TOWARDS TRUSTWORTHY RATING SYSTEMS FOR MOBILE APP STORES

A Dissertation in
Computer Science and Engineering
by
Zhen Xie

© 2016 Zhen Xie

Submitted in Partial Fulfillment
of the Requirements
for the Degree of

Doctor of Philosophy

August 2016
The dissertation of Zhen Xie was reviewed and approved* by the following:

Sencun Zhu  
Associate Professor of CSE & IST  
Dissertation Advisor, Chair of Committee

Guohong Cao  
Professor of CSE

George Kesidis  
Professor of CSE & EE

Zhen Lei  
Associate Professor of EME

Mahmut Kandemir  
Professor of CSE  
Graduate Program Chair

*Signatures are on file in the Graduate School.
Abstract

With the fast development of smartphones and tablets, application (or app) stores (e.g., Android, iOS) have already grown to be a huge marketplace. Their vendors maintain rating systems to help smartphone users find best apps. Since positive ratings and reviews will very likely lead to more downloads and monetary benefit, the rating systems have become a target of manipulation by some app developers. These developers turn to websites like bestreviewapp.com and buy bulk ratings and reviews in order to promote their own apps or demote their rivals’ apps. To ensure app stores run healthily, app store vendors are forbidding paid ratings and reviews. However, they are lacking the knowledge of collusive reviewers, manipulated apps, and the entire underground market. Moreover, they also need efficient tools to detect collusive reviewers and manipulated apps.

In this thesis, we will introduce our study on how to detect collusion groups, how the underground market works, and how to efficiently detect manipulated apps.

First, we thoroughly analyzed the features of hidden collusion groups and proposed a novel method (GroupTie) to discover collusive reviewers. Since members of a hidden collusion group work together frequently and their ratings often deviate more from apps’ quality, collusive actions will enhance their relation over time. We built a relation graph named tie graph and detected collusion groups by applying graph clustering. Simulation results showed that the false positive rate of GroupTie approaches 0.31% and its false negative rate is around 9.29%. We also applied our method to detect hidden collusion groups among the reviewers of 89 apps in Apple’s China App Store. A large number of reviewers were discovered belonging to a large collusion group and several small groups.

Second, as a clear understanding of the underground review trading market was lacking, we studied this underground market and statistically analyzed the promotion incentives, characteristics of promoted apps and suspicious reviewers. To collect promoted apps, we built an automatic data collection system, AppWatcher, which monitored 52 paid review service providers for four months and...
crawled all the app metadata from their corresponding app stores. Finally, AppWatcher exposed 645 apps promoted in app stores and 29,680 apps promoted in some popular websites. The underground market was then reported from various perspectives (e.g., service price, app volume). We identified some interesting features of both promoted apps and suspicious reviewers, which are significantly different from those of randomly chosen apps. Finally, we built a simple tracer to narrow down the suspect list of promoted apps in the underground market.

Third, we designed an algorithm to expose the apps (i.e., abused apps) whose ratings have been manipulated by collusive attackers. Specifically, we modeled the relations of raters and apps as biclique communities and proposed four attack signatures to identify malicious communities, where the raters are collusive attackers and the apps are abused apps. We further designed 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 and 67 apps still exist on Oct. 15, 2015. 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 decreased 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 such as credit card numbers, geographical locations to do further verification.
# Table of Contents

List of Figures ix  
List of Tables xii  
Acknowledgments xiii

## Chapter 1

### Introduction

1.1 Background and Motivations ................................. 1  
1.2 Contributions ...................................... 3  
  1.2.1 Hidden Collusion Group Discovery ......................... 3  
  1.2.2 Underground Market Study .............................. 3  
  1.2.3 Large-Scale Abused App Detection in Mobile App Stores .. 4  
1.3 Dissertation Outline ................................... 5

## Chapter 2

### Related Work

2.1 Underground Market Study .................................. 6  
2.2 Proposed Approaches .................................... 6

## Chapter 3

### Toward Hidden Collusion Group Discovery in App Stores 10

3.1 Introduction ........................................... 10  
3.2 Preliminaries .......................................... 11  
  3.2.1 Rating Behaviors ........................................ 11  
  3.2.2 Attack Model ........................................ 12  
3.3 GroupTie ............................................... 14  
  3.3.1 Overview ............................................. 14  
  3.3.2 Tie and Tie Graph .................................... 16  
  3.3.3 App Quality Estimation ............................... 20
<table>
<thead>
<tr>
<th>Topic</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.3.3.1 Correlation Coefficient</td>
<td>20</td>
</tr>
<tr>
<td>3.3.3.2 Correlation Coefficient in Ideal Scenario</td>
<td>21</td>
</tr>
<tr>
<td>3.3.3.3 Correlation Coefficient in General Scenario</td>
<td>23</td>
</tr>
<tr>
<td>3.3.3.4 Apps’ Quality Estimation</td>
<td>26</td>
</tr>
<tr>
<td>3.3.4 Collusion Groups Discovery</td>
<td>28</td>
</tr>
<tr>
<td>3.4 Experiment and Result Analysis</td>
<td>31</td>
</tr>
<tr>
<td>3.4.1 Simulation Study</td>
<td>31</td>
</tr>
<tr>
<td>3.4.1.1 Experiment Settings</td>
<td>31</td>
</tr>
<tr>
<td>3.4.1.2 Results Analysis</td>
<td>32</td>
</tr>
<tr>
<td>3.4.1.3 Influence of Tie Threshold</td>
<td>34</td>
</tr>
<tr>
<td>3.4.2 Discovering Collusion Groups in App Store</td>
<td>35</td>
</tr>
<tr>
<td>3.4.2.1 Data Description and Detector Settings</td>
<td>35</td>
</tr>
<tr>
<td>3.4.2.2 Findings</td>
<td>37</td>
</tr>
<tr>
<td>3.5 Discussion</td>
<td>40</td>
</tr>
</tbody>
</table>

**Chapter 4**

Unveiling the Underground Market of Trading Mobile App Reviews

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1 Introduction</td>
<td>42</td>
</tr>
<tr>
<td>4.2 Data Collection Methodology</td>
<td>43</td>
</tr>
<tr>
<td>4.2.1 Service Provider Collection</td>
<td>43</td>
</tr>
<tr>
<td>4.2.2 Service Provider’s Meta Data Collection</td>
<td>44</td>
</tr>
<tr>
<td>4.2.3 Promoted App Collection</td>
<td>44</td>
</tr>
<tr>
<td>4.2.4 Ethical Consideration</td>
<td>45</td>
</tr>
<tr>
<td>4.3 App Store Promotion</td>
<td>46</td>
</tr>
<tr>
<td>4.3.1 Dataset</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2 Analysis of Service Providers</td>
<td>46</td>
</tr>
<tr>
<td>4.3.2.1 Service Launch Time</td>
<td>47</td>
</tr>
<tr>
<td>4.3.2.2 Service Price and Reward</td>
<td>47</td>
</tr>
<tr>
<td>4.3.2.3 Promoted App Volume</td>
<td>48</td>
</tr>
<tr>
<td>4.3.3 Analysis of Promotion Incentive</td>
<td>49</td>
</tr>
<tr>
<td>4.3.3.1 Promotion Incentives</td>
<td>49</td>
</tr>
<tr>
<td>4.3.3.2 Promotion Completion Time</td>
<td>50</td>
</tr>
<tr>
<td>4.3.4 Analysis of Promoted Apps</td>
<td>51</td>
</tr>
<tr>
<td>4.3.5 Analysis of Recruited Reviewers</td>
<td>53</td>
</tr>
<tr>
<td>4.3.6 Promoted App Tracer</td>
<td>55</td>
</tr>
<tr>
<td>4.4 Web Promotion</td>
<td>59</td>
</tr>
<tr>
<td>4.4.1 Dataset</td>
<td>59</td>
</tr>
<tr>
<td>4.4.2 Analysis of Service Providers</td>
<td>59</td>
</tr>
<tr>
<td>4.4.2.1 Service Launch Time</td>
<td>59</td>
</tr>
</tbody>
</table>
Chapter 5
Large-Scale Abused App Detection in Mobile App Stores 67
5.1 Introduction .............................................. 67
5.2 Problem Formulation .................................... 69
  5.2.1 Preliminaries ....................................... 69
  5.2.2 Problem Statement and Challenges ................. 71
5.3 Attack Signature Generation ............................. 72
  5.3.1 Observations and Insights ........................... 72
  5.3.2 Attack Signatures .................................. 73
    5.3.2.1 S1: High Burstiness of Biased Ratings .......... 73
    5.3.2.2 S2: High Co-rating Frequency .................... 75
    5.3.2.3 S3: Correlation Coefficient Abnormality .......... 76
    5.3.2.4 S4: Rating Score Distribution Abnormality ..... 77
  5.3.3 Calculation of TMB Suspicious Level ............... 77
5.4 Large-scale Abused Apps Detection ..................... 79
5.5 Experimental Analysis .................................. 82
  5.5.1 Initial Experimental Settings ....................... 82
  5.5.2 Experimental Results ................................ 82
    5.5.2.1 Malicious Temporal Biclique Communities ....... 82
    5.5.2.2 Validation and Detection Accuracy ............... 84
    5.5.2.3 Attack Behavior ................................ 86
    5.5.2.4 Finding Abused Apps in Other App Stores ....... 87
    5.5.2.5 Performance Overhead ........................... 88
    5.5.2.6 Parameters Study ............................... 89
5.6 Discussions ............................................. 93
  5.6.1 Adversarial Challenges .............................. 93
  5.6.2 Possible Solutions .................................. 95
5.7 Discussion .............................................. 95
5.8 Conclusions ............................................ 96
List of Figures

3.1 Collusion Attack. App1 and App4 are high quality apps and App2 is a low quality app. Several groups have been hired by the developers to promote App2 or hired by the competitors of App4 to demote App4. ........................................ 13
3.2 The probability of co-rating apps. Here the $x$-axis represents the number of commonly rated apps, and the $y$-axis represents $\ln(t(x))$ where $t(x) = \frac{\text{#pairs of reviewers co-rating} \ x \ \text{apps}}{\text{total pairs of reviewers}}$ ........................................ 15
3.3 DoA toward an app with quality 4.3 stars. ................................. 17
3.4 PoA toward an app with quality 2.1 stars. ................................. 18
3.5 The tie graph including all the nodes. Thick solid line represents a strong positive tie, thin solid line represents a strong negative tie, and dashed lines represent weak ties. ........................................ 19
3.6 One example of version updates.............................................. 23
3.7 The false positive .............................................................. 25
3.8 The distribution of CC values (rounded to nearest)). .................. 26
3.9 The positions an app’s quality $Q_k$ and its final rating $R_k$ if the correlation coefficient $r(V, Z) > 0$. ................................. 28
3.10 The tie graph of two collusion groups $A=\{1, 2, 3, 4\}$ and $B=\{6, 7, 8, 9, 10\}$. In this tie graph, all the weak ties and negative ties have been removed. ........................................ 29
3.11 An example of 4-clique-community. There are four 4-cliques which are $\{1, 2, 3, 4\}(c1)$, $\{1, 2, 4, 5\}(c2)$, $\{2, 4, 5, 6\}(c3)$ and $\{2, 5, 6, 7\}(c4)$. $c1$ is adjacent to $c2$ and they share three nodes. Similarly, $c2$ is adjacent to $c3$ and $c3$ is adjacent to $c4$. Any two 4-cliques can reach each other through such adjacent neighbors. ................................. 30
3.12 The FNR of GroupTie with PoA............................................. 33
3.13 The FNR and FPR of GroupTie with PoA. ............................. 35
3.14 The FNR and FPR of GroupTie with DoA. ............................ 35
3.15 Number of collusion groups with $k$. .................................... 38
4.1 The interactions between AppWatcher and review trading market. The detailed actions between them are as follows. A1: App developers upload their apps to app stores; A2: Some developers buy reviews from service providers; A3: The “website monitor” module monitors target service providers to collect paid reviews. If the service provider is recruiting reviewers online, “assignment collector” module will join as a passive reviewer to collect assigned app information; A4: the module of “app meta data crawler” crawlers paid reviews along with their dates; A5: “app meta data crawler” retrieves these promoted apps’ meta data from their corresponding app stores. 44

4.2 Service Providers Statistics 47

4.3 The maximum number of reviews that can be delivered in a single package. 48

4.4 Promotion incentives 49

4.5 The number of days needed to complete reviews for an app. 51

4.6 Average ratings from reviewers who have provided both ratings and text comments to iOS apps. 52

4.7 Total number of reviewers who have provided both ratings and text comments to iOS apps. 52

4.8 App category distribution 53

4.9 Number of reviewers (y-axis) who have reviewed certain number of promoted apps (x-axis) 54

4.10 Number of apps reviewed by one reviewer. 54

4.11 The average ratings to all apps from reviewers rated different number of apps. 55

4.12 Our promoted app tracer that starts from a set of promoted apps. 57

4.13 Average rating of suspicious apps. 57

4.14 Average ratings provided by suspicious reviewers. 58

4.15 App volume and service price of each service provider 60

4.16 Monthly changes of app volume and average ratings 61

4.17 Total number of reviewers of an app. 62

4.18 App category distribution (iOS) 62

4.19 The proportion of positive ratings in each website 64

4.20 Promotion incentives 64

5.1 The process of building a partial bipartite graph. Each square node represents an app and each circle represents a rater. 69

5.2 High burstiness of biased ratings (app:525378313) 74

5.3 High co-rating frequency (app: 525378313 and 474429394) 75
5.4 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.

