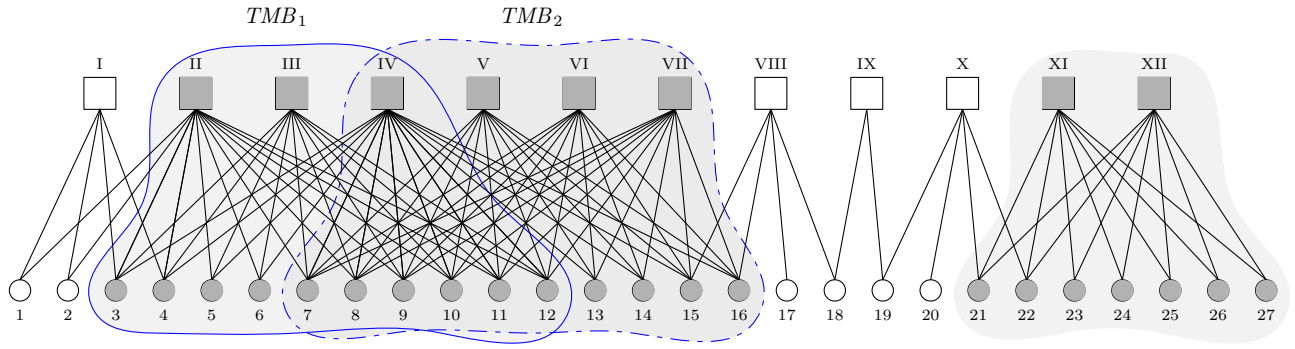


Figure 1: The process of building a partial bipartite graph. Each square node represents an app and each circle represents a rater.

and for each edge $(a, u) \in E^s$, let $T_{a,u}$ represent the time when u rates app a.

Definition 2. Temporal maximal biclique *(*TMB-$[M_a, M_s, \Delta t]$*) is a biclique formed by a subgroup of raters G_i^s and apps $A(G_i^s)$ they have co-rated, such that*

$$|A(G_i^s)| \geq M_a \tag{3}$$

$$|G_i^s| \geq M_s \tag{4}$$

$$\forall a \in A(G_i^s), u \in G_i^s, \exists (a, u) \in E^s \tag{5}$$

$$\forall a \in A(G_i^s), u \in G_i^s, \exists t_a \in \mathbb{R}, s.t. |t_a - T_{a,u}| \leq \Delta t, \tag{6}$$

Here, Eq. 3 defines the minimum size of the app set and Eq. 4 defines the minimum size of a subgroup. Eq. 5 defines a biclique where all the raters in the group have rated all the apps. Eq. 6 defines the temporal relationship among the ratings by each subgroup for each app a. Specifically, to each app a, all its ratings from the subgroup were posted in a $2\Delta t$ time window (centered at some moment t_a).

We call the larger group formed by multiple adjacent subgroups a *community*. The adjacent subgroups and their rated apps result in overlapping maximal bicliques, which form a *biclique community*. Similarly, multiple overlapping temporal maximal bicliques (TMBs) form a community, which is called a *temporal biclique community* (TBC).

Definition 3. Temporal Biclique Community *(*TBC-$[M_a, M_s, M_{ao}, M_{so}, \Delta t]$*) is a community formed by the union of TMBs, which each is at least as large as* TMB-$[M_a, M_s, \Delta t]$ *and can reach one another through a series of adjacent TMBs. Two TMBs are adjacent if they share at least M_{ao} apps and their common raters are at least M_{so}.*

Together there are five system parameters (i.e., M_a, M_s, M_{ao}, M_{so}, Δt) in our detection system. From the above definition we can see that *TMB*-$[M_a, M_s, \Delta t]$ defines the minimum size of maximal bicliques that may be considered as part of a biclique community. As shown in Fig. 1, the collusion group and its targets form a *TBC*-$[6, 14, 1, 6, \Delta t]$ biclique community where the bicliques (e.g., TMB_1, TMB_2, TMB_3) are larger than *TMB*-$[3, 10, \Delta t]$ and they share at least 1 app and 6 raters.

Finally, if $G_i^s \subseteq G^c[M_s, M_{s_o}]$, the TMB it has formed is also called a *malicious temporal maximal biclique* (m-TMB). The TBC formed by m-TMB(s) is called a *malicious TBC* (m-TBC). The apps in an m-TBC will be considered as abused apps and the raters in an m-TBC will be taken as *collusive attackers*. The ultimate goal of this work is to identify abused apps, through discovering m-TBC.

2.2 Problem Statement and Challenges

Given a bipartite graph $W(V \bigcup G, E)$, our goal is to discover abused apps by way of locating and identifying malicious temporal biclique community (*m-TBC*) while minimizing the false positive rate (i.e., the percentage of normal apps that have been misidentified as abused apps.). There are two major challenges in locating and identifying *m-TBCs*.

The first challenge is due to the need for *accurate detection* (i.e., detection effectiveness) under specific conditions and constraints such as incomplete bipartite graph, biased but normal raters, diverse behaviors of collusive attackers.

The second challenge is due to the need for *large-scale graph processing*. Given the massive number of accounts and apps in an app market, the bipartite graph we are processing is enormous. For example, in both Apple App Store and Google Play, there are over 1 million apps. There were about 800 million users in Apple App store by April 23, 2014 [**?**] and over 900 million users in Google Play [21]. Clearly, the demands on computation and memory are both extraordinary.

3. ATTACK SIGNATURE GENERATION

With respect to the challenge of identifying a malicious TBC, we first introduce the observations and insights from data crawled by AppWatcher [23]. Based on these insights, we further design four attack signatures to identify a TMB. Lastly, we introduce several rules to calculate the suspicious level of a TMB.

3.1 Observations and Insights

To identify malicious TMBs, the key is to find effective signatures for discriminating malicious TMBs from benign ones. As defined in Def. 2, a TMB involves two subjects: apps and raters. Hence, we seek attack signatures that can differentiate abused apps from non-abused ones and attack signatures that can discriminate collusive attackers from benign ones. Previously, in AppWatcher [23], the authors have studied the underground market of trading app reviews, and in GroupTie [22], the authors have observed that some unique patterns of collusion behavior. In this section, we will describe some insights from the observations of GroupTie [22] and AppWatcher [23], which are helpful for generating attack signatures.

- Biased ratings are usually posted within a short time (mostly a week or two [23]), and meanwhile, promoted

apps had low ratings and relatively few raters before promotion. As a result, promotion attacks would cause both the number of raters and average ratings of an app increase suddenly. After the promotion, raters would diminish to normal and ratings would also drop. Therefore, as proved in GroupTie [22], weekly average ratings and weekly rater quantities would have strong correlation. That is, if no promotion happens, the weekly average ratings **for the same version of app** would have no or little relationship with the number of raters. But if being promoted, the overall weekly average ratings would strongly correlate with weekly raters' quantities.