5.5 Strong abnormal features of benign apps and reported apps

5.6 Attack Behaviors

5.7 The comparison of crawling time and computational time.

5.8 The influence of $M_a$, $M_s$, $M_l$, and $M_u$ on precision and performance.

5.9 The changes of precision and performance with the change of $H_t$.

5.10 The structure of all the m-TBCs. Each m-TBC is formed by adjacent m-TMBs (represented by a rectangle) which share at least one reviewer. The text in a node is the app set of the m-TMB.
List of Tables

3.1 Confusion matrix of rating behaviors ................................. 12
3.2 Possible reviewer roles .............................................. 12
3.3 The correlation coefficient (CC) of top ten apps. .................... 24
3.4 The relation between CC value intervals and the ratio of abused apps in each interval (demoted apps indicated by *) .................. 27
3.5 The average value of FPR and FNR under attacks like PoA, DoA and OA. ................................................................. 33
3.6 Abuse signs of apps ..................................................... 37
3.7 Collusion groups discovered by GroupTie. .............................. 39
4.1 Rating comparison between random apps and promoted apps. ...... 63
5.1 The parameters of the system and their default values. ............... 82
5.2 The distribution of reported apps and benign apps regarding consecutive reviewer ids (CRI), exact review history(ERH), and concentrated review distribution(CRD). ........................................ 86
5.3 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 are crawled after the clearance. For abused apps, we also checked their existence on 10/15/2015 and denote them with +. ................................. 87
I would like to express my deepest gratitude to my supervisor Dr. Sencun Zhu for his full support, expert guidance, understanding and encouragement throughout my study and research. Without his incredible patience and timely wisdom, this dissertation would not have been possible.

I am also grateful to Dr. Guohong Cao, Dr. George Kesidis and Dr. Zhen Lei for attending my Ph.D. dissertation committee and providing the great inspiration and insightful comments on my dissertation.

Finally, I would like to thank my wife, parents, sister and brother for their unconditionally support. I would not have been able to complete this thesis without their continuous love and encouragement.
Chapter 1 — Introduction

1.1 Background and Motivations

In recent years, smartphones have gown very fast and the sales of smartphones beat out feature phone sales globally in the year of 2013 [1]. Unlike feature phones, smartphones can install many third party apps to provide more functionalities. To better meet the demand of smartphone users, various apps have been developed targeting specific platforms like iOS, Android and they can be installed directly through smartphones. Meanwhile, the developers can earn money by selling their apps in the centralized marketplace created by platform vendors. This new ecosystem ensures more and more apps are released and at the same time, various new apps attract more customers to adopt smartphones and buy apps.

In each market (also called app store), there are hundreds of thousands apps (e.g., 1,000,000+ apps in iTunes by April 23, 2014 [2]) and tens of billions of downloads (e.g., 70 billion downloads in iTunes). One of the key factors in getting more downloads is positive ratings and reviews. First, customers often refer to the overall rating (probably also read some reviews) before downloading an app. Positive ratings will likely give customers more confidence to download an app. Second, each app store maintains ranking charts, which list high quality and popular apps. The apps at higher ranks have more opportunity to be exposed to customers. As their ranking algorithms (e.g., iTunes, GooglePlay [3]) take ratings as an important factor, more positive ratings will help an app reach a higher rank. Hence, app developers should be motivated to deliver higher quality apps to get more positive ratings and reviews. Nevertheless, driven by the monetary reward, some
developers would like to take a shortcut, which is to promote their apps through buying positive ratings and reviews.

Due to the large number of ratings and reviews per app (e.g., 5,000 on average for iOS apps [2]), individual or small-group (e.g., friends) rating promotion does not work effectively in app stores. Consequently, many sites like bestreviewapp.com, reviewfordev.com 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 and the apps whose ratings have been manipulated, abused apps.

For the healthiness of app ecosystems (especially not to discourage honest developers), paid ratings and reviews are forbidden by app store vendors (e.g., Clause 3.10 of “App Store Review Guidelines” [4], “Google Play Developer Program Policies” [5]). Moreover, according to “The FTC’s Revised Endorsement Guides” [6], paid reviews are illegal if endorsers fail to show whether they are paid to write the reviews (currently almost no reviewers state whether their reviews are paid or not). In spite of this, many apps are still promoted with paid ratings and reviews [7] [8]. As such, catching the abused apps and the collusive attackers is an urgent task.

However, because of the massive number of accounts (e.g., 400 million users in Apple App store by July 11, 2013 [2]), ad-hoc or manual detection mechanisms are not going to 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. Moreover, attack patterns are also different with those in traditional online stores. As a result, traditional techniques and models like Reputation System [9] [10] [11], Clique Detection [12] [13] [14], Maximum Independent Set [15] are not suitable anymore.

To build trustworthy rating systems for mobile apps, we have proposed a model (named GroupTie) to detect hidden collusion group, conducted the study of underground review trading market (named AppWatcher), and proposed a linear algorithm to detect abused apps. We briefly introduce these studies and list the contributions below.
1.2 Contributions

1.2.1 Hidden Collusion Group Discovery

To defend collusive misbehavior, we proposed a novel method in Chapter 3, GroupTie, to captures two essential features of collusive misbehavior. One is that they often leave extreme ratings (e.g., 5) to fulfill their goals. The other feature is that they often work together to attack the same set of apps because of their limited resources such as accounts. Since such collusive actions will enhance their relations over time, we built a graph to capture their pairwised relations. Simulation results showed that the false positive rate of GroupTie approaches 0.31% and its false negative rate is around 9.29%.

The main contributions are listed as below.

• **New Finding:** We have found the correlation coefficient between the variation of weekly average ratings and that of weekly reviewer quantity is a good indicator to find whether attackers exist or not. We also proved that the correlation coefficient becomes zero in ideal scenario.

• **Tie Graph:** We have proposed a novel method, i.e., tie graph, to capture the relationship between benign reviewers and collusive attackers. The tie graph would generate strong positive ties for collusive attackers, strong negative ties for attackers with different opinions, and weak tie for benign reviewers. In this way, the searching of collusive attackers becomes searching for subgraphs with strong positive ties.

• **Group Patterns:** We have analyzed collusive groups and discovered some interesting patterns like high review intensity, similar review history, consecutive review ids. These new patterns can be further used to design more advanced models.

1.2.2 Underground Market Study

Driven by huge monetary reward, some mobile app developers turn to the underground market to buy positive reviews instead of doing legal advertisements. These promotion reviews are either directly posted in app stores like iTunes and
Google Play, or published on some popular websites that have many app users. As a clear understanding of this app promotion underground market was lacking, we unveiled this underground market and statistically analyzed the promotion incentives, characteristics of promoted apps and suspicious reviewers. To collect promoted apps, we built an automatic data collection system, AppWatcher, which monitored 52 paid review service providers for four months and crawled all the app metadata from their corresponding app stores. We identified some interesting features of both promoted apps and suspicious reviewers, which are significantly different from those of randomly chosen apps. Finally, we built a simple tracer to narrow down the suspect list of promoted apps in the underground market.

Our key contributions can be summarized as follows.

- **Market Study:** To our best knowledge, we were the first to study the underground market of mobile app review promotion. Our results is very helpful to understand the current situation and growth of this market.

- **New Discoveries:** We had found some interesting features of both promoted apps and suspicious reviewers. These features are significantly different from those of randomly chosen apps and they are critical to design advanced algorithms for detecting promoted apps or reviewers.

- **Promoted Apps:** Except service providers, it is very difficult to get the ground truth of whether an app is a promoted app or not. All the promoted apps we had collected by joining as a recruited reviewer are definitely promoted.

- **Promoted App Tracer:** We designed a simple app tracer, which can be extended by adding heuristics we discovered, to narrow down the suspect list of promoted apps in the underground market.

### 1.2.3 Large-Scale Abused App Detection in Mobile App Stores

Considering the massive number of reviewers, a trustworthy rating system need an efficient algorithm to detect abused apps. We first modeled the relations of raters and apps as biclique communities and proposed four attack signatures to identify malicious communities, where the raters are collusive attackers and the apps are
abused apps. We further designed a linear-time search algorithm to enumerate such communities in an app store. Our system was implemented and run against Apple App Store of China, UK and USA.

In summary, our work offered 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). The suspect list can be further verified by app store vendors using other information such as credit card numbers, geographical locations. In addition, we implemented a tool, which can be used by app store vendors or any third party.

- **New Discoveries:** Based on the abused apps our algorithm reported, 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 algorithms.

### 1.3 Dissertation Outline

In summary, the rest of this dissertation is organized as follows: In Chapter 2, we present previous work on both underground market and proposed approaches to detect collusive attackers and abused apps. In Chapter 3, we present GroupTie, which is a model to detect collusive attackers by building a weighted graph and then searching for k-cliques from this graph. In Chapter 4, we offer the study on underground review trading market and present the characteristics of attackers, abused apps and reviews. In Chapter 5, we introduce a linear algorithm to efficiently and accurately label abused apps. Finally, Chapter 6 concludes our studies and presents future work.
Chapter 2 —
Related Work

2.1 Underground Market Study

Underground market is the center of connecting illegal merchants with special service providers including CAPTCHA solving systems [16], Twitter account spam [17], fake reviews, etc. These markets usually rely on some popular crowd-sourcing websites like Amazon’s Mechanical Turk, Freelancer. Amazon’s Mechanical Turk has been studied from various perspectives including worker demographics [18] [19], behavior research [20], marketplace [21]. Some other crowd-sourcing websites like Zhubajie [22], Sandaha [23], MicroBlogs [24] were also studied [25] from the perspective of campaigns on these websites. Freelancer was studied [16] from the role of freelance labor in web service abuse like account registration and verification, social network linking. Some forums was also studied in trading twitter spam accounts [17]. Unlike the above underground markets, app review market consists of service providers whose services only focus on writing mobile app reviews. They tried many sophisticated strategies to elude detection algorithms and some of them have built a reward system to recruit reviewers from the wild.

2.2 Proposed Approaches

Even though there is little research on the rating system of app stores except [7], traditional rating systems like Yelp, Amazon, IMDB, eBay have been studied a lot [26] [27] [28] on collusion attacks. They are also a significant threat to other applications like reputation system in P2P network, voting pool, cloud computing.
To combat the collusion attacks, lots of approaches have been proposed in these areas.

**Reputation System**  To defend against collusion attacks to the feedback system in P2P networks, Kamvar et al. [9] employed a global reputation system and relied on pre-trusted nodes in the network to preclude the network of collusion groups. Based on the power-law distribution of nodes’ degrees, Zhou et al. [10] proposed a method relying on power nodes instead of per-trusted nodes to deal with collusion attacks. PeerTrust [11] is a reputation system that gives malicious nodes lower reputation, assuming that the colluders always give high ratings to their partners and bad ratings to others. Unlike in P2P networks, reviewers in app stores only send ratings and reviews to apps and reviewers do not rate each other. Moreover, apps’ quality, which is the basis for evaluating reviewers’ rating, is unknown.

**Clique Detection**  In the context of cloud computing, collusion attack could break the integrity of the data collected from individual nodes. Du et al. [12] offered a mechanism to pinpoint the malicious service providers by detecting nodes outside maximum cliques. Stab et al. [13] presented a mechanism to detect collusion nodes by exploiting how often they work together in the majority or minority and how often they are in opposite groups. Lee et al. [14] designed an algorithm to find the cliques and furthermore, detect the malicious nodes and decrease their influence on the reputation. These methods were applied in grid computing, and after each task the result about whether a node cheats or not is easy to verify. However, reviewers’ ratings to apps are subjective and not binary either. Moreover, the full graph including all the nodes is too large to detect cliques efficiently. GroupTie only builds graph for the suspects who have strong ties, which is much smaller than the full graph.

**Maximum Independent Set**  Araujo et al. [15] proposed a maximum independent set approach for collusion detection in voting pools, aiming at classifying nodes as correct or incorrect. The approach first builds a vote against graph where two node in each edge voted against each other. Since each edge represents the disagreement between the nodes in two ends, they must belong to different groups, collusion group or honest group. Assuming that the largest plurality of nodes that do not vote against each other is correct, the detection of collusion groups is to find the maximum independent set from the votes against graph. Since collusion
group members in an app store could rate randomly to hide their roles, the ratings
to the non-target app of two members in a group could be totally different like
one is “1 star” and the other one is “5 star”. Thus, it is not suitable to apply
maximum independent set in apps store.

**Collaborative Filtering**

Collaborative Filtering (CF) is used in many recommender models and systems. Su et al. [29] classified CF techniques to memory-based CF [30] [31] [32] [33], model-based CF [34] [35] [36] and hybrid CF techniques [37] [38]. Neighborhood based CF is a representative technique of memory-based CF and it computes similarity between users or between items using Pearson correlation or vector cosine similarity, and the similarity is the basis for prediction. Cai et al. [33] modeled recommendation between friends in social networks based on neighborhood-based CF. Similar to CF which is to find nodes with the same
taste, collusion group discovery tries to find reviewers with the same purpose.
However, unlike users’ taste, users’ purpose is hidden under the ratings which is
critical for collusion group discovery. GroupTie is to estimate apps’ quality and
ties between reviewers to model deviations, which reflect reviewers’ purpose.

**Feature Engineering** is a method to extract features of collusion groups and
apply them to identify other groups. Mukherjee et al. [27] proposed algorithm to
identify collusion groups who send fake reviews. It first uses a frequent itemset
mining (FIM) [39] method to find candidate groups and employs an eight indica-
tors evaluation system to detect collusion groups. Allahbakhsh et al. [26] offered
a similar method employing six indicators to discriminate collusion groups with
honest groups and they also built a biclique to represent the relationship between
reviewers and products. Beutel et al. [40] proposed a method named 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. [41]
proposed NeighborWatcher to trace comment spammers who post spam links on
third party forums, blogs etc by exploiting the structure of spamming infrastruc-
ture.

These methods require the full knowledge of the user-item relationship to build
a complete bipartite graph and some of them are even NP-hard problem, making
them inapplicable to mobile app stores. Instead, AppWatcher is a search algorithm
that builds partial user-app bipartite graph on the fly to detect collusive behaviors
and its time complexity is linear to app number in a store.
Data Mining has not been applied to detect collusion groups, but it is a powerful tool to detect individual spam rater and it is suitable to detect collusion groups. It was first applied to detect spam opinion by Nitin et al. [42]. They studied the data from Amazon and detected individual fake reviews by way of supervised learning where duplicated reviews and near duplicated reviews were used as positive training data. Ott et al. [43] emphasized psycholinguistic methods and text analysis. Their approach took standard word and part of speech as the training data for supervised learning. Chandy et al. [7] proposed a latent class model to classify apps from iOS app store. They manually acquired apps with spam ratings as training data and test data to do supervised learning.
Chapter 3 —
Toward Hidden Collusion Group Discovery in App Stores

3.1 Introduction

In this chapter, we aim to discover hidden collusion groups and systematically analyze the problems in App Stores. Specifically, we are searching for answers to the following questions: how serious is the collusion group problem in current app stores? what are the characteristics of collusion groups? how to discover the hidden collusion groups efficiently and accurately? These questions are concerned by platform vendors. Correct answers will guide the vendors to clean up app stores and catch developers who have manipulated the feedbacks. It also can help customers decide whether to trust ratings and reviews of an app. As for developers, it is useful to monitor whether their apps’ feedbacks are manipulated by opponents or not.

To discover collusion groups, the most difficult task is to identify group members. Salehi-Abari et al. [44] defined collusion as “A collaborative activity that gives to members of a colluding group benefits they would not be able to gain as individuals”. The definition shows two essential characteristics of collusion groups. One is that members of a collusion group need to work together to fulfill their purposes. They have to rate together more frequently than independent reviewers because they only hold a small portion of total accounts. The other one is that their ratings consistently deviate to the same side from apps’ actual quality; for example, group members usually provide high ratings to promote an app or low
ratings to demote it. But for honest reviewers, their deviations are distributed more randomly.

Based on the above observations, we propose a novel method called GroupTie to identify group members. We first prove that correlation coefficient between the variation of average ratings in a time period (e.g., weekly) and the variation of its reviewer numbers in each period across different versions is equal to zero if all the reviewers rate independently. Then, we design a method to estimate apps’ quality considering their correlation coefficient. We further calculate the pairwise tie strength between users and build a tie graph to represent reviewers’ relation. By applying graph clustering on tie graph, we can classify all the nodes to several clusters which are considered to be collusion groups. Simulation results show that false positive rate of the method is up to 0.31% and false negative rate is about 9.29% in our simulation setting.

We have further applied our method to discover collusive reviewers in Apple app store. On May 21th, 2012, we collected 200 apps from its China market, including 100 top paid apps and 100 top free apps. GroupTie identified 8,853 reviewers among all 818,545 reviewers as possible collusion group members, which account for 1.08%. We also find that 3,677 reviewers belong to a large group, which contains 2,652 members, and several small groups.

3.2 Preliminaries

3.2.1 Rating Behaviors

For honest reviewers, we assume their ratings are independent. An independent rating means its variance is independent of other ratings. As for collusion group members, their ratings are correlated with respect to specific apps and usually deviate far away from the quality. For example, most of their ratings are the same, irrespective of the app’s actual quality.

All rating behaviors can be classified into three types w.r.t. apps’ quality. As shown in Table 3.1, they are honest rating, promoted rating and demoted rating. Honest rating, denoted by $H$, is to leave ratings around apps’ actual quality. Demoted rating, denoted by $D$, is to rate much below apps’ actual quality. Promoted rating, denoted by $P$, is to rate much above than apps’ actual quality.
### Table 3.1. Confusion matrix of rating behaviors

<table>
<thead>
<tr>
<th>Quality \ Rating</th>
<th>Positive</th>
<th>Negative</th>
<th>Demoted rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>High</td>
<td>Honest rating</td>
<td>Demoted rating</td>
<td></td>
</tr>
<tr>
<td>Low</td>
<td>Promoted rating</td>
<td>Honest rating</td>
<td></td>
</tr>
</tbody>
</table>

### Table 3.2. Possible reviewer roles

<table>
<thead>
<tr>
<th>role</th>
<th>Role description</th>
</tr>
</thead>
<tbody>
<tr>
<td>(0, 0, 0)</td>
<td>A reviewer who never provides ratings.</td>
</tr>
<tr>
<td>(0, 0, P)</td>
<td>A reviewer who always provides positive ratings.</td>
</tr>
<tr>
<td>(0, D, 0)</td>
<td>A reviewer who always provides negative ratings.</td>
</tr>
<tr>
<td>(0, D, P)</td>
<td>A reviewer who will only provide positive ratings to his partners’ apps and negative ratings to his opponents’ apps.</td>
</tr>
<tr>
<td>(H, 0, 0)</td>
<td>An honest reviewer.</td>
</tr>
<tr>
<td>(H, 0, P)</td>
<td>A reviewer who will provide positive ratings to his partners’ apps and honest ratings to others.</td>
</tr>
<tr>
<td>(H, D, 0)</td>
<td>A reviewer who will provide negative ratings to his opponents’ apps and honest ratings to others.</td>
</tr>
<tr>
<td>(H, D, P)</td>
<td>A reviewer who will provide positive ratings to his partners’ apps, negative ratings to his opponents’ apps and honest ratings to other apps.</td>
</tr>
</tbody>
</table>

The above rating behaviors form the basic behaviors set \( \{H, D, P\} \). We define role as a combination of different behaviors and the possible roles are enumerated in Table. 3.2. Every reviewer chooses a strategy on what kind of role she/he will follow each time. Similar to the player’s strategy in game theory [45], there are two types of strategies. One is the pure strategy, which means that a reviewer chooses a specific behavior from the behaviors set to follow. For example, to promote an app, the reviewer chooses a behavior like \( P \) and leaves a high rating. The other type of strategy is a mix strategy, which means that the reviewer chooses actions based on probabilities. For example, in the same scenario, the reviewer may choose the behavior \( H \) with probability 20%, \( D \) with probability 10% and \( P \) with probability 70%. Clearly, the pure strategy is a special case of the mix strategy. In this paper, we adopt the pure strategy to simplify the definition of attack models.

### 3.2.2 Attack Model

To promote an app, members of a collusion group will collaborate and rate high no matter how bad its real quality is. Similarly, to demote an app, they work together...
Figure 3.1. Collusion Attack. App1 and App4 are high quality apps and App2 is a low quality app. Several groups have been hired by the developers to promote App2 or hired by the competitors of App4 to demote App4.

to provide low ratings. What is more, some members could randomly act as honest reviewers to hide their purposes. These collective rating actions neglecting apps’ real quality are called collision attacks. According to the roles described in Section 3.2.1, the possible roles for collusion group members are \{(0, 0, P), (0, D, 0), (0, D, P), (H, 0, P), (H, D, 0), (H, D, P)\}, as shown in Table. 3.2.

In this paper, we classify all the collusion attacks into three types: promotion only attack, demotion only attack and orchestrated attack.

Promotion only attack (PoA) The collusion groups only promote apps’ rating by providing higher ratings than apps’ quality. The roles of group members could be \{(0, 0, P), (H, 0, P)\}.

Demotion only attack (DoA) The collusion groups only demote apps’ rating by providing lower ratings than the real quality. The roles of group members could be \{(0, D, 0), (H, D, 0)\}.

Orchestrated attack (OA) The collusion groups promote the target apps and demote the opponents of the target apps at the same time. The possible roles of group members are \{(0, 0, P), (0, D, 0), (0, D, P), (H, 0, P), (H, D, 0), (H, D, P)\}.

For example, as shown in Fig. 3.1, reviewer1 and reviewer2 take roles \((H, 0, P)\) and \((0, 0, P)\), respectively and they together perform PoA against App2. Reviewer3 and reviewer4 take roles \((H, D, P)\) and \((0, D, 0)\), respectively and they together perform DoA against App4. Reviewer5 and reviewer6 take roles \((H, D, P)\) and \((0, D, P)\), respectively and they perform both PoA and DoA. All the reviewers can form a group to single perform an OA.

In this paper, we aim to discover attackers that form collusion groups to achieve
their goals like raising average scores. We do not target at detecting other types of attackers like independent attackers, one-time attackers, or random attackers. Independent attacker is the reviewer that attacks an app on his/her own. Considering the massive number of reviewers for an app, independent attack has little influence on the app's overall score. One-time attackers are those promoting or demoting an app only once. For example, a developer might ask her friends and relatives to promote her app. Theoretically, there could also be attacks launched by a collusion group that behaves randomly. For example, they may demote an app in the first week and then promote it in the following weeks, for whatever reasons. We will not address these types of attacks in this work.

Note that in our work we do not claim the detection results are 100% accurate due to the extreme difficulty of getting the ground truth. Rather, our algorithm output can provide a much-narrowed-down suspect list for further investigation by app stores. App stores may use additional evidences which we do not have (e.g., information about reviewer accounts) to further pinpoint the abused apps and the collusion groups.

3.3 GroupTie

3.3.1 Overview

User accounts in app stores (e.g., Google Play or Apple iTunes) are bound with hardware devices (e.g., smartphones or tablets), so collusion groups are only likely to account for a small portion among all user accounts. Hence, they often have to collaborate in order to impact the rating of an app significantly. More specifically, their collaboration has two essential characteristics. First, for either promotion or demotion, they provide similar ratings deliberately and collectively. Their ratings significantly deviate from the real quality in the same way, e.g., giving 5 to low quality apps of 1 or 2 stars or the other way around. But for an honest reviewer, his ratings’ deviations are likely to be distributed randomly. Even though for some special apps, an individual user’s ratings may have similar deviation skewness with collusion group members, such a phenomenon would not happen for most of the apps.

Second, the collaboration of collusion group members could occur many times;
The number of common rated apps

The log value of the percentage

The probability of co-rating apps. Here the $x$-axis represents the number of commonly rated apps, and the $y$-axis represents $\ln(t(x))$ where $t(x) = \frac{\# \text{pairs of reviewers co-rating } x \text{ apps}}{\text{total pairs of reviewers}}$.

consequently, they have a much higher probability to rate the same set of apps than any other individual reviewers. In general, co-rating (i.e., rating the same apps) is not common. To see this, we collected 553,005 reviewers from Apple’s China App Store and we found that 0.163% pairs of the reviewers had two commonly rated apps and this number decreases to 0.012% for three commonly rated apps. As illustrated in Fig. 3.2, co-rating probabilities decrease sharply with the increase of the number of commonly rated apps.

Because of the above characteristics, the relations of group members are increasingly strengthened when they co-rate more and more apps, and eventually their relations become very close. Detection of group members is to detect such close relations. Here we outline the three challenges to be solved for collusion group discovery.

- First, how to define and quantify users’ relations and model their rating skewness?
- Second, how to discover collusion groups from pairwise relations?
- Third, how to improve detection accuracy under the influence of biased reviewers, adaptive attackers, version updating, etc.? For example, biased reviewers might be misidentified as attackers, hence increasing the false positive rate (See Section 3.3.2). Adaptive attackers might try hard to disguise
their roles and avoid being exposed. It would raise the false negative rate (See Section 3.3.2). Besides, developers usually update their apps and upload new versions for downloading. Some of the versions might include good features and thus will receive better reviews. Some of them might bear bugs accidently and get low ratings. This will increase the complexity of discovering collusion attacks (See Section 3.3.3.2).

Next, we propose a model to deal with the above three challenges. We borrow the concept of tie [46] to represent the relation between two reviewers and construct tie graph to represent the relations among all reviewers. Then, we propose a novel method to estimate apps’ quality. Finally, given tie graphs, we apply a \( k \)-clique communities clustering algorithm to discover collusion groups.

### 3.3.2 Tie and Tie Graph

**Definition.** Tie is the relation between two reviewers.

In this paper, **tie** represents the possible relation between two reviewers, where the term **relation** means the probability of two reviewers belonging to the same collusion group.

**Definition.** Deviation measures how much a rating deviates from an app’s real quality.

Formally, let the lower case letter denote a reviewer and the upper case letter denote an app. \( R(i, K) \) represents the rating of reviewer \( i \) to app \( K \) and \( Q(K) \) denotes the real quality of app \( K \) (we will show how to estimate \( Q(K) \) later). \( Dev(i, K) \) denotes the deviation of reviewer \( i \)'s rating to the quality of app \( K \) and it follows Eq. 3.1.

\[
Dev(i, K) = \begin{cases} 
0 & \text{\( R(i, K) \) does not exist} \\
R(i, K) - Q(K) & \text{Otherwise}
\end{cases}
\]  

**Definition.** Tie Strength measures the closeness of a relation.

The tie strength between reviewer \( i \) and reviewer \( j \) is denoted by \( Tie(i, j) \),
which follows Eq. 3.2

\[
Tie(i, j) = \sum_{K=0}^{N} Dev(i, K) \ast Dev(j, K),
\]  

(3.2)

where \( N \) is the number of commonly rated apps by \( i \) and \( j \). For members of a collusion group, their tie strengths would be high positive values due to their common rating actions. Based on tie strength, we classify ties into three types: strong ties, weak ties and absent ties by setting a tie threshold (e.g., 16 in our experiments). A strong positive tie has tie strength larger than the threshold, which indicates attraction between two reviewers. A strong negative tie has tie strength lower than the negative threshold (e.g., -16), which indicates repulsion between two reviewers. Weak ties are neutral ties with low tie strength (e.g., between -16 and 16), which indicate the relation is uncertain between two reviewers. Absent ties means the relation information between two reviewers is missing.