- As promoted apps often had few ratings, posting bulk biased ratings in a short time would cause a bursty growth of high ratings. This bursty time could indicate the existence of collusive attackers.
- As biased reviewers often co-rate many apps, their collaboration would cause the burstiness to exist in other apps they promoted.
- Within the promotion time window, ratings from biased reviewers are most positive ones. Therefore, the proportion of positive ratings would be much higher than in other time periods.

3.2 Attack Signatures

Based on the above insights, we derive four important features of collusive attackers and abused apps as our attack detection signatures. Signatures S1 and S2 are the bases to discover the malicious TMBs (m-TMBs) formed by abused apps and collusion groups. S3 and S4 are to identify abnormal behaviors from the perspective of apps. Note that we are not using these signatures separately but integrate them to derive L_a, the suspicious level of each app, and L_g, the suspicious level of each group in a TMB.

3.2.1 S1: High Burstiness of Biased Ratings

Conceptually, *high burstiness of biased ratings* means that many biased ratings are posted in a short time interval. For the effectiveness of attacks, collusion groups (i.e., subgroups) often generate lots of biased ratings in a small attack window in order to improve the rating score quickly. As a result, it causes a burstiness of biased ratings, which does not show up again unless there exist other manipulation attacks against the same app. In Fig. 2, we depict the weekly rater numbers of app 525378313 for different rating scores across the total period of 60 weeks. We can observe that 5-star raters concentrate in a short time interval, while rater quantities for other rating scores (i.e., 1, 2, 3, 4) are distributed all over the lifetime.

Two questions arise here: *how to model this feature and discover it for a given app?* We will defer the technical details until Section. 4, but only give a summary here. Each occurrence of this feature is modeled as a TMB-$[1, M_s, \Delta t]$, which is a temporal maximum biclique where at least M_s reviewers have rated this app during a $2\Delta t$ time window. For a given app, our algorithms will output TMB-$[1, M_s, \Delta t]$s if any.

Certainly, this feature (i.e., TMB-$[1, M_s, \Delta t]$) may also occur in some normal apps (e.g., popular apps). Those raters of the TMB-$[1, M_s, \Delta t]$ may actually be independent raters who happened to have posted similar ratings at the same time. Hence, a single TMB-$[1, M_s, \Delta t]$ cannot tell us for sure whether these raters are collusive attackers. Hence, if an app contains no more than one such TMB, we will ignore it. We will pick another app that has not been inspected to repeat the searching for high burstiness of biased ratings. We will use the next signature (S2) to boost the detection accuracy by building a TMB-$[M_a, M_s, \Delta t]$ where $M_a > 1$.

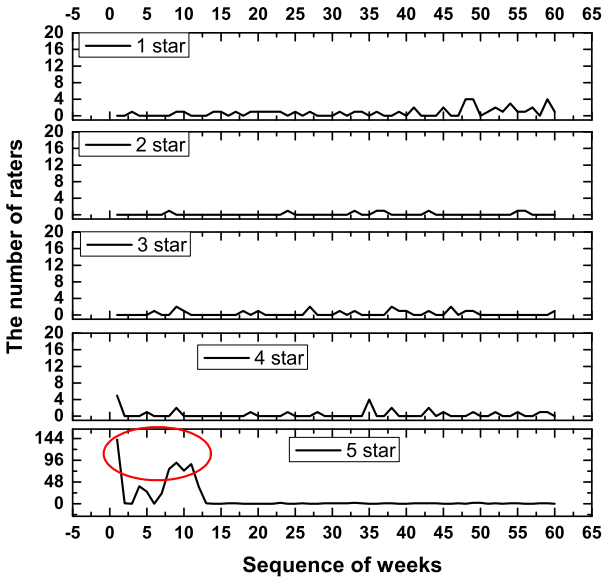


Figure 2: The HBBR feature of app 525378313.

3.2.2 S2: High Co-rating Frequency

Conceptually, the *high co-rating frequency* signature means that bursty biased ratings from the same group of users can be observed multiple times. This is because attack companies do not control an arbitrary number of accounts and hence they reuse the accounts to manipulate multiple apps. The higher frequency of co-rating, the more likely of being a collusion group. For example, Fig. 3.2.1 shows the number of common raters between app 474429394 and app 525378313 in each week[2]. The common raters existed in several (almost) consecutive weeks and no common raters were found in other times. This clearly indicates that they were organized to post ratings to these two apps.

If a group of raters are found to be the cause of multiple occurrences of *high burstiness of biased ratings* and the number of their co-rated apps is $n \geq M_a$, they will form a TMB-$[M_a, M_s, \Delta t]$. Hence, it is natural to model signature S2 as TMB-$[M_a, M_s, \Delta t]$. Now the question is: *how do we discover signature S2 from multiple occurrences of signature S1?* Again we will defer the technical details until Section. 4, but rather first discuss here the indication and application of this signature. While it is not unusual for a relative small number of users to co-rate two apps, it would be very suspicious for a big group of users to co-rate multiple apps in a short time window. Hence, in our system, we give a suspicious level score L_{tmb} to each TMB. A TMB will be labeled as a malicious one (i.e, a m-TMB) if its L_{tmb}

[2]Another strong evidence that they were promoted is that their common raters have consecutive IDs, as shown in Section. 5.2.2.

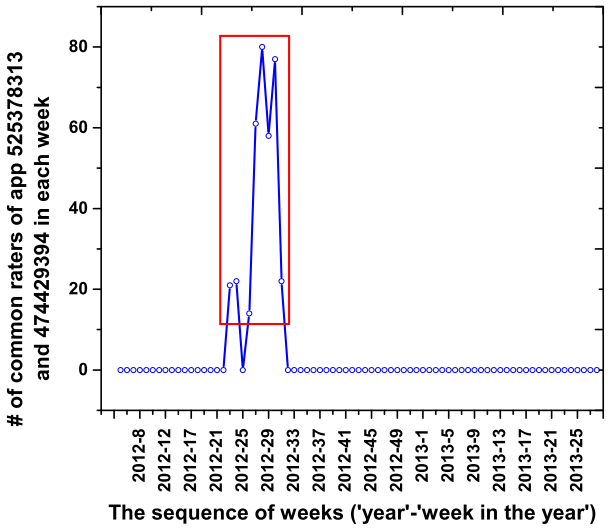


Figure 3: Number of common raters of app 525378313 and app 474429394 in the same week.

exceeds a certain threshold L_{th} (e.g., 0.25). Given m-TMBs, we will identify malicious TBCs, which each might be a rating manipulation company. The details will be presented in Section 4.