**Definition. Tie graph** is a weighted undirected graph which contains all the pairwise relations among all reviewers. The vertex set is all the reviewers and the edges are the ties between them.

For example, as shown in Fig. 3.3 and Fig. 3.4, four reviewers rate a high quality app and a low quality one, respectively. Among them, two reviewers are honest and the other two are members of a collusion group. Honest reviewers give ratings around the quality, and collusive ones perform promotion and demotion, respectively. We can calculate the tie strength for each pair of them.

**Figure 3.3.** DoA toward an app with quality 4.3 stars.

(a) Ratings

(b) Deviations
By following Eq. 3.2, we can calculate the tie strengths in Fig. 3.3.

\[ Tie(1, 2) = -0.21 \quad Tie(1, 3) = -2.31 \quad Tie(1, 4) = -1.61 \]
\[ Tie(2, 3) = -0.99 \quad Tie(2, 4) = 0.69 \quad Tie(3, 4) = 7.59 \]

![Diagram showing tie strengths](image)

**Figure 3.4.** PoA toward an app with quality 2.1 stars.

Similarly, we can get the tie strengths in Fig. 3.4.

\[ Tie(1, 2) = 0.11 \quad Tie(1, 3) = -3.19 \quad Tie(1, 4) = -3.19 \]
\[ Tie(2, 3) = -0.29 \quad Tie(2, 4) = -0.29 \quad Tie(3, 4) = 8.41 \]

Finally, by combining the tie strengths generated in both figures, we can get the overall tie strengths between each pair of the four reviewers. We can further generate the tie graph shown in Fig. 3.5.

\[ Tie(1, 2) = -0.1 \quad Tie(1, 3) = -5.5 \quad Tie(1, 4) = -4.8 \]
\[ Tie(2, 3) = -1.28 \quad Tie(2, 4) = 0.4 \quad Tie(3, 4) = 16 \]

Obviously, the tie between two members in a collusion group is the strongest and the other ties can be considered to be weak ties compared to the strong positive ties.

In practice, biased reviewers who always rate high/low may form two different types of strong ties: one case is that when they happen to rate the same set of apps with other biased reviewers, and the other case is when they rate the same set of apps as a collusion group do. In the first case, these biased reviewers form a group. Considering the chance of co-rating illustrated in Fig. 3.2, the size of the group formed by chance will not be very large, and we can easily filter them
out by the algorithm proposed in Section 3.3.4. In reality, biased reviewers are prone to rate popular apps. However, popular apps are usually not the targets of collusion groups because popular apps have many more reviewers than a collusion group has. Thus, the attack has little influence. But theoretically, the second case could happen and these biased reviewers would be misidentified as attackers. This case will cause the false positive rate to increase. However, if we further check other features mentioned in Section 3.4.2.1, like whether they have similar review history, consecutive reviewer IDs as attackers, other type of account information, biased reviewers can also be identified and removed from suspect lists.

On the other hand, if knowing GroupTie beforehand, some attackers might choose adaptive strategies to avoid forming strong ties. For example, each attack of a collusion group will increase tie strength between any two group members. For any two of them, they have to rate another app and leave two totally different ratings (i.e., one is high and the other is low) to annihilate the past impact on tie strength. As a result, if a collusion group has $n$ members, after co-rating a single app, they have to rate $\binom{n}{2}$ different apps to reset their tie strengths caused by such a co-rating, which is very difficult to execute in reality. For example, if a collusion group has 200 members, they have to choose another 20100 apps and build negative ties to cancel out previous positive ties. This would remarkably increase the cost of their organization and what is worse, this strategy could also lead to the risk of exposing their roles.

Figure 3.5. The tie graph including all the nodes. Thick solid line represents a strong positive tie, thin solid line represents a strong negative tie, and dashed lines represent weak ties.
3.3.3 App Quality Estimation

To calculate tie strength, we need to know apps’ real quality. However, in reality, app’s quality measurement is subjective, and its score displayed in the app store could have been severely skewed by attacks. To estimate apps’ quality, our idea is to first detect the abnormality of the rating behavior for an app and then remove such abnormality. In this section, we first prove that there is no linear relation between the average rating and the number of reviewers in each period (e.g., in each week) when no attacks exist. Then, we propose a method to estimate apps’ quality.

3.3.3.1 Correlation Coefficient

Correlation coefficient is a measurement of the linear relationship between two variables. If it is a positive value, the increase of one variable is likely to lead to the increase of the other one, and so is the decrease. That is, their variances are synchronous. Similarly, if the value is a negative one, their variances are opposite. Here we adopt Pearson’s correlation coefficient \[47\] to measure the linear relationship between two variables. It is denoted by \(r(Y, X)\) for variables \(Y\) and \(X\), as shown in Eq. 3.3.

\[
r(Y, X) = \frac{Cov(Y, X)}{\sigma_Y \ast \sigma_X} = \frac{E(Y - u_Y)(X - u_X)}{\sigma_Y \ast \sigma_X},
\]  

(3.3)

where \(Cov(Y, X)\) denotes the covariance of \(X\) and \(Y\). \(u_X\) and \(u_Y\) are the mean values of variables \(X\) and \(Y\), and \(\sigma_X\) and \(\sigma_Y\) represent the variances of \(X\) and \(Y\), respectively.

In this work, we divide time into periods of certain unit, e.g., weeks. Suppose an app has been updated a few times, and each version \(k\) has the lifetime of \(kn\) periods. Let \(\bar{Y}_{ki}\) be its average rating for period \(i\) w.r.t. version \(k\). Its average ratings are denoted by \(\bar{Y}_k = \{\bar{Y}_{k1}, \bar{Y}_{k2}, ..., \bar{Y}_{kn}\}\) with \(u_{\bar{Y}_k}\) being the overall average. The number of reviewers during each period is a random variable denoted by \(X_k = \{X_{k1}, X_{k2}, ..., X_{kn}\}\) with the overall average \(u_{X_k}\). Then, the correlation coefficient between average rating and number of reviews in each period is \(r(\bar{Y}_k, X_k)\).
3.3.3.2 Correlation Coefficient in Ideal Scenario

Intuitively, if all the reviewers of an app rate independently, then the average rating by one group of its reviewers should be about the same to that by another group if the app’s quality does not change (i.e., there is no version update). In other words, the increase or decrease of reviewers’ number should not change the average rating. From a different perspective, since each rating reflects the app’s quality, it can be considered as a sample on app’s quality. As these samples are independent and reflect the same app’s quality, it is reasonable to assume they follow the same distribution and this scenario is named the ideal scenario.

In the ideal scenario, it can be proven that no linear correlation exists between the average rating (sample mean) and the number of reviewers (sample size). Thus, we have the following theorem.

**Theorem 1.** If the ratings of an app w.r.t. version \( k \) are fully independent and have identical distribution, the correlation coefficient between average rating \( \bar{Y}_k \) and number of its reviewers \( X_k \) in each period is equal to zero when \( X_k \) is large enough.

**Proof.** Let us first divide time into periods of certain unit, e.g., weeks, and suppose there are totally \( n \) periods for an app. The average rating of each period is a random variable denoted by \( \bar{Y}_k = \{\bar{Y}_{k1}, \bar{Y}_{k2}, ..., \bar{Y}_{kn}\} \). The mean value of \( \bar{Y}_k \) is denoted by \( u_{\bar{Y}_k} \). The number of reviewers during each period is a random variable denoted by \( X_k = \{X_{k1}, X_{k2}, ..., X_{kn}\} \) with mean value \( u_{X_k} \).

Let \( Y_{ki} = \{Y_{ki1}, Y_{ki2}, ..., Y_{kim}\} \) denote the sequence of independent and identical distributed ratings in the \( i \)th period having mean \( u \) and variance \( \sigma^2 \). \( \bar{Y}_{ki} \) is the average rating in this period, which is expressed in Eq. 3.4. The number of reviewers in \( i \)th period is denoted by \( Y_{ki} \).

\[
\bar{Y}_{ki} = \frac{Y_{ki1} + Y_{ki2} + ... + Y_{kim}}{m} \quad (3.4)
\]

According to the central limit theorem, \( \bar{Y}_{ki} \) follows a normal distribution \( N(u, \frac{\sigma}{\sqrt{m}}) \) given that \( m \) is large enough. That is to say, if \( Z_{ki} = \bar{Y}_{ki} | X_{ki} \), \( Z_{ki} \) will follow a normal distribution \( N(u, \frac{\sigma}{\sqrt{m}}) \) and \( u = E(Z_{ki}) = E(\bar{Y}_{ki}) = E(\bar{Y}_k) = u_{\bar{Y}_k} \).
probability density of \( Z_{ki} \) is denoted by \( f(Z_{ki}) \) and \( f(Z_{ki}) = f(\bar{Y}_{ki}|X_{ki}) \).

\[
\text{Cov}(\bar{Y}_k, X_k) = E[(\bar{Y}_k - u\bar{Y}_k)(X_k - uX_k)] = E(\bar{Y}_kX_k) - uX_kE\bar{Y}_k - u\bar{Y}_kEX + uX_ku\bar{Y}_k
\]

\[
E(\bar{Y}_kX_k) = E(X_k(\bar{Y}_k|X_k)) = \sum_{i=1}^{n}(X_{ki}p(X_{ki})\sum_{j=1}^{m}((\bar{Y}_{kj}|X_{ki})p(\bar{Y}_{kj}|X_{ki})))
\]

\[
= \sum_{i=1}^{n}(X_{ki}p(X_{ki})(\bar{Y}_{ki}|X_{ki})p(\bar{Y}_{ki}|X_{ki})) = \sum_{i=1}^{n}(X_{ki}p(X_{ki})) \int_{Z_{ki}}(Z_{ki}f(Z_{ki}))dZ_{ki}
\]

\[
= \sum_{i=1}^{n}(X_{ki}p(X_{ki}))E[Z_{ki}] = u\bar{Y}_k \sum_{i=1}^{n}(X_{ki}p(X_{ki})) = u\bar{Y}_kuX_k
\]

Hence, \( \text{Cov}(\bar{Y}_k, X_k) = E(\bar{Y}_kX_k) - uX_ku\bar{Y}_k = 0 \).

Furthermore, according to Eq. 3.3, Pearson’s correlation coefficient \( r(\bar{Y}_k, X_k) = \frac{\text{Cov}(\bar{Y}_k, X_k)}{\sigma_{\bar{Y}_k}\sigma_{X_k}} = 0 \).

In practice, the quality of an app might be changed due to version updates. As a result, the average rating of one group is probably different to another group who rates over another version. For example, before updating a version with good features, the developer did a lot of advertising. Now for the new version, its average rating, even without promotion attack, could be improved. Another example is mentioned at the beginning that a version of an app is updated with bugs. Honest reviewers will leave low ratings to this version even though reviewers will rate high to other versions. In this cases, the correlation coefficient between average ratings and number of reviewers across different versions is not always equal to zero. To eliminate the influence of version updates, we will need to normalize the average ratings. Specifically, we will redefine correlation coefficient as the relation between the variation of average ratings and the variation of its reviewer numbers in each period across different versions.

Fig. 3.6 shows how to calculate the variation of average ratings and the variation of reviewer quantities. Here, an app has been updated three times and thus it has three versions. For both version 1 and version 2, reviewers rated the app for five weeks. For version 3, only four weeks have ratings. Each rectangular in the figure
represents one week. We first calculate the average ratings (e.g., $\bar{Y}_{11}$ for the first week of version 1) and the number of reviewers in each week (e.g., $X_{11}$). For version 1, we then calculate the overall average of the weekly average ratings (e.g., $\bar{u}_{\bar{Y}_1} = (\bar{Y}_{11} + \bar{Y}_{12} + \bar{Y}_{13} + \bar{Y}_{14} + \bar{Y}_{15})/5$) and the overall average of weekly reviewer numbers (e.g., $\bar{u}_{X_1} = (X_{11} + X_{12} + X_{13} + X_{14} + X_{15})/5$). Let $V_{ki}$ be the variation of the average rating in period $i$ w.r.t. version $k$, then $V_{ki} = \bar{Y}_{ki} - \bar{u}_{\bar{Y}_k}$. Similarly, we can derive $Z_{ki}$, which is the variation of the number of reviewers in period $i$ w.r.t. version $k$, as $Z_{ki} = X_{ki} - \bar{u}_{X_k}$.

Formally, let $V$ represent the variation of average ratings of all the versions where $V = \{V_{11}, V_{12}, ..., V_{ki}, ...\}$ and $Z$ represents the variation of reviewer numbers of all the versions where $Z = \{Z_{11}, Z_{12}, ..., Z_{ki}, ...\}$. Based on Theorem 1, we can further derive the following lemma.

**Lemma 1.** If the ratings of an app are fully independent and have identical distribution, the correlation coefficient between the variation of average ratings $V$ and the variation of reviewer quantity $Z$ in each period is equal to zero.

### 3.3.3.3 Correlation Coefficient in General Scenario

Unlike in the ideal scenario, in general scenario honest reviewers and collusion groups may coexist. For honest reviewers, they usually download apps and write comments after playing them for a while directly from mobile devices. Their experiences are independent and they usually do not refer to other reviews while
Table 3.3. The correlation coefficient (CC) of top ten apps.