The next question is: *what if some benign users happen to give similar ratings in the same time period?* It is possible for benign users who have similar interests to install and rate a similar set of apps at a short time. However, the apps would rarely have the following features: *correlation coefficient abnormality* and *rating score distribution abnormality*. Hence, the suspicious level of the TBC they form would be low (i.e., indicating a benign TBC) and these benign users would not be easily mistaken as collusive attackers.

3.2.3 S3: Correlation Coefficient Abnormality

In GroupTie [22], it was found that for an app without collusive attackers, often its average rating *for the same version* in one week is almost equal to that in another week. This is because the distribution of app scores due to individual reviewers in different weeks are almost the same. In other words, the weekly average ratings do not depend on the number of raters, and hence theoretically there is no correlation between them (i.e., their correlation coefficient is around 0). On the other hand, because collusion groups manipulate app ratings by, for example, all giving 5 stars, the larger the collusive group is, the more the app rating will deviate from its true value. As such, the correlation coefficient between weekly average ratings and weekly rater numbers for the same version of app can indicate whether collusive attackers exist.

The *Correlation coefficient abnormality* signature is used to measure the abnormality of the relationship between the variations of average ratings and variations of rater numbers. If the value of correlation coefficient significantly deviates from 0, it does indicate the existence of collusion groups. However, if the value is around 0, it cannot guarantee there is no collusion group. For example, if there exists one group to promote the app and another group to demote it at the same time, correlation coefficient might not be able to reflect the existence of these two collusion groups. In this case, the other signatures can play more important roles in detecting collusion groups. Whichever case it is, the value of correlation coefficient will help determine L_a^0, the initial suspicious level of an app.

3.2.4 S4: Rating Score Distribution Abnormality

The *Rating score distribution abnormality* (RSDA) signature is used (in addition to *correlation coefficient abnormality*) in determining the initial suspicious levels of apps that fall in a TMB-$[M_a, M_s, \Delta t]$. In other words, this signature will not apply to apps not in any TMB-$[M_a, M_s, \Delta t]$. We will discuss its application in Section 3.3. For now we only focus on its meaning.

In different time points of a single app version, rating distribution (i.e., the number of raters in each rating score) is most likely similar to each other. However, for ratings within the attack time window, the percentage of positive ratings increases dramatically. This could be an indication that this app has been abused in this week. Unlike the signature S3, which examines the relation between weekly average ratings and rater numbers, S4 indicates whether the suspicious group of raters have actually caused the deviation of rating score distribution. As such, in suspicious time slots, we can detect whether the rating score distribution is changed by measuring RSDA.

To measure RSDA of an app, we calculate the ratio of number of raters giving positive ratings (i.e., 4 or 5 stars) to number of raters giving negative ratings (i.e., 1 or 2 stars). The ratio of each week, denoted by $r(i)$, is calculated in the following way.

$$r(i) = \frac{\#\text{ of raters giving 4 or 5 stars in } i\text{th week} + 1}{\#\text{ of raters giving 1 or 2 stars in } i\text{th week} + 1} \quad (7)$$

Note that we add 1 to both the numbers of 4 or 5 star raters and 1 or 2 star raters to prevent the division-by-zero problem. We then define the RSDA of the ith week, $R(i)$, by comparing $r(i)$ against the overall average. That is, $R(i) = r(i)/\frac{1}{n}\sum_{i=1}^{n} r(i)$. Formally, in the lifetime of an app, if there exists at least one $2\Delta t$ time interval, within which all the RSDAs have values $R(i)$ above a threshold H_t (e.g., 10), we will mark this app as an abused one. Such a RSDA is denoted as RSDA-$[\Delta t, H_t]$.

3.3 Calculation of TMB Suspicious Level

In our system, both subgroups and apps in a TMB will be assigned suspicious level ratings. There is a mutual dependency between these two ratings. If an app has been abused, there must exist at least one collusive subgroup responsible for the abuse. Therefore, the suspicious level of an app, L_a, depends on whether there exists a collusive subgroup among its raters. On the other hand, the suspicious level of a subgroup, L_g, depends on the suspicious status of their commonly rated apps. If multiple co-rated apps have been confirmed to be abused, this group is very likely to be a collusive subgroup.

To calculate the suspicious level of a TMB, we combine the four attack signatures. S1 and S2 are used to locate TMBs. The larger of the TMB, the more suspicious of the group. S3 and S4 are used to initialize the suspicious level of an app. Lastly, we use their mutual dependency to update each other iteratively.

Specifically, for a TMB-$[M_a, M_s, \Delta t]$, let the actual number of their co-rated apps be $n \geq M_a$ and the actual subgroup size be $m \geq M_s$. We will apply the following rules to calculate L_a and L_g.

- **Rule R1: on initial subgroup suspicious level L_g^0** If the size of a TMB is above an upper threshold M_u (e.g., 600), i.e., $m * n > M_u$, its subgroup suspicious level $L_g = 1$. If its size is below a lower threshold M_l (e.g., 300), i.e., $m * n < M_l$, $L_g = 0$. When $M_l \leq m * n \leq M_u$, L_g will be set as the average suspicious level of the n co-rated apps.
- **Rule R2: on initial app suspicious level L_a^0** If the suspicious level of an app has been calculated previously (note that this app might also belong to another TMB that was processed earlier), in the context of this TMB, L_a^0 of this app will be that value calculated earlier. Otherwise, if this app contains RSDA-$[\Delta t, H_t]$, $L_a^0 = 1$. If no such a RSDA exists, L_a^0 is set to the correlation coefficient value of this app (signature S3).

- **Rule R3: on updating L_a and L_g** Given a TMB, its L_g will be set as the average suspicious level of the n co-rated apps. After that, for each co-rated app, if its L_a is above L_g, no change will be made. Otherwise, if $L_a < L_g$, then $L_a = L_g$.

Rule R3 makes sure that the suspicious level of an app is nondecreasing. The rationale is: if an app is obviously abused, whether we find a suspicious subgroup or not we should not reduce its suspicious level. On the other hand, for a low suspicious app, if it is found rated by a highly suspicious group, its L_a should be raised to that of the updated L_g. Finally, we set the suspicious level of the TMB $L_{tmb} = L_g$.

4. LARGE-SCALE ABUSED APPS DETECTION

To address the challenge of large-scale graph processing, our algorithm searches for m-TBC *iteratively*, shown in Algorithm 1. The process of the algorithm starts with an app (e.g., app II in Fig. 1) which can be *any app* chosen randomly, either abused or benign. If it is an abused one, among all its raters, there must exist at least one collusive subgroup which has rated this app (e.g., raters 3∼12 in Fig. 1). Often, some benign users (e.g., raters 1 and 2) might have also posted similar ratings to the app as the attackers, so one cannot distinguish them without further evidence. However, for attackers, it is very likely that they work together again to abuse other apps. It is very unlikely for those benign users to give *similar ratings* (e.g., 5 stars) to *the same set of apps* at around *the same time period* as the collusive subgroup did. Therefore, after retrieving the information of all the raters (e.g., the rater set including app $1 \sim 12$) of the app (i.e., app II), we continue to obtain all the apps rated by each rater but keep only those having been co-rated by enough raters (e.g., ≥ 5) as the app set (e.g., app II∼VII). The rater set and the app set form a bipartite graph, which contains collusive subgroups and their targets.