<table>
<thead>
<tr>
<th>Rank</th>
<th>App Id</th>
<th>App Name</th>
<th>CC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>483583569</td>
<td>Mobilocation</td>
<td>0.414</td>
</tr>
<tr>
<td>2</td>
<td>362949845</td>
<td>Fruit Ninja</td>
<td>-0.175</td>
</tr>
<tr>
<td>3</td>
<td>449735650</td>
<td>Where’s My Water?</td>
<td>0.023</td>
</tr>
<tr>
<td>4</td>
<td>439615801</td>
<td>Plants vs. Zombies (Chinese version)</td>
<td>NA</td>
</tr>
<tr>
<td>5</td>
<td>414664715</td>
<td>Order &amp; Chaos @Online</td>
<td>-0.008</td>
</tr>
<tr>
<td>6</td>
<td>491231653</td>
<td>Richman 4 fun</td>
<td>-0.088</td>
</tr>
<tr>
<td>7</td>
<td>400973408</td>
<td>Asphalt 6: Adrenaline</td>
<td>0.036</td>
</tr>
<tr>
<td>8</td>
<td>508720652</td>
<td>Quick Call Divert</td>
<td>NA</td>
</tr>
<tr>
<td>9</td>
<td>435728194</td>
<td>Shou Ji Ling Sheng</td>
<td>0.249</td>
</tr>
<tr>
<td>10</td>
<td>449595696</td>
<td>Office Assistant Pro</td>
<td>0.082</td>
</tr>
</tbody>
</table>

leaving comments. Besides, even though human herding behavior [48] will influence their decision on buying an app, it has no explicit impact on review contents which express their own experience. For these reasons, the independency of user reviews are still held in general scenario. However, the existence of collusion groups will likely change the correlation coefficient of an app.

Since the correlation coefficient in the ideal scenario is equal to zero, its non-zero value in general scenarios could indicate the existence of collusion attacks as well as the type of collusion attacks (i.e., promotion or demotion). For example, for the top 10 paid apps in Apple App Store of China (as of May 21, 2012), we calculated their correlation coefficients (CC) by setting the time period to one week, as listed in Table 3.3.2. We found that the app named “Mobilocation” (Rank #1) has the highest CC score of 0.414 and the app named “Shou Ji Ling Sheng” (Rank #9) has the second highest CC score (0.249). We confirmed that these two apps were indeed promoted because of their low quality. They were reported to abuse the rating system in some websites [49] [50] and “Mobilocation”¹ has been removed from Apple App Store before May 7, 2013. We further plot the relationship between variance of weekly average rating and variance of reviewer quantities in each week for the first, second, and ninth apps, as shown in Fig. 3.7 From Fig. 3.7(a) and Fig. 3.7(c), it is obvious that the variation of weekly average rating of this app “Mobilocation” and “Shou Ji Ling Sheng” increase synchronously with variation.

¹The app’s reviews are still accessible in the iTunes through the link http://ax.phobos.apple.com.edgesuite.net/WebObjects/MZStore.woa/wa/viewContentsUserReviews?id=xxx&pageNumber=0&sortOrdering=1&type=Purple+Software. Make sure to change your location to China and replace xxx with an app id.
of the weekly number of reviewers. However, this phenomenon does not exist in Fig. 3.7(b). This app did not show obvious attacks based on our checking.

![Graphs showing the variation of weekly average ratings and weekly number of reviewers.](image)

(a) App id: 483583569 (Rank #1)

(b) App id: 362949845 (Rank #2)

(c) App id: 435728194 (Rank #9)

**Figure 3.7.** The variation of weekly average ratings and weekly number of reviewers. The x-axis is the index of each week ordered by date. The left y-axis represents the number of reviewers and the right y-axis represents the average rating.

We have examined the top 100 paid apps and 100 free apps (as of May 21, 2012) and filtered out those apps with lifetimes less than nine weeks (so that our result has more statistical significance). This gave us totally 89 apps and about half
for each type of apps. The distributions of these apps’ CC values are illustrated in Fig 3.8. From the shapes we can see that in both free and paid app cases, the distribution approximates to Gaussian Distribution, though skewing slightly to positive of free apps. That is, while majority of the apps look normal, a good portion of them could have been attacked.

We further checked each app looking for signs of promotion or demotion (such signs are discussed in details in Section 3.4.2.1). Among the 22 apps with CC larger than 0.2 and 2 apps with CC value less than -0.2, 18 apps were confirmed to be promoted and 2 apps were demoted. Table 3.4 presents a detailed distribution of CC values for these 89 apps. We can see that large CC values indicate high possibility of attacks. Specifically, a positive CC value indicates a promotion dominating attack whereas a negative CC value indicates a demotion dominating attack.

### 3.3.3.4 Apps’ Quality Estimation

Normally, the square of correlation coefficient (also called coefficient of determination, $R^2$, or $R$ squared) is a statistical measurement of how well a regression line approximates the real data points. In our context, $R^2$ can help us determine the percentage of average rating variations that have linear relationship with rat-
Table 3.4. The relation between CC value intervals and the ratio of abused apps in each interval (demoted apps indicated by *)

<table>
<thead>
<tr>
<th>CC Range</th>
<th>Free</th>
<th>Paid</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.5, 1]</td>
<td>0/1</td>
<td>1/1</td>
<td>1/1</td>
</tr>
<tr>
<td>(0.4, 0.5)</td>
<td>1/2</td>
<td>2/2</td>
<td>3/4</td>
</tr>
<tr>
<td>(0.3, 0.4)</td>
<td>3/5</td>
<td>2/2</td>
<td>5/7</td>
</tr>
<tr>
<td>(0.2, 0.3)</td>
<td>1/3</td>
<td>6/7</td>
<td>7/10</td>
</tr>
<tr>
<td>[-0.2, 0.2)</td>
<td>4/30</td>
<td>7/35</td>
<td>11/65</td>
</tr>
<tr>
<td>[-1, -0.2)</td>
<td>2^*/2</td>
<td>0</td>
<td>2/2</td>
</tr>
</tbody>
</table>

ing population variations in a period. We call the average ratings of such period abnormal ratings because normal (honest) rating variation should have no such linear relationship with rating population variation. We aim to estimate an app’s real quality by removing the impact (not ratings) from the abnormality of ratings instead of identifying which rating is abnormal. Note that in our work the goal of app quality estimation is to use estimated scores for tie strength calculation (and then collusion group discovery). We do not use estimated scores as the real scores for apps, so high estimation accuracy is not the goal (To obtain the real scores, we will need to first remove the ratings by identified collusion groups and then recalculate their scores. This is however out of the scope of our current work.) Next we introduce the detail for app quality estimation.

For an app with version $k$, the abnormal ratings cause its final rating in the app store (i.e., $R_k$) to deviate from its real quality (i.e., $Q_k$). When CC is a positive value, $R_k$ is higher than $Q_k$ and $Q_k$ is bounded between $[1, R_k]$. As shown in Fig. 3.9, $Q_k \in [1, R_k]$ and $R_k - 1$ is the range of $Q_k$. Here the range $R_k - Q_k$ is mainly caused by promotion attacks. The more severe the attack is, both $R^2$ and $R_k - Q_k$ will increase, and vice versa. Let $\frac{R_k - Q_k}{R_k - 1}$ be the normalized value of the range $R_k - Q_k$ indicating the severity of the attack, we can approximate it by a linear function of $R^2$; that is, $\frac{R_k - Q_k}{R_k - 1} = p \ast r^2(V, Z)$, where $p$ is a system parameter reflecting the linear relation between $\frac{R_k - Q_k}{R_k - 1}$ and $R^2 = r^2(V, Z)$. On the other hand, when CC is a negative value, $R_k$ is below $Q_k$. Hence, $Q_k$ is bounded between $[R_k, 5]$ and $5 - R_k$ is the range of $Q_k$. Here the range $Q_k - R_k$ is mainly caused by demotion attacks. The more severe the attack is, both $R^2$ and $Q_k - R_k$ will increase, and vice versa. Similarly, we can normalize $Q_k - R_k$ as $\frac{Q_k - R_k}{5 - R_k}$ and approximate it with $R^2 = p \ast r^2(V, Z)$. Let $E_r$ follow Eq. 3.8, we can get Eq. 3.7.
Figure 3.9. The positions an app’s quality $Q_k$ and its final rating $R_k$ if the correlation coefficient $r(V, Z) > 0$.

\[ p \cdot r^2(V, Z) = \frac{R_k - Q_k}{R_k - E_r} \quad (3.7) \]

\[ E_r = \begin{cases} 1 & \text{if } r(V, Z) \geq 0 \\ 5 & \text{if } r(V, Z) < 0 \end{cases} \quad (3.8) \]

Finally, to derive app quality $Q_k$, we can rewrite Eq. 3.7 as Eq. 3.9.

\[ Q_k = R_k - p \cdot r^2(V, Z)(R_k - E_r) \quad (3.9) \]

Note that in our system, the system parameter $p$ must not be greater than $\frac{1}{r^2(V, Z)}$ to guarantee the right hand of Formula 3.7 will not exceed 1. In practice, once we set a threshold CC value (that is, if an app has CC value above the threshold, it will be considered as suspicious), we can set $p$ accordingly. In the simulation and experiments of this work, we set the threshold CC value as 0.25 and $p = 15$. During calculation, whenever $p \cdot r^2(V, Z) > 1$, we will set it to 1. As a result, $Q_k = E_r$. This means, for a clearly promoted app (i.e., $|CC| > 0.25$), we will estimate its score as 1 (according to Formula 3.9). This may be inaccurate compared to its actual quality (again the purpose of our estimation is not to report actual score for market use). However, it has the advantage of exposing strong ties when two reviewers both gave 5 to this promoted app.

### 3.3.4 Collusion Groups Discovery

After estimating apps’ quality following Eq. 3.9, we are able to calculate the pairwise tie strength following Eq. 3.2 and further build a tie graph to capture all the
Figure 3.10. The tie graph of two collusion groups \( A = \{1, 2, 3, 4\} \) and \( B = \{6, 7, 8, 9, 10\} \). In this tie graph, all the weak ties and negative ties have been removed.

pairwise relations among reviewers. Like the scenario shown in Fig. 3.5, the ties between reviewers are different. Specifically, tie strength between collusion group members is accumulated through each group action and their ties will finally become strong positive ties. For honest reviewers, their rating deviations from apps’ quality are randomly distributed and thus positive ties are harder to form. Therefore, to detect collusion groups, we only need to care about the strong positive ties and neglect the other types of ties.

Discovery of collusion groups finally becomes partitioning the tie graph which only contains strong positive ties. Here, we exploit two characteristics of collusion groups. First, even though collaboration enhances the relation, group members are not always connected to each other in the tie graph because they do not necessarily all participate in the same set of tasks or always behave the same. For example, as shown in Fig. 3.10, reviewer8 has never collaborated with reviewer9 even though they both belong to the same group \( B \). It is possible that reviewer8 only promotes apps and reviewer9 only demotes apps. The other three reviewers in group \( B \) perform both promotion and demotion.

Second, different collusion groups are not necessarily separated from each other. In reality, they could be connected for some reason. In Fig. 3.10, group \( A \) and group \( B \) are connected by reviewer5 because reviewer5 might by chance have rated some common apps with reviewer2 and reviewer6, respectively. As a result, even though there is a strong tie between reviewer5 and reviewer2, they do not belong to the same collusion group. The same reason holds for reviewer5 and reviewer6. Hence, one collusion group does not need to be fully separated from the other groups.

In this section, considering the above two characteristics, we introduce an algorithm to detect collusion groups. Based on the observation that a typical member
Figure 3.11. An example of 4-clique-community. There are four 4-cliques which are 
\{1, 2, 3, 4\}(c1), \{1, 2, 4, 5\}(c2), \{2, 4, 5, 6\}(c3) and \{2, 5, 6, 7\}(c4). c1 is adjacent to 
c2 and they share three nodes. Similarly, c2 is adjacent to c3 and c3 is adjacent to c4. 
Any two 4-cliques can reach each other through such adjacent neighbors.

in a community is linked to many other members but not all of them, a community 
can be interpreted as a union of smaller complete (fully connected) subgraphs. Let 
k refer to the number of nodes in a subgraph; then a subgraph is also called a 
k-clique. Palla et al. [51] defined a k-clique-community as a union of all k-cliques. 
All the k-cliques can be reached from each other by way of a series of adjacent k-
cliques, where “adjacent” means sharing \(k-1\) nodes. For example, Fig. 3.11 shows 
a 4-clique-community where four 4-cliques comprise a community. This definition 
expresses the essential feature of community: members can be reached through 
well-connected subsets of nodes. The other parts cannot be reached through k-
cliques are probably another k-clique community. Meanwhile, a single node can 
be in several communities. Thus, the whole graph consists of overlapping commu-
nities.

Since these features of communities also exist in collusion groups, discovering 
collusion groups is equivalent to find k-clique-communities on the tie graph with 
only strong positive ties. It is called k-clique-groups in this work. We can adapt 
the algorithm introduced in [51] to discover k-clique-groups. The variance of k 
influences the structure of collusion groups. If \(k = 2\), a tie graph is considered to 
be a collusion group because all nodes are connected. For example, there is only 
one single collusion group in Fig. 3.10, which contains all the reviewers. Similarly, 
a 3-clique-group is given by the union of triangles that can be reached from one to 
another through a series of shared edges. As \(k\) increases, the size of collusion groups 
shrinks, but on the other hand, it becomes more cohesive since their members have 
to be part of at least one k-clique. For instance, Fig. 3.10 is actually a 4-clique-
groups which contains group A and group B. Group B contains two 4-cliques and they share a 3-clique.

Since we are looking for k-clique-groups, those groups smaller than k will be discarded because they are unable to complete a successful attack with few group members. These small groups may be formed by biased reviewers who happen to rate the same apps. They can also be some random reviewers even though they rate honestly. Even though we misidentify some honest reviewers as attackers, it is easy to further filter them out with other features discussed in Section 3.4.2.1. In practice, we may use GroupTie as the first round of discovery and set a small k to catch as many attackers as possible during the next round of detection.