Based on our definition, each collusive subgroup and its targets forms a TMB. To discover TMBs, we first enumerate all the maximal bicliques from the bipartite graph using algorithms like [15]. After that, we extract TMBs (if any) out of them. Specifically, for each maximal biclique, we group each rating (i.e., an edge) by the app version and rating type (i.e., positive or negative rating). For each group, we divide ratings into weeks and only keep ratings in bursty weeks (when weekly rater quantities are above the average). If the bursty rating posting times are within $2 * \Delta t$, the group will be reported as a TMB. The we calculate the suspicious level of each TMB using rules discussed in Section. 3.3 and identify malicious ones (i.e., m-TMB). For each m-TMB, we set each of its apps as the new starting app and repeat the entire process to look for *adjacent* m-TMB (e.g., TMB_2 enclosed by dashed line in Fig. 1). If no more m-TMB is found, the entire m-TBC would have been discovered. Next, we pick another (probably benign) app (e.g., app VIII) as the new starting app and repeat the process. If we find a small TMB instead of a community, we discard it and pick another app as the starting one (e.g., app IX) and repeat this process again. The process will be repeated until we find another abused app (e.g., app XI) and the whole process will start over. When all apps have been inspected once, the process will terminate and we get all the abused apps.

To catch collusive groups, current techniques [2][3][17] rely on the algorithm of Frequent Item Mining (FIM) [1], which requires the entire bipartite graph as input and is proved to be NP-hard [24]. In contrast, our approach starts from a very small bipartite graph built with the raters of a randomly chosen app and the apps co-rated by at least a certain number of raters[3] (i.e., M_s). Specifically, if we define a *step* as examining one community, the above mechanism ensures that our processing on a bipartite graph is a *forward walking*, which means each step goes to a new community with at least one uninspected app. Hence, the detection of all the malicious biclique communities only takes a limited number of *step*s. In other words, for a connected bipartite graph $W(V \bigcup G, E)$, our algorithm is able to detect all the malicious communities within $|V|$ *step*s, where $|V|$ is the number of apps in the entire market. In the worst case, our algorithm will examine all the ratings of an app. Hence, the complexity of our algorithm is $O(|V| * |G|)$.

5. EXPERIMENTAL ANALYSIS

5.1 Initial Experimental Settings

Our algorithm was implemented in JAVA 1.6. The experiments were run on a Ubuntu Server 12.04, equipped with a four-core Intel i7-2600k processor and 16G memory.

We applied our tool to inspect Apple App Store of China on July 17, 2013, setting all the parameters to the values mentioned in Section 3 and restated in Tab. 1. Due to the huge number of apps and reviews in the store, we only ran our tool for 33 hours and 31 minutes. Following the algorithm with a starting app 525378313, our tool examined $2,188$ apps with $4,841,720$ reviews and $1,622,552$ reviewers on the fly. These apps were from 16 categories out of total 20 categories.

5.2 Experimental Results

5.2.1 Malicious Temporal Biclique Communities

Out of $2,188$ apps and $1,622,552$ reviewers, our tool reported 57 malicious temporal biclique communities (m-TBCs),

[3]Based on our definition, apps co-rated by only a small number of raters will not be considered and hence are discarded.

Table 1: The parameters of the system and their default values in experiments.

Parameters	Default Values	Description
M_a	2	The minimum number of co-rated apps in a TMB, see Def. 2.
M_s	100	The minimum number of reviewers in a TMB, see Def. 2.
M_{ao}	2	The minimum number of apps shared by two adjacent TMBs, see Def. 3.
M_{so}	50	The minimum number of reviewers shared by two adjacent TMBs, see Def. 3
M_l	300	The minimum number of edges in a TMB to be considered, see Rule R1 in Section. 3.3.
M_u	600	The maximum number of edges in a TMB to be considered, see Rule R1 in Section. 3.3.
H_t	10	The threshold of RSDA, see Section. 3.2.4.
Δt	4 (weeks)	The half time window of temporal ratings, see Def. 2.
L_{th}	0.25	The threshold of L_{tmb}.
N_l	3000	The number of rating metadata to be crawled for an app, see Algorithm. 1.
N_u	15000	The minimum number of reviewers of an app to be considered as popular app, see Algorithm. 1.

Algorithm 1: Malicious Temporal Biclique Community Discovery

```
1  Procedure Discovering TBCs(W(V ⋃ G, E), App A)
2      Q← ∅, m-TBCs← ∅, m-TBC← ∅;
3      ENQUEUE(Q,A);
4      ExaminedAppList ← ∅;
5      while Q! = ∅ do
6          v ← DEQUEUE(Q);
7          ExaminedAppList.add(v);
8          S_g ← The most recent N_l raters of app v;
9          if |S_g| ≥ N_u or |S_g| < M_s then
10             continue;
11         S_v ← All the apps reviewed by raters in S_v;
12         W_mb ← Discover maximal bicliques in bipartite
           graph W(S_v ∪ S_g, E) by Alg [15];
13         m-TMBs ← Discover TMBs(W_mb). ;
14         for ∀ TMB in m-TMBs do
15             <S'_v, S'_g> ← TMB;
16             if m-TBC is ∅ then
17                 m-TBC ← TMB;
18             else
19                 for ∀TMB' ∈ m − TBC do
20                     <S''_v, S''_g> ← TMB';
21                     if |S'_v ∩ S''_v| ≥ M_ao and |S'_g ∩ S''_g| ≥ M_so
                       then
22                         m-TBC.add(TMB');
23                     else
24                         m-TBCs.add(m-TBC);
25                         m-TBC← ∅;
26             ENQUEUE(Q,(S_v \ ExaminedAppList));
27     return m-TBCs;
```

which contain 156 malicious temporal maximal bicliques (m-TMBs) in total. These m-TBCs consist of 108 apps and 17,621 reviewers.

As illustrated in Fig. 4, each subgraph represents a m-TBC and each square node stands for a m-TMB. For an edge in a m-TBC, it connects two m-TMBs which share at least 2 apps (i.e., $M_{ao} = 2$) and 50 reviewers (i.e., $M_{so} = 50$). In the figure, three largest m-TBCs contain 75 m-TMBs out of the total 156 m-TMBs. One m-TBC consists of 6 m-TMBs and 8 m-TBCs contain 3 m-TMBs. The remaining 45 m-TBCs only have one m-TMB each.