3.4 Experiment and Result Analysis

We have implemented our collusion groups discovery model in JAVA based on Fedora 13 and stored all the data in MYSQL 5.1.56. Two types of experiments are conducted; one is the simulation to evaluate the algorithm and the other one is to detect collusion groups in an actual app store.

3.4.1 Simulation Study

3.4.1.1 Experiment Settings

We first randomly choose 29 apps from Apple App China Store and these apps look free of attacks. We set each period to be one week because it is a general period of our daily life; accordingly, the lifetime of an app is described in terms of number of weeks. For our simulation purpose, we consider all of their original ratings honest.

We then introduce one collusion group to manipulate $N_{apps}$ apps ($N_{apps} = 4$ by default\(^4\)) by adding positive, negative, or honest ratings. In the simulation, the collusion group controls two parameters: $P_{period}$, the percentage of weeks in an app’s lifetime to introduce attacks; and $P_{raters}$, the ratio of the number of malicious raters to the population of raters in each week for an app. Once the collusion group chooses an app to attack (i.e., to promote or demote), its members may take one or more attack strategies, as described in Section 3.2.2: PoA, DoA and OA. In PoA,

\(^4\)We also tried 8 target apps, which showed similar results
the attackers try to rate high to apps with low quality. They have roles \((0, 0, P)\) and \((H, 0, P)\), and 50\% of them follow \((0, 0, P)\) and the other 50\% follow \((H, 0, P)\). The attackers with role \((H, 0, P)\) provide honest ratings and high ratings randomly. Similarly, in DoA, half of the attackers follow role \((0, D, 0)\), and the other half with role \((H, D, 0)\) rate honestly or negatively with 50\% probability. In the case of OA, half of the abused apps will experience PoA and the other half experience DoA.

Two metrics are used in our evaluation: false positive rate (FPR) and false negative rate (FNR). Here FPR measures the percentage of identified attackers which actually do not belong to the collusion group, and FNR represents the proportion of attackers which have evaded detection. When measuring strong or weak ties, we set the tie threshold as 8.

### 3.4.1.2 Results Analysis

Under the above attacks, we calculate the false positive and false negative rates of GroupTie, as shown in Table 3.5. From the table we can see that the overall FPR and FNR of GroupTie is about 0.31\% and 9.29\%, respectively. GroupTie has lower FPR but higher FNR in the PoA case than in the DoA case. Its performance with OA is just between that with PoA and with DoA.

The differences of FPR and FNR between the PoA case and the DoA case are mainly caused by the initial skewness of correlation coefficient (CC). In our simulation study, we assume these 29 apps are free of attacks; that is, theoretically their CCs should be about 0. However, their actual CCs mostly skew to positive. We observe that initial positive skewness of CC leads to the result that the quality of an app (i.e., \(Q_k\) in Eq. 3.9) becomes overestimated after the app quality calibration process, compared to the case when the original app is truly attack-free (i.e., when its CC is close to 0), even though the attacks work the same way in both cases.

As a result, according to Eq. 3.1, the deviations of ratings from app’s quality are decreased, so are tie strengths according to Eq. 3.2. That is, strong ties will be fewer. In the end, the constructed tie graph is smaller and hence FPR decreases and FNR increases.

Since the FPR of GroupTie is very low in our simulation study, our discussion will be focused on its FNR. The results are shown in Fig. 3.12 which corresponds to the PoA case (we do not show the results for the DoA and OA cases because the results are similar) In Fig. 3.12, its FNR has no obvious change when \(P_{raters}\)
Table 3.5. The average value of FPR and FNR under attacks like PoA, DoA and OA.

<table>
<thead>
<tr>
<th>Attacks</th>
<th>FPR</th>
<th>FNR</th>
</tr>
</thead>
<tbody>
<tr>
<td>PoA</td>
<td>0.04%</td>
<td>11.59%</td>
</tr>
<tr>
<td>DoA</td>
<td>0.62%</td>
<td>7.99%</td>
</tr>
<tr>
<td>OA</td>
<td>0.25%</td>
<td>8.27%</td>
</tr>
<tr>
<td>Overall</td>
<td>0.31%</td>
<td>9.29%</td>
</tr>
</tbody>
</table>

Figure 3.12. The FNR of GroupTie with PoA.

increases. The reason is as follows. As $P_{raters}$ determines the number of attackers in each week selected for attacks, at first, its increase will cause its weekly average rating to depend on the number of reviewers for that week. However, its further increase in that week (i.e., having more attackers in one week) will not strengthen the overall linear relationship that is calculated based on many weeks. We notice that when $P_{raters}$ is larger than a certain threshold, it will have little influence on the values of CC and $R^2$ for an app. As a result, because $R^2$ is little affected, the estimated quality of an app (i.e., $Q_k$ in Eq. 3.9) will be about the same. Based on the similar reasoning as above, the resulted tie graph will be about the same, so FPR and FNR will not change much with $P_{raters}$.

On the other hand, we observe that the FNR of GroupTie decreases as $P_{period}$ increases in the PoA case. This is because a larger value of $P_{period}$ results in more weeks to be attacked, and hence more data points (i.e., weekly average ratings) to become linearly dependent on the number of rating users in each week. Consequently, CC and $R^2$ of the app increase. Based on Eq. 3.9, Eq. 3.1 and Eq. 3.2, we
can know that more strong ties will appear and the final tie graph will cover more users. Hence, the FNR will decrease.

### 3.4.1.3 Influence of Tie Threshold

To study the influence of tie threshold, we set up another group of experiments. A collusion group with 100 members is set to attack four of the apps. Under different attacks like PoA, DoA and OA, we change the threshold of tie strength ($T_{hp}$) to observe the changing trend of FPR and FNR.

From Fig. 3.13 we can see that the trend of FPR and FNR under PoA. FPR decreases to 0 with the increase of tie threshold while FNR increases from 0 to 1 when the tie threshold is larger than 62. The reason is that the increase of tie threshold will filter out ties between honest reviewers at first, which leads to the decrease of FPR. However, its further increases will discard some ties between attackers, and it is why the FNR increases from 0 to 1 suddenly.

Under the attack of DoA, shown in Fig. 3.14, the trend of FPR and FNR are almost the same and the difference is that FPR decreases more rapidly than PoA. Since CC skews to positive of honest ratings, less honest reviewers will be misidentified in DoA than PoA when tie threshold is small. Since OA is a mixture of PoA and DoA, we did simulations for three different settings of OA: one is 67% PoA and 33% DoA; the second is 50% PoA and 50% DoA and the last is 33% PoA and 67% DoA. The trend of FPR and FNR is similar to PoA and DoA.

The limitation of this simulation is that there is only one collusion group and that is why FNR increases to 1 suddenly when tie threshold exceeds 62. In reality, FNR grows more gently while tie threshold increases. In a word, the increase of tie threshold will cause FPR to descend (until 0) and meanwhile FNR to grow (until 1).

According to the performance of FPR and FNR under PoA, DoA and OA, tie threshold is a important criterion to differentiate ties between honest reviewers from ties between attackers. Even though some of the ties between honest reviewers are still larger than the tie threshold, these honest reviewers are hard to form a $k$-clique collusion group with others which requires fully connected. Therefore, tie threshold is effective to filter out most ties among reviewers.
3.4.2 Discovering Collusion Groups in App Store

3.4.2.1 Data Description and Detector Settings

We collected and analyzed the data of 200 apps from Apple’s China App Store, which consist of 100 top paid apps and 100 top free apps on May 21, 2012. Among them, 111 apps have lifetime shorter than 9 weeks, so they were not included for further study. We also removed 37,863 anonymous reviewers and their reviews. In the end, we were left with the test set of 89 apps, 818,545 reviewers and 1,042,832
Establishing the ground truth on the suspiciousness of these apps and their reviewers is certainly a challenge here. Manually checking all the reviews to identify possible attacks is not feasible in two aspects: scalability (the number of reviews is too huge) and accuracy (many evidences are hidden). As such, we turn to build an automatic tool that leverages the following heuristics to expose suspicious apps and the collusion reviewer groups. Note that when introducing these heuristics, the examples of apps and reviewer ids were not discovered from our test set consisting of these 89 apps and 818,545 reviewers.

**Review Intensity** represents the distribution on the number of ratings over each unit time. Basically, if one checks all the reviews of an app and orders them by “helpfulness”, one may notice that many same type of reviews are posted in a very short period. For example, among the reviews of the app with id 499814295, most five star ratings were posted on April 28, 2012 and May 22, 2012. Among all the reviews of app 499805269, there were 188 most helpful reviews and 183 of them were five star posted on June 1, 2012. This phenomenon reflects the collaborative rating behavior of collusion groups to make the attacks more effective.

**Skewed Ratings** means that the distribution of rating scores of an app in a short time period is very skewed compared to its overall score distribution. For example, for the app with id 474429394, its overall rating summary for version 1.5.2 is (28, 9, 25, 37, 241) (rating scores ordered from one star to five star). We also collected its ratings from July 1, 2012 to Aug 3, 2012 and got its rating summary (1, 0, 0, 3, 136) for this time period. It is easy to observe that the ratio of five star to one star ratings increases from 241/28 = 8.61 to 136/1 = 136.00. Therefore, the developer might have hired a collusion group to promote its app version 1.5.0. Another example is app 499805269. Its rating summary of version 2.5.6 was (2, 1, 0, 0, 183). If checking all its ratings in Jun 1, 2012, the rating summary was (0, 0, 0, 183). This indicates all its five ratings appeared in one day. However, for normal apps, the ratio does not change a lot in different periods. This phenomenon is due to collaborative promotion which generates massive five star ratings in a short time window to be effective.

**Highly Similar Review History** means group members have co-rated many common apps in the past. For example, we find a group of 171 reviewers all have rated and nearly only rated apps with ids (499814295, 525948761, 485252012,
Table 3.6. Abuse signs of apps

<table>
<thead>
<tr>
<th>Sign</th>
<th># of apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reviews Intensity or Skewed Ratings</td>
<td>9</td>
</tr>
<tr>
<td>Highly Similar Review History</td>
<td>7</td>
</tr>
<tr>
<td>Consecutive Ids</td>
<td>2</td>
</tr>
<tr>
<td>Removed Apps</td>
<td>6</td>
</tr>
</tbody>
</table>

496474967, 499805269) in the past. Only a few of them also rated 460351323.

**Consecutive Review Id** means reviewer IDs of some groups are very close. Even though we do not know the mechanism for id (i.e., Apple ID) generation in iTunes, it is still weird that many reviewers with very close ids reviewed the same set of apps. For example, the following reviewers have commonly reviewed two apps with ids 474429394 and 525378313, respectively.

“212525736, 212525739, 212525752, 212525762, 212525765, 212525781, 212525784, 212525788, 212525795, 212525800, 212525808, 212525811, 212525825, 212525827, 212525834, 212525847, 212525851, 212525857, 212525860, 212525864, 212525903, 212525990, 212525996”

If iTunes generates user ids in an incremental way, this group probably applied these IDs in a very fast way (e.g., through bots). This sign would indicate not only the existence of a collusion group, but also the co-rated apps by these reviewers are probably abused apps.

**Removed Apps** means some apps have been removed from app stores, due to reasons like rating manipulation as reported by the news.

Note that we cannot claim the results based on these heuristics are 100% accurate, but our algorithm output can provide a much-narrowed-down list for further investigation by app stores. App stores may use additional evidences which we do not have (e.g., information about reviewer accounts) to further pinpoint the abused apps and the collusion groups.

3.4.2.2 Findings

Based on the above signs, we implemented a tool to check the apps. For the 24 apps (out of the total 89 apps) whose CC values were above 0.2 or below −0.2, we

---

5The 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.
identified 18 of them as suspicious (labeled as abused apps), because they bore one or more signs mentioned above, as shown in Table 3.6. The relation between CC value intervals and the ratio of suspicious apps in each interval has been shown in Table 3.4. Furthermore, among 818,545 reviewers, we found that 8,853 reviewers (i.e., 1.08%) had strong ties, and they formed 734,848 strong positive ties. We applied the $k$-clique-communities algorithm described in Section 3.3.4. As illustrated in Fig. 3.15, when $k$ increases from 10 to 100, the number of communities decreases sharply from 136 to 9. When $k$ further increases, the number of communities continues to decrease until to the largest $k$ value (i.e., 1,588). The reason is that when $k$ increases, more and more groups cannot satisfy the $k$-clique requirement any more and they are not considered as a group.

When $k$ equals to 100, we found 3,677 reviewers forming 9 groups, which contained one large group with 2,652 members and 8 smaller groups. We also checked their target apps by counting the commonly rated apps of the group members. As shown in Table 3.7, the 8 small groups targeted at some apps to do promotion or demotion and they have some specific characteristics. For example, the target apps of group 1 actually belonged to one developer. Group 4 seemed to have two types of tasks: one was to promote 324101974 and the other was to demote 444934666. Group 9 was the largest group and it was almost 13 times larger than the other groups. Similar to group 4, its targets included both demoted (e.g., demoting the products of Tencent Inc. [52]) and promoted apps. Indeed, among all the target apps...

![Figure 3.15. Number of collusion groups with $k$.](image)
Table 3.7. Collusion groups discovered by GroupTie.

<table>
<thead>
<tr>
<th>#</th>
<th>Group size</th>
<th>Target App ID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>104</td>
<td>363966906, 431194169</td>
</tr>
<tr>
<td>2</td>
<td>107</td>
<td>444934666</td>
</tr>
<tr>
<td>3</td>
<td>145</td>
<td>441216572, 512500671</td>
</tr>
<tr>
<td>4</td>
<td>146</td>
<td>324101974, 444934666</td>
</tr>
<tr>
<td>5</td>
<td>151</td>
<td>324101974, 483583569</td>
</tr>
<tr>
<td>6</td>
<td>180</td>
<td>414478124</td>
</tr>
<tr>
<td>7</td>
<td>198</td>
<td>324101974, 363966906</td>
</tr>
<tr>
<td>8</td>
<td>195</td>
<td>512500671</td>
</tr>
<tr>
<td>9</td>
<td>2652</td>
<td>324101974, 414478124, 441216572, 483583569, 512500671, 363966906</td>
</tr>
</tbody>
</table>

apps listed in Table 3.7, all were labeled as abused apps previously. This indicates these 9 groups are highly likely collusion groups.

Besides co-rating abused apps, these groups were further found some suspicious signs (including consecutive ids and high review intensity) for most of these identified groups. For example, from the members of group #1, we found some consecutive reviewer ids like “192543956, 192544000, 192544012; 19288476, 19288856, 192888638, 192888756, 192888780, 192888811, 192888859, 192888891, 192889021, 192889058, 192889084, 192889232”. Also, they rated both apps 363966906 and 431194169 within several days. Indeed, high review intensity phenomenon appears in all these groups but group #2 and group #6 (these two groups seemed to demote their targets belong to Tencent Inc.).

For other groups, we first identified the abnormal period of the apps and then checked how many reviewers had reviewed those apps within abnormal periods. As for group #4, we found the developers had promoted their apps in certain time periods. For app 324101974, the time periods were Dec 1, 2011, Dec 02, 2011, Dec 03, 2011, Jan 19, 2012, Jan 20, 2012, May 14, 2012 and May 15, 2012. We finally found 94 members had left five star to this app during these days. For group #5, we found 127 members (accounting for 84.11%) reviewed app 483583569 during the abnormal period from April 29, 2012 to May 24, 2012. And there were 98 (accounting for 64.47%) members also reviewed app 324101974 during its abnormal period. For group #7, we found 113 members (accounting for 55.39%)
reviewed app 324101974 during its abnormal period. However, only 29 members found having reviewed app 363966906 within abnormal period of June 27, 2012. As for group #8, we found 144 members (accounting for 74.23%) reviewed app 512500671 within its abnormal period between April 19, 2012 and May 23, 2012.

Similar sign was also found in group #9. However, such sign was not found among group #2 and group #6. These two groups were actually intend to demote their targets which belong to Tencent Inc. Considering these products are popular in China store, these two groups are probably normal group who hate Tencent’s products. As for the largest group, nearly 86.38% of the reviewers worked together to attack at least one target app. That is because not all of the reviewers have to work together.

3.5 Discussion

Next we discuss several issues related to the limitations and future improvement of GroupTie.

**Attacks To GroupTie**  GroupTie is for detecting the close relation between collusion group members. Clearly, a collusion group, once knowing that GroupTie is in place, may try its best to evade the detection, with different levels of difficulty, complexity, efficiency and cost. For example, they can try strategies like randomizing their ratings or collaborators for different apps to make their attacks more stealthy at the cost of lower attack effectiveness. Clearly, there will be an arms race which could attract more research work in the future. We believe our current work, though as a first step, has greatly raised the bar for attackers.

**Impact of App Popularity**  For popular apps, our conjecture is that collusion attacks are less likely as the developers do not have the motivation to perform further promotion at the risk of their reputation. For their competitors, little impact could be generated by demoting popular apps because popular apps often have a large number of ratings, e.g., thousands to even millions. To demote popular apps, a very huge collusion group is needed, but this puts itself into a high risk once being exposed. In our future work, we will investigate the app popularity issue further. One idea is to assign more popular apps with lower weights in calculating reviewer tie strengths. In this way, even the probability of two honest reviewers co-rating popular apps is higher, their tie strength will grow much slower,
ultimately reducing false positives.

**Computational Complexity and Storage Overhead** We note that storage overhead is not an issue here if an app store is going to run our algorithm, because it has the rating/review information already. The computational complexity is a concern if we want to run our algorithm over the entire store, that is, for all raters and all apps. This is because we need to compute tie strength for each pair of users if they have ever co-rated an app and the number of such pairs is huge for an entire store. It will not be possible to compute tie strength for all pairs in physical memory. In practice, one optimization could be to run the algorithm only for suspicious apps and their associated reviewers. Further, we may distribute tie strength computation by leveraging the MapReduce framework to build tie graphs.

**Application to Other App Stores** GroupTie is designed based on general principles and assumptions, so it should also be applicable to Google Play or BlackBerry App World. However, as a third party, we have a constraint to apply GroupTie over Google Play because Google Play only displays up to 480 reviews for each app. Nevertheless, vendors like Google can apply our algorithms by themselves.
Chapter 4 —
Unveiling the Underground Market of Trading Mobile App Reviews

4.1 Introduction

The great popularity of smart devices (e.g., tablets and smartphones) has attracted more and more software developers to engage in mobile application (app) development. As of September 2014, both Apple App Store [2] and Google Play [53] offered 1.3 million apps. The entire app market has reached up to 35 billion dollars in 2013 and it could hit 70 billion dollars by 2017 [54]. Meanwhile, the competition to attract customers has become more furious than ever. To promote their apps, some developers choose to buy reviews from companies like bestreviewapp [55], dailyappshow [56]. This review trading market will cause great damage to app market players. On one hand, these paid reviews may mislead honest customers and cause both time and money losses to them. On the other hand, it will demoralize those developers who try their best effort to improve app qualities in order to receive more and better reviews. Therefore, paid reviews are forbidden by app store vendors.

By the place where a paid review is posted, the app review market can be grouped into two types. The first type is to post reviews directly in app stores (referred to as app store promotion). Unlike traditional product stores, app stores are highly centralized; for example, iTunes is the only store that distributes apps for non-jailbroken devices of iTouch, iPhone, and iPad. Therefore, app reviews posted in these stores could be read by all the app users. However, as one user
is only allowed to write one review for an app, non-conforming developers have
to find many reviewers to promote their apps. This is often difficult. Thus, some
service providers have established a business to charge developers who seek reviews
and pay app users who like to make money for writing app reviews. The second
promotion type is to post reviews on popular websites (referred to as web promo-
tion), which have a large population of readers and followers in social networks like
Facebook and Twitter. They may quickly broadcast app reviews to their readers
and followers who are potential app users, and meanwhile, these websites would
get paid by app developers. Until now, this app review trading market is still
under the ground and unknown to the community.

In this chapter, we aim to unveil the underground market of trading mobile
app reviews and collect a set of promoted apps as the ground truth to further
study their statistical characteristics. We joined their promotion communities as
a passive reviewer or follower and collected apps assigned to us. We have built
an automatic data collection system, called AppWatcher, to monitor 52 service
providers every day. In four months, AppWatcher has gathered 645 apps reviewed
by app store promotion and 29,680 apps reviewed by web promotion. It has
also collected app meta data like rating, text comment, and reviewer information,
and randomly chosen 186,263 apps from Apple App Store for comparison study.
We have statistically studied service providers, promotion incentives, promoted
apps, and suspicious reviewers. Moreover, we have compared their features with
randomly chosen apps and tested the significance of their difference.

4.2 Data Collection Methodology

4.2.1 Service Provider Collection

Our basic way to find app promotion service providers is searching for their websites
with the help of search engines.

To look for app store review providers, we search with various combinations
of keywords such as “paid”, “buy”, “app”, “review”, “rating”, “app stores”. For
website review providers which charge for app reviews, we use the combinations
of “paid”, “buy”, “app”, “review”, “evaluation”, “rating”. In addition, we have
found one blog [57] that keeps updating the list of such websites.
4.2.2 Service Provider’s Meta Data Collection

To collect a service provider’s meta data information, we read its pages like FAQ, About, Pricing. Specifically, we collected such data as service price, reward to reviewers, service launching time. Service launching time is the time when a service provider started to review apps. Normally, a website shows its creation time (i.e., year) at the bottom of the front page. If there is no such information, we would refer to the whois information of the domain and use the domain creation time instead.

4.2.3 Promoted App Collection

We have built an automated data collection system, named AppWatcher, to gather promoted apps from the service providers. AppWatcher is implemented in JAVA and Python and it has 12,908 lines of source code. As illustrated in Fig. 4.1, the system includes four major modules: website monitor, assignment collector, app meta data crawler, and database.

**Figure 4.1.** The interactions between AppWatcher and review trading market. The detailed actions between them are as follows. A1: App developers upload their apps to app stores; A2: Some developers buy reviews from service providers; A3: The “website monitor” module monitors target service providers to collect paid reviews. If the service provider is recruiting reviewers online, “assignment collector” module will join as a passive reviewer to collect assigned app information; A4: the module of “app meta data crawler” crawlers paid reviews along with their dates; A5: “app meta data crawler” retrieves these promoted apps’ meta data from their corresponding app stores.

**Website monitor** is used to collect promoted apps from websites (i.e., service
providers) where the reviews are published. It retrieves information like app id, review publishing date, author, review content, and editor’s rating. Then, it saves such structural data to the database.

**Assignment collector** is used to gather the information of promoted apps from service providers which recruit reviewers from the wild. We registered an account in each service provider, and waited for assignments (i.e., apps to be reviewed) from service providers. We then parsed the assignment page and retrieved app identities. This module was run twice a day to monitor the change of assignments and record the time when a new app is added and the time when it is gone.

**App meta data crawler** is for retrieving an app’s detailed information from its app store. For each app collected by Website Monitor and Assignment Collector, this module searches the app meta data (e.g., app description and reviewers) from its app store. The structural data is stored in the database for further analysis. For iTunes, the collector is able to crawl all the reviews of an app. However, Google Play only allows to view up to 5,000 latest reviews for any app, so our collector only collects the latest 5,000 reviews or all reviews (when total number is below 5,000).

### 4.2.4 Ethical Consideration

Our study on the underground market for review/ranking fraudulence is ultimately related to the real app market. One important rule in our experiments is not to influence the real market. Therefore, we have chosen to be an passive observer. Specifically, when crawling apps from a website, we slowed down the speed of crawling (i.e., twice a day) to avoid traffic jamming. In addition, when joining as a reviewer in a reviewer recruiting website, we only passively received the assignments without posting any review for any app. When crawling app meta data from app stores, we also limited the downloading speed.
4.3 App Store Promotion

4.3.1 Dataset

We have found 26 websites that have been involved in app store promotion. Among them, five websites were closed after being exposed very recently. These websites are used to advertise their services. Ten of them are even used to recruit reviewers online, and two of these ten websites require to review an app or pay for registration fees before joining. To not violate the ethical regulation, we deployed AppWatcher to only monitor the remaining eight websites. We registered an account as a reviewer in each website and recorded apps assigned to us. We logged into these websites twice a day and observed the change of these assignments. In total, we have collected 645 apps, which were mostly promoted from July 1st to October 7th, 2014\(^1\).

Among these promoted apps, 618 are from the iOS platform while only 27 apps are from the Android platform. Since the population of promoted Android apps is not sufficient for statistical study, we chose to focus on iOS apps. 54 iOS apps, which were promoted by “reviewfordev”, have been removed from Apple App Store, so we collected the meta data of 564 iOS apps with 497, 259 text comments and 462, 896 reviewers.

For comparative study, we also randomly selected 179,353 apps from Apple App Store, which include 10% apps from each subcategory that has over 100 apps and all apps in each subcategory with less than 100 apps\(^2\). We collected their meta data, including 9,399,014 text comments, and 6,722,558 reviewers.

4.3.2 Analysis of Service Providers

For all the service providers engaged in app store based promotion, we mainly study when their services started, how much they charge customers, how much they reward reviewers, and how many apps they have reviewed.

\(^1\)“reviewfordev” displays all the apps it has promoted, and some of these apps were promoted before our monitoring.

4.3.2.1 Service Launch Time

Service launch time is the time when service providers start to review apps. The first service was started in 2010 and then more and more service providers emerged in 2012 and 2013. On their websites, many of them mention “1 million apps in iTunes” as the incentive for using their services. With over a million apps in an app store, it is extremely difficult for a new app to acquire initial reviews, while with no or few reviews an app would be ranked far behind and look much less attractive to app users. Therefore, their review promotion services could help app developers booster up their apps out of this difficult time.

4.3.2.2 Service Price and Reward

Service price is the price of writing one text review along with a rating score. In Fig. 4.2(a), we depict the lowest price for one review from each service provider. The highest price is $12.5 from “buyklout” [58], and the lowest one is around $0.5 from “androidappreviewsources” [59]. Basically, buying more reviews will be given more discounts. For example, “appiness” [60] charges $9 (US dollars) for each review if buying 5 reviews, $8 for 10, $7 for 50, and $6 for 100 reviews.

The reward for writing a review varies from $0.5 to $5. For each qualified review, “appiness” and “reviewfordev” pay $5 and $4, respectively, while most others pay $0.5. Some service providers use other type of rewards. For example, “appredeem” pays 1000 points, which can be cashed out; “appwin” sets up a lottery for app reviewers.
4.3.2.3 Promoted App Volume

Except “reviewfordev” [61], none of the service providers shows the total number of promoted apps on their websites. However, some of them recruit online reviewers, so we can join as a passive reviewer and monitor app review assignments for more than four months. As shown in Fig. 4.2(b), the most popular service provider is “bestreviewapp” [55], which has accepted 251 apps. Some other providers like “apprebates” [62](84), “getappreviews” [63] (38), “appredeem” [64] (34), “promodispenser” [65] (32), and “giftmeapps” [66](30), have less apps. For the rest three, only 3 or less apps were found. In total, we have found 473 apps promoted within the monitoring period.