For the three largest m-TBCs, they contain nearly 50% of all the m-TMBs. However, their members are not the most. The largest collusion group is indeed a m-TBC consisting of only one m-TMB which has 2,422 reviewers. The large number of m-TMBs in a m-TBC indicates the collusion group of the m-TBC has been divided into lots of smaller subgroups.

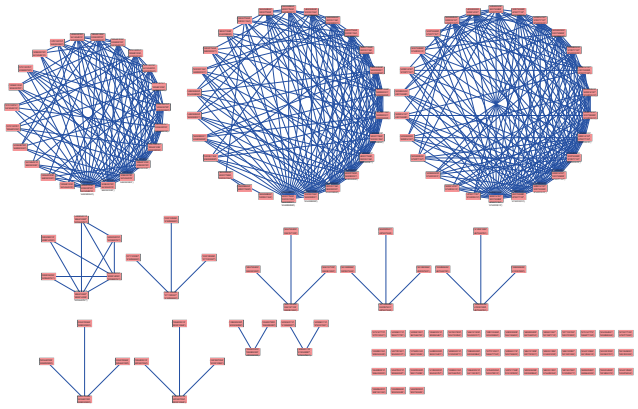


Figure 4: The structure of all the m-TBCs. Each m-TBC is formed by adjacent m-TMBs (represented by a rectangle) which share at least 2 apps and 50 reviewers. The text in a node is the app set of the m-TMB.

More edges between m-TMBs show more complex division and more complicated organization of its subgroups.

For the 45 m-TBCs which contain only one m-TMB each, their collusion groups were not split. It means their members work together each time. Some of them could also belong to a much larger collusion group whose subgroups were not tightly connected. They may be taken as separate collusion groups by our algorithm as defined in Def. 1.

5.2.2 *Validation and Detection Accuracy*

Precision is used to measure the accuracy of our algorithm. It is defined as the percentage of reported apps which are actually abused by some collusion groups.

Establishing the ground truth on whether these reported apps are actually abused by a collusion group is a challenge. It is very difficult to know even for Apple Inc. or Google, and that might be the reason why so many abused apps have not been flagged. Due to the huge volume of apps and reviews, it is almost impossible to manually integrate all the information and discriminate abused apps from benign ones. We have asked three students to verify the reported apps; however, they failed to find clues by way of reading reviews because reviews of benign and abused apps are both short as shown in Fig. 6(c). We also contacted Apple Inc. and asked them to help verify these apps, but they did not respond.

For verification purpose, we therefore wrote a few small programs to extract much stronger abnormal features of collusion groups or abused apps, which are specific in some app stores instead of being applicable to all the app stores. Specifically, we look for the existence of *Consecutive Reviewer Ids* (i.e., review ids of some group members are very close because of applying these accounts sequentially in a short time), *Exact Review History* (i.e., some group members have reviewed *exactly* the same set of apps), and *Concentrated Review Distribution* (i.e., most reviews of an app are posted in a very short time). If any of them were found, the group would be highly likely a collusion group and the apps they attacked would highly likely be abused apps.

Consecutive Reviewer Ids means the reviewer ids of some group members are very close. Even though we do not know the exact mechanism for id generation in iTunes, it is very unlikely that so many reviewers with close ids have rated the same set of apps. For example, the following reviewers have commonly reviewed the two apps 474429394 and 525378313 mentioned in Section. 3.2.2.
"212525736, 212525739, 212525752, 212525762, 212525765, 212525781, 212525784, 212525788, 212525795, 212525800, 212525808, 212525811, 212525825, 212525827, 212525834, 212525847, 212525851, 212525857, 212525860, 212525864"[4]

Our conjecture is that iTunes assigns ids sequentially, and a rating manipulation company somehow has successfully registered many accounts almost concurrently. To our observation, GooglePlay is likely to assign ids in a more random way. Hence, this consecutive reviewer ids feature is not general, although it does help us verify abused apps in this specific experiment.

Exact Review History means some group members have reviewed *exactly* the same set of apps. Considering the huge number of apps in an app store, it is uncommon for so many reviewers to rate exactly the same set of apps. For example, we find a group of 171 reviewers who all have rated and only rated apps with ids (499814295, 525948761, 485252012, 496474967, 499805269). This feature is much stricter than the high co-rating frequency feature used in our algorithm. For high co-rating frequency, we look for reviewers who have co-rated more than M_{ao} apps. By exact review history we further check whether some of these reviewers have rated the same set of apps.

Concentrated Review Distribution means that most reviews of an app are posted in a very short time. Normally, the reviews of an app are distributed through its life time. For benign apps, it is uncommon for most reviews being posted within several weeks. However, for an abused app, this could happen if its most reviews are posted by collusion groups. This feature is much stronger than the high burstiness of biased ratings (HBBR) feature in our algorithm. In HBBR, our algorithm only looks for consecutive weeks that have more reviews than other weeks and, moreover, these weeks must have more than M_{so} reviews in total. The concentrated review distribution feature further checks whether these reviewers account for most of the reviewers.

To show the effectiveness of these three features, we also randomly choose another set of 108 benign apps (e.g., apps developed by famous companies outside China) and studied these three features by taking all the reviewers of each app as a group. Since the feature of concentrated review distribution needs to label several weeks as suspicious ones, we first sort all the weeks by the reviewer quantity and label the top 4 weeks as the candidates.

Table 2: The distribution of reported apps and benign apps regarding consecutive reviewer ids (CRI), exact review history(ERH), and concentrated review distribution(CRD).

Features	Thresholds	Reported apps	Benign apps
CRI	0.1	51	0
ERH	0.2	38	0
CRD	0.8	83	5

As illustrated in Fig. 5, these three features of reported apps are drastically different with those of benign apps. If setting the thresholds as those in Tab. 2, 51 reported apps have the feature of consecutive reviewer ids, 38 apps have exact review history, and 83 apps have concentrated review distribution. In contrast, only five benign apps have the feature of concentrated review distribution. Therefore, without considering these 4 apps as abused, the detection precision is 96.3%.

5.2.3 Attack Behavior

Next, we unveil the attack behaviors of collusion groups based on the 104 abused apps reported in Section 5.2.2.

Attack Launch Time For each version of an app, we sort all the weeks by time and look for the weeks when the collusion group posted the first review. As illustrated in Fig. 6(a), the attacks are often launched at the first two weeks after updating a new app version. In a new version, some functions might have been introduced and the developer might want to promote the app timely for greater impact.