Except service providers, probably no one knows the accurate count of paid reviewers (or accounts). However, some of them have offered review packages, which state the number of reviews they are able to deliver for one app. For app stores like iTunes, each user is only allowed to write one review for each app. Therefore, the maximum number of reviews they are advertising to deliver for one app could reflect the number of reviewers they can hire or the number of accounts under their control. From Fig. 4.3, we can see that “4xn” [67] has the largest number of reviewers (i.e.,1,000). The second largest one is “getappreviews” [63], which can deliver 500 reviews at a time. The others can deliver less than 200 reviews.

![Figure 4.3. The maximum number of reviews that can be delivered in a single package.](image)

\[
\text{Maximum # of reviews}
\]

\[\begin{align*}
4xn & \quad \text{getappreviews} & \quad \text{androidappreviewsourcing} & \quad \text{appness} & \quad \text{buyreviews} & \quad \text{appreviews4sale} & \quad \text{reviewfordev} \\
1000 & \quad 800 & \quad 600 & \quad 400 & \quad 200 & \quad 0
\end{align*}\]
4.3.3 Analysis of Promotion Incentive

4.3.3.1 Promotion Incentives

To find out when and why a developer made his decision to start app promotions, we analyzed app events like version updates and rating changes that happened before the promotion.

First, we retrieved all the version update history from iTunes\(^3\) and identified the most recent version update before the promotion. Further, we calculated the intervals (in terms of days) between the version update and the promotion. The average interval is 39 days. Fig. 4.4(a) shows the distribution of all the intervals. Specifically, 47% promotions were launched within one week after a new version update, 62% within two weeks, 80% within 60 days, and 92% within 120 days.

Second, we studied apps’ weekly average ratings for four weeks before the promotion. As shown in Fig. 4.4(b), nearly half of average ratings are one star, 80% average ratings are below three stars, and 90% below four stars. That is, most apps had received low average ratings before promotion.

Third, we looked into the distribution of weekly reviewer quantity. As illustrated in Fig. 4.4(c), more than 80% apps have less than 10 reviewers in a week and more than 90% apps have less than 50 reviewers. That is, most promoted apps have few reviewers in weeks before promotion.

\(^3\)Google Play does not provide version update events on its website, so we only focus on iTunes.
Statistically, low average ratings and few reviewers are the most likely motivation for promotion.

4.3.3.2 Promotion Completion Time

To avoid the notice of app store vendors, most service providers try to post reviews gradually rather than posting bulk reviews in a very short time. Another reason could be that promotion reviews are completed by human, which limits the speed. Nevertheless, most of the reviews are promised to be completed within one week (e.g., appiness, bestreviewapp, appreviews4sale), or two weeks (e.g., appreviews-source). For example, as shown below, appiness mentions its strategy in its FAQ page.

Q: How long does it take to complete the reviews? A: It’s not a precise science because people are involved. Reviewers agree to review an app within 72 hours of receiving the request. Some may do it immediately, some may do it after a day and some may push it to the limit. However, we like this 72 hour window because this ensures that the downloads and reviews looks natural.

Q: I’d like to order 100 or more reviews but I want it to be spread out over time? A: No problem. First, reviewers can take anywhere from 0 to 72 hours to review the app, so you will have a natural trickling in of reviews.

To confirm this, we studied the number of days needed to complete a review task. The quickest task was completed within one day, while the slowest one took up to 115 days. The average completion time was 21 days. As illustrated in Fig. 4.5, nearly 77% promotions were completed within 30 days and nearly 90% promotions were completed within 60 days.

A few service providers state that they can deliver the reviews within a very short time like one day (e.g., 4xn, madapp [68]). madapp [68] states that it can deliver 1,000 organic reviews (i.e., written by real users) within one day. Given such a big number, our conjecture is that it may leverage bots to automatically generate and post some reviews.

\[^{4}\text{This website has been closed very recently.}\]
4.3.4 Analysis of Promoted Apps

For all the promoted apps, we collected their meta data and then studied their platforms, average ratings, total number of reviewers, category distributions, and developers. For the purpose of comparison, we also randomly chose the same number of iOS apps that have at least one text comment.

All the promoted apps are either on iOS (96%) or on Android (4%). Even though both app stores have more than 1.3 million apps, iOS app developers were reported to have 85% more revenue than Android apps [69]. This could motivate more developers to promote their iOS apps than Android apps. As promoted Android apps are too few, next we only focus on iOS apps.

The average ratings from reviewers who posted text comments are shown in Fig. 4.6. Over 90% promoted apps have positive (i.e., ≥3 stars) average ratings and 80% average ratings are above 4 stars. Compared to randomly chosen apps, promoted apps have much higher average ratings. To test whether their difference is statistically significant, we conducted a Welch’s t-test with the null hypothesis that the mean of promoted apps’ average ratings are identical to that of random apps’ average ratings. The test returned a small p-value (p-value<0.05), which denied the null hypothesis and supported the hypothesis that promoted apps have significantly larger average ratings than random apps.

We also depict the total number of reviewers. From Fig. 4.7 we can see that promoted apps have more reviewers than random apps. Their difference was also tested to be significant by Welch’s t-test.

Figure 4.5. The number of days needed to complete reviews for an app.
The sequence of apps

Average Rating (iOS)

Promoted Apps
Random Apps

Figure 4.6. Average ratings from reviewers who have provided both ratings and text comments to iOS apps.

Promoted Apps
Random Apps

Figure 4.7. Total number of reviewers who have provided both ratings and text comments to iOS apps.

To study the category distribution, we count the number of apps in each category. As an iOS app could be assigned multiple categories, the app would be counted in more than one category. As shown in Fig. 4.8(a), the top first category is “Games” following by “Newsstand”, “Photo & Video”, “Lifestyle”, and “Productivity”. Compared to random apps in Fig. 4.8(b), the category distribution of promoted apps is nearly the same. However, promoted apps in each category (except “Social Networking”, “Medical”, and “Navigation”) have more paid apps than random apps have.

As for app developers, all the promoted apps belong to 373 developers. Among them, 79 developers (21%) have at least two apps while the remaining 294 developers (79%)
only own one app. The top three developers have 15, 13, and 13 promoted apps, respectively.

4.3.5 Analysis of Recruited Reviewers

For promoted iOS apps, we collected all the reviewers who posted text comments. In this section, we study their behaviors in terms of co-rating and average rating. For comparative study, we randomly chose the same number of reviewers from Apple App Store.

Even though we cannot tell which reviewers were recruited, those who have reviewed multiple promoted apps after promotion started are suspicious. Given the set of 564 promoted apps, we first collect the number of reviewers who reviewed them, and then calculate the number of reviewers (y-axis) who have reviewed a certain number of promoted apps (x-axis), shown in Fig. 4.9. We can observe that while the most number of reviewers reviewed only one promoted app, many reviewers have rated over tens of promoted apps. There are even two reviewers who reviewed 246 promoted apps. In contrast, for random (i.e., randomly chosen) apps, none of their reviewers have reviewed more than three apps in the chosen set (we do not show a figure here). Therefore, active reviewers of promoted apps are likely recruited reviewers.

Further, we collect those reviewers (also called suspicious reviewers) who reviewed at least two promoted apps during monitoring time. In total, we find 1,363 such reviewers. We then count the total numbers of apps in the app store they have reviewed, including apps that are not in our known promotion app set. In
Fig. 4.9. Number of reviewers (y-axis) who have reviewed certain number of promoted apps (x-axis)

Fig. 4.10, we list all such reviewers in x-axis based on the number of reviewed apps (in the decreasing order). On average, each suspicious reviewer reviewed 9 apps, while the highest number is close to 800. We can also observe that the first 200 suspicious reviewers all have reviewed at least 100 apps in the app store. In contrast, less than 30 random reviewers have reviewed at least 100 apps in the app store. Therefore, overall suspicious reviewers have reviewed significantly more apps than random reviewers.

Fig. 4.10. Number of apps reviewed by one reviewer.

Furthermore, we study the average ratings provided by reviewers. For suspicious reviewers, we chose those who had reviewed at least two and at least six promoted apps. While for random reviewers, we randomly chose those who had
reviewed at least two and at least six apps in the app store. As illustrated in Fig. 4.11, for those who reviewed at least two apps, nearly 1,200 (88%) suspicious reviewers, but only 800 (59%) random reviewers, have average ratings higher than 4 stars. In the case of six apps, nearly 400 (67%) suspicious reviewers, but only 200 (33%) random reviewers, have average ratings higher than 4.5. Therefore, average ratings from suspicious reviewers are significantly higher than that from random reviewers in both cases. This observation is also supported by a Welch’s t-test.

![Figure 4.11](image.png)

**Figure 4.11.** The average ratings to all apps from reviewers rated different number of apps.

### 4.3.6 Promoted App Tracer

Joining as a recruited reviewer to monitor review assignments, we have collected many promoted apps from several service providers. However, these apps are probably only a small part of all promoted apps. This is because this kind of promotion services has been launched for years, while AppWatcher only monitored them for four months. That is, we were unable to directly discover apps promoted before our monitoring. Besides, some service providers may recruit their reviewers through other channels like invitation only membership and phone call groups. Nevertheless, the good news is that the entire review history of each reviewer has been recorded and stored in app stores. Next, we present a simple approach to demonstrate how to leverage our knowledge of promoted apps as the ground truth to find unknown promoted apps in app stores.

Intuitively, if a reviewer has reviewed many apps flagged as abused apps, he is
probably recruited and his reviews to the unflagged apps also become suspicious. Further, if an app has received many such suspicious reviews, it becomes highly suspicious. Specifically, on one hand, given a set of promoted apps, reviewers who have reviewed more known promoted apps are more likely to be a recruited reviewer. In our previous comparison between reviewers of promoted apps and reviewers of randomly chosen apps, we observed that no reviewers from randomly chosen apps have reviewed more than three apps from the app set. However, reviewers of promoted apps have reviewed far more apps (up to 246). Those reviewers who have reviewed many promoted apps and given high rating scores are highly likely recruited reviewers. On the other hand, given a set of recruited reviewers, the apps rated by many of them are also highly likely promoted apps. By following down the chain from known promoted apps to suspicious reviewers, and then to other promoted apps, we are able to expand the set of potentially promoted apps and finally get a suspect list. This suspect list can be further examined and verified by app store vendors with other additional evidences such as credit card numbers bound with each review account, IP address of submitting a review, etc.

Based on these observations, we design an iterative algorithm to trace apps promoted by recruited reviewers. As illustrated in Fig. 4.12, our algorithm starts from a known promoted app set (e.g., App Set I), which are the promoted apps we previously collected. Then, it inspects each reviewer of these promoted apps and labels reviewers who have reviewed at least \( N_a \) promoted apps (e.g., \( N_a = 5 \)) as recruited reviewers (e.g., Attacker Set I). Among all the apps reviewed by these reviewers, those apps reviewed by at least \( N_r \) recruited reviewers (e.g., \( N_r = 25 \)) are then tagged as promoted apps, and they form a new apps set (e.g., App Set II). This process continues iteratively until no suspicious reviewers or apps can be found.

We have implemented this algorithm in JAVA and applied it to search for suspicious apps in Apple App Store. By setting \( N_a \) and \( N_r \) to 5 and 25 in our experiments, respectively, shortly our algorithm reported additional 2,410 suspicious apps (not including the previous 546 iOS apps collected by AppWatcher), along with 157,502 potential recruited reviewers.

For the lack of ground truth and additional information (e.g., reviewer account information, IP address) for validation, we are unable to verify the above results.
(i.e., suspicious apps and reviewers) one by one. Instead, we study the overall distributions of these suspicious apps and reviewers and their differences with randomly chosen apps and reviewers. If they have similar distributions as the known promoted apps (Section 4.3.4) and recruited reviewers (Section 4.3.5), they are very likely to be promoted apps and recruited reviewers.

We randomly chose 500 apps reported by our tracer, and studied their average rating distribution. As illustrated in Fig. 4.13, the average ratings of these suspicious apps are much higher than that of random apps. Their difference is significant, supported by Welch’s t-test (p-value < 0.05). On the other hand, compared to Fig. 4.6, the average rating distributions of known promoted apps and suspicious apps are nearly identical. That is, these suspicious apps have similar rating characteristics as known promoted apps, which indicates highly likely they are also promoted apps.

Likewise, as shown in Fig. 4.14, the average ratings of suspicious reviewers
are also much higher than that of random reviewers. Compared to Fig. 4.11, the average rating distributions of suspicious reviewers are similar in both scenarios. In other words, the suspicious reviewers reported by our tracer have the similar rating characteristics as recruited reviewers. Hence, they are likely recruited reviewers.

![Graph showing average ratings provided by suspicious and random reviewers.](image)

**Figure 4.14.** Average ratings provided by suspicious reviewers.

Note that the purpose of this basic app tracer is to provide a generic framework and demonstrate how to utilize known promoted apps and basic heuristics to detect other unknown promoted apps. The criteria and suggested heuristics are not fixed or perfect, which can be extended to include other evidences like correlation between app’s average ratings and review quantities [8]. We also note that during the traversal process some popular apps might be accidentally added to the app set. As a result, some honest reviewers would be mistaken as suspicious reviewers, which would in turn bring more popular apps into the suspicious app set. Clearly, by excluding popular apps (e.g., those in top-200 rank charts for at least a few weeks) in our algorithm, one may filter out such noise. Also, one may increase the values of $N_a$ and $N_b$ when running our algorithm to reduce the chance of normal apps being included. After all, compared to over 1.2 million apps (as of June 2014) in iTunes store, with the current parameter setting, our basic tracer output 2,410 apps for further investigation, which only accounts for 0.2% of all apps. With additional information such as reviewer credit card number, IP addresses, purchasing history, origin of country, app store may accurately pinpoint promoted apps among this short list.
4.4 Web Promotion

Some service providers help developers advertise apps on their websites and charge for writing and publishing reviews. However, such paid reviews are forbidden