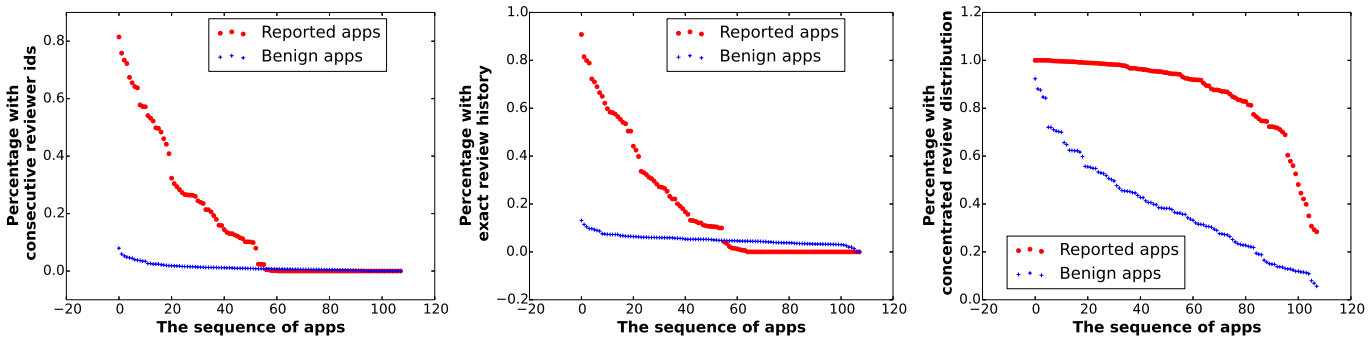
Attack Length The attack length means how long attacks last. As shown in Fig. 6(b), most attacks only last one week. Some of the attacks could last for two weeks and a few of them last longer. It would be more effective to promote the rating score if the attack length is shorter, but at a higher risk of being exposed.

Attack Review Length The attack review length is the number of word segmentations in reviews posted by attackers. From Fig. 6(c), we can see that the review lengths from benign users and attackers are both very short. The average length of reviews in word segmentations is 6.8 from benign users and 7.7 from attackers. One possible explanation is: since most reviews are posted through smartphones, the review lengths are much shorter than those in traditional stores. Note that this phenomena makes any review content-based detection mechanism infeasible.

5.2.4 Finding Abused Apps in Other App Stores

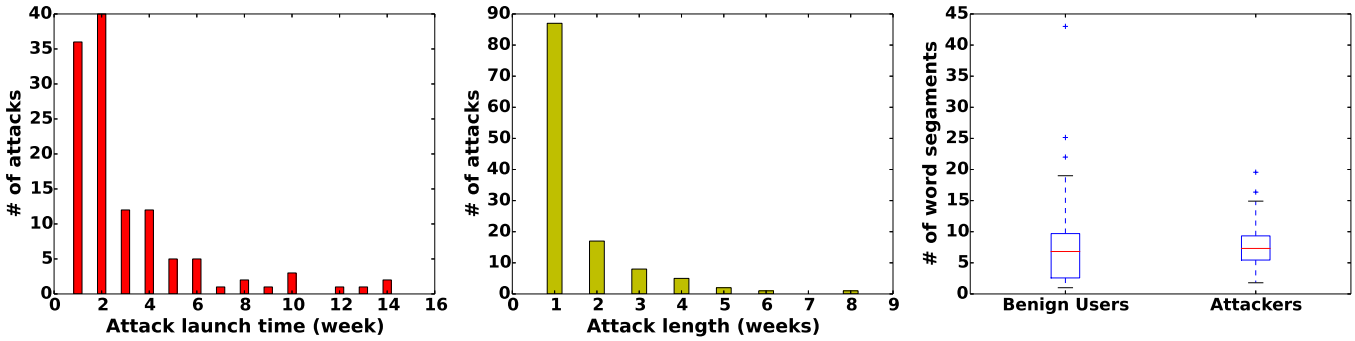
We observed that Apple launched an action of clearing up App Stores in August, 2013, following "App Store Review Guidelines" [13]. Note that the criteria for the clearance included not only just abused apps but also other types of apps like spyware. Since the exact reason for the removal of each app was not given, we could not use the results as the ground truth for our evaluation. Nevertheless, we consider it partially useful to indicate the detection capability of our algorithm.

[4]The review history of a reviewer can be accessed through the link https://itunes .apple.com/WebObjects/MZStore.woa/wa/viewSoftware?id=xxx. Make sure to change your location to China and replace xxx with a reviewer id.



(a) Consecutive reviewer ids (b) Exact review history (c) Concentrated review distribution

Figure 5: Strong abnormal features of benign apps and reported apps



(a) Attack launch time (b) Attack length (c) Attack review length

Figure 6: Attack Behaviors

After the clearance action, we ran our tool against Apple App Store of China, USA, and United Kingdom, respectively. Due to the huge volume of meta data, we only inspected $10k - 23k$ apps from each app store, as presented in Tab. 3. Here, the first dataset (i.e., dataset I) was the one crawled before the clearance and used in our earlier experiment. For the two iTunes China datasets, as we can see from Tab. 3, the proportion of abused apps dropped from 4.75% to 0.94% after the clearance. From the post-clearance data sets, we can see that, iTunes China and UK had 0.94% and 0.92% abused apps, respectively. The app abuse status of iTunes USA was less severe than the other two app stores, and its abused apps were about 0.57% of all apps.

In our followup check on Oct. 15, 2015, many abused apps we reported two years ago still existed in the app store. For example, 67 abused apps from dataset I still existed in the store. One possible explanation is that Apple does not have effective techniques to detect abused apps. Note that, by only removing some abused apps and some reviews, the clearance did not hurt the attackers very much, because they can involve in future manipulation attacks. Our algorithm can help app stores identify the collusion groups and take further actions against them.

5.2.5 *Performance Overhead*

The performance measurement was run on a Ubuntu Server 12.04, equipped with a four-core Intel i7-2600k processor and

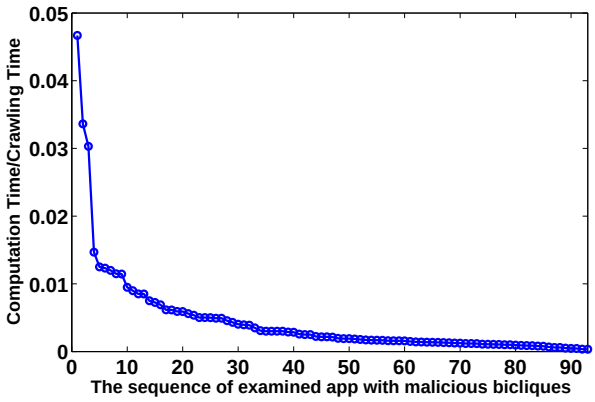


Figure 7: The comparison of crawling time and computational time.

16G memory. Our tool was set to one thread in order to make measurements comparable.

The total running time includes crawling time of retrieving app reviews from the websites and computational time of detecting malicious temporal biclique communities (m-TBCs). Since our tool has to establish connections with the websites (i.e., app stores), the crawling time is much longer than computational time. In reality, the crawling time is hundreds of times more than the computational time. For

Table 3: The dataset we have crawled from iTunes for detecting abused apps. The data of iTunes China* was crawled at July, 2013 before the clearance of Apple Inc. and other data sets were crawled after the clearance. For abused apps, we also checked their existence on 10/15/2015 and denote them with $^+$.

Dataset	App Store	# of Apps	# of Reviews	# of Reviewers	# of Abused Apps(%)	# of Attackers(%)
I	iTunes China*	2,188	4,841,720	1,622,552	104 (4.75%),[67(3.1%)]$^+$	16,747 (1.03%)
II	iTunes China	21,831	9,320,807	2,903,206	206 (0.94%),[97(0.44%)]$^+$	27,345 (0.94%)
III	iTunes UK	10,579	11,761,493	3,962,057	97 (0.92%),[74(0.70%)]$^+$	11,456 (0.29%)
IV	iTunes USA	23,615	18,925,438	5,818,330	135 (0.57%),[100(0.42%)]$^+$	17,174 (0.30%)

example, as illustrated in Fig. 7, of all the 93 examined apps, the computational time is only a very small fraction of the crawling time, less than 1% in 84 cases and less than 2% in 91 cases. We believe the performance bottleneck of crawling can be easily avoided when our tool is run by platform vendors like Apple Inc., Google Inc.. This is because they may access local databases storing all the meta data, and the time needed to load from a database is far less than that from websites. Therefore, we do not count crawling time when evaluating the performance of our tool in the next.

We ran our tool three times starting from the same starting app and recorded their average computational time. On average, each m-TBC has $2,449$ reviewers and 4.6 maximal bicliques. The average computational time of inspecting one m-TBC is 167.3 ms.

Theoretically speaking, our tool could examine all the 1.5 million apps in Apple App store or in GooglePlay Store in 69.7 hours, excluding the crawling time. Moreover, the computational time could be much less if running the system in parallel by assigning starting apps in different countries.

6. DISCUSSIONS

Our algorithm is for discovering apps abused by collusion groups whose behaviors conform to one or more of the four attack signatures mentioned in Section. 3.2. Certainly, if knowing our algorithm, a collusion group may try more sophisticated strategies to evade the detection.

6.1 Adversarial Challenges

Sybil attack is a way that a single attacker can emulate the behavior of multiple users by forging multiple identities (i.e., accounts) [9]. If a collusion group is capable of forging enough accounts (e.g., millions), they can launch an attack using one set of accounts and never use these accounts again. In this way, these accounts would not form any collusion group. Lacking the connection between these accounts and the real collusion groups, our algorithm would be unable to identify these malicious accounts as collusion groups. Since many benign users only post one review as these accounts, it is hard to differentiate them only by their reviews. Nevertheless, ratings from these accounts would probably still introduce some abnormal attack signatures like *"correlation coefficient abnormality"* (signature S3). Moreover, if combining their identities with other information like device identities, IP addresses, which are hard to forge, such sybil attack could be largely limited. When deploying our algorithm, app store vendors can replace app store accounts with merged identities to mitigate this attack.

Slow attack is a strategy that slows down the speed of posting fake reviews in order not to leave attack signatures. For example, a collusion subgroup may spread their reviews in a time period larger than Δt so that they would not be exposed as HBBRs. Note that in our system, by default Δt is set to 4 weeks. To launch a successful slow attack, the fake reviews have to be posted in over 4 weeks. According to AppWatcher [23], typically fake reviews are required to be completed in two weeks. Moreover, Δt is a system parameter we choose, so the attacker cannot know it. For show attacks, we can increase Δt to capture them, although large Δt would decrease the precision.

Greedy attack is a strategy to control the size of collusive groups while maximizing the usage of each reviewer. Since our approach discovers attackers by looking for temporal bicliques (TMBs), a collusion group could avoid forming such a TMB to disguise their behaviors. As the minimal TMBs being inspected within $2\Delta t$ is at least M_a apps and M_s raters, the maximal TMB that a collusion group can form without being discovered is at most M_a apps and at most $M_s - 1$ raters while maximizing each member's ability. Then, at most $\frac{M_s-1}{2\Delta t}$ reviewers can join the attack to rate these M_a apps within each unit time. As such, their posting rates are limited by our algorithm. Moreover, we can always adjust the system parameters to fail such greedy attacks, which have to happen before our detection.

Shuffling attack is to shuffle all the members in order not to form static subgroups. For example, a collusion group consists of n members and it randomly selects m members to promote an app each time. Therefore, they can form $\binom{n}{m}$ different subgroups while any two subgroups share at most $m-1$ members. To avoid forming any TMB, they only assign tasks to the subset where any two subgroups only share $M_b - 1$ members. This subset includes at most $\lceil \binom{n}{M_b} / \binom{m}{M_b} \rceil$ different subgroups, which are the maximum number of apps this group can promote without being exposed. As we can see, if the group wants to promote as many apps as possible, the size of each subgroup (i.e., m) would be decreased to M_b, which makes its promotion less effective. On the contrary, if they try to post as many reviews as possible, the size of each subgroup would be far larger than M_b, which limits the number of apps they can abuse.

6.2 Possible Solutions

Our algorithm is capable of narrowing down the suspect list from a large number of groups. However, if knowing our algorithm, these groups can take various strategies as we mentioned above. Since some parameters can balance false positive rate and false negative rate, we can adjust them to capture more attackers at the cost of higher false positive rate. For example, if we increase the value of Δt, we could discover collusive groups who are doing slow attacks; if we decrease M_b, we could also find some collusive groups doing shuffling attacks. At the same time, some benign groups would be put into suspect list. Therefore, for practical reason, we suggest an adaptive deployment of our system by choosing parameters to capture as many suspects as possible. Then, our algorithm can refine the suspect list by

adjusting parameters one by one.

7. RELATED WORK

In this section, we discuss some related work on defending against collusion attacks.

Feature Engineering is a method to extract features of collusion groups and apply them to identify other groups. Mukherjee et al. [17] and Allahbakhsh et al. [2] proposed algorithms to identify collusion groups who send fake reviews using frequent itemset mining(FIM) [1]. Beutel et al. [6] proposed CopyCatch to catch collusive attackers which have lockstep behavior (i.e., launch attacks in a short time) when generating fraudulent "like" page in Facebook. Zhang et al. [25] proposed NeighborWatcher to trace comment spammers who post spam links on third party forums, blogs etc by exploiting the structure of spamming infrastructure.

Reputation Management System is a popular type of methods[8][19][14][26] to combat collusion attacks to reputation systems who promote or demote user's reputation collaboratively. Unlike P2P network, there is no direct relationship between app reviewers, which makes it inapplicable in mobile app store.

Clique/Maximum Independent Set Detection is a popular method in cloud computing to detect collusion groups who may send falsified results to break data integrity. Stab et al. [18] offered a method to detect collusive attackers by exploiting the frequency they work together in the majority or minority and how often they work in opposite groups. Du et al. [10] presented RunTest to pinpoint collusion groups which are outside the maximal cliques. Based on the assumption that malicious nodes collaborate much more frequently than honest nodes, Lee et al. [16] and Xie et al. [22] proposed algorithms to find cliques formed by those malicious nodes and they belong to the same collusion group. Maximum Independent Set was proposed by Araujo et al. [5] to fight against collusion attacks to voting pool.

Considering the huge size of apps and raters in an app store, algorithms like FIM, clique/maximum independent set detection are not efficient to detect abused apps as well as collusive attackers. As for reputation management, there is no direct relationship between reviewers and moreover, the majority rule is not applicable because of collusion groups could be the majority. Moreover, the features used in previous methods do not work in mobile app stores because attacker behaviors are too much different. For example, they may use review content to discriminate attackers from benign users; however, the contents are very short and highly similar between attackers and benign users in mobile app store. Hence, it would be unfair to compare different methods whose feature are not specified .

8. CONCLUSIONS

In this paper, we formalized the problem of abused app discovery in mobile app stores and presented four attack signatures to describe the behaviors of collusive attackers. Moreover, we proposed a linear algorithm to locate and identify abused apps and collusive attackers. Following the algorithm, we implemented a tool which can be easily deployed by app store vendors or a third party. We applied it to detect abused apps in Apple Store of China, United Kingdom, and United States of America. Our algorithm can greatly narrow down the suspect list from all apps (e.g., below 1% as shown in our paper). App store vendors may then use other information like geographical locations to do further verification.

9. ACKNOWLEDGEMENT

This work was partially supported by NSF grants: CCF-1320605 and CNS-1618684. We also appreciate all the anonymous reviewers for their helpful comments.

10. REFERENCES

[1] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[2] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Norman Foo, and Elisa Bertino. Detecting, representing and querying collusion in online rating systems. *CoRR*, abs/1211.0963, 2012.

[3] Mohammad Allahbakhsh, Aleksandar Ignjatovic, Boualem Benatallah, Seyed-Mehdi-Reza Beheshti, Norman Foo, and Elisa Bertino. Representation and querying of unfair evaluations in social rating systems. *Computers & Security*, 41(0):68 – 88, 2014. 8th IEEE International Conference on Collaborative Computing: Networking, Applications and Worksharing.

[4] Apple. https: //developer.apple.com/app-store/review/guidelines/.

[5] Filipe Araujo, Jorge Farinha, PatrÃąÂłcio Domingues, Gheorghe Cosmin Silaghi, and Derrick Kondo. A maximum independent set approach for collusion detection in voting pools. *J. Parallel Distrib. Comput.*, 71(10):1356–1366, 2011.

[6] Alex Beutel, Wanhong Xu, Venkatesan Guruswami, Christopher Palow, and Christos Faloutsos. Copycatch: stopping group attacks by spotting lockstep behavior in social networks. In *Proceedings of the 22nd international conference on World Wide Web*, pages 119–130, 2013.

[7] Rishi Chandy and Haijie Gu. Identifying spam in the ios app store. In *Proceedings of the 2nd Joint WICOW/AIRWeb Workshop on Web Quality*, WebQuality '12, pages 56–59, New York, NY, USA, 2012. ACM.

[8] Bled Electronic Commerce, Audun J?sang, and Roslan Ismail. The beta reputation system. In *In Proceedings of the 15th Bled Electronic Commerce Conference*, 2002.

[9] John R Douceur. The sybil attack. In *Peer-to-peer Systems*, pages 251–260. Springer, 2002.

[10] Juan Du, Wei Wei, Xiaohui Gu, and Ting Yu. Runtest: assuring integrity of dataflow processing in cloud computing infrastructures. In *Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security*, pages 293–304. ACM, April 2010.

[11] FTC. http://www.business.ftc.gov/documents/ bus71-ftcs-revised-endorsement-guideswhat-people-are-asking.

[12] Google. https://play.google.com/about/ developer-content-policy.html.

[13] App Store Review Guidelines:. https://developer.apple.com/app-store/review/.

[14] Sepandar D. Kamvar, Mario T. Schlosser, and Hector Garcia-Molina. The eigentrust algorithm for reputation management in p2p networks. In *Proceedings of the 12th international conference on World Wide Web*, pages 640–651. ACM, May 2003.

[15] Enver Kayaaslan. On enumerating all maximal bicliques of bipartite graphs. In *9th Cologne-Twente Workshop on Graphs and Combinatorial Optimization*, page 105, 2010.

[16] HyunYong Lee, JongWom Kim, and Kyuyong Shin. Simplified clique detection for collusion-resistant reputation management scheme in p2p networks. In *Communications and Information Technologies (ISCIT), 2010 International Symposium on*, pages 273–278, Oct 2010.

[17] Arjun Mukherjee, Bing Liu, and Natalie Glance. Spotting fake reviewer groups in consumer reviews. In *Proceedings of the 21st international conference on World Wide Web*, WWW '12, pages 191–200, New York, NY, USA, 2012. ACM.

[18] Eugen Staab and Thomas Engel. Collusion detection for grid computing. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 412–419. IEEE Computer Society, May 2009.

[19] Jianshu Weng, Chunyan Miao, and Angela Goh. An entropy-based approach to protecting rating systems from unfair testimonies. *IEICE - Trans. Inf. Syst.*, E89-D(9):2502–2511, September 2006.

[20] wiki. http://en.wikipedia.org/wiki/App_Store_(iOS).

[21] wiki. http://en.wikipedia.org/wiki/Google_Play.

[22] Zhen Xie and Sencun Zhu. Grouptie: toward hidden collusion group discovery in app stores. In *Proceedings of the 2014 ACM conference on Security and privacy in wireless & mobile networks*, pages 153–164. ACM, 2014.

[23] Zhen Xie and Sencun Zhu. Appwatcher: unveiling the underground market of trading mobile app reviews. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, page 10. ACM, 2015.

[24] Guizhen Yang. The complexity of mining maximal frequent itemsets and maximal frequent patterns. In *In KDD' 04: Proceedings of the tenth ACM SIGKDD International Conference on Knowledge Discovery and Data mining*, pages 344–353. ACM Press, 2004.

[25] Jialong Zhang and Guofei Gu. Neighborwatcher: A content-agnostic comment spam inference system. In *NDSS*. Citeseer, 2013.

[26] Runfang Zhou and Kai Hwang. Trust overlay networks for global reputation aggregation in p2p grid computing. In *Proceedings of the 20th international conference on Parallel and distributed processing*, pages 29–29. IEEE Computer Society, April 2006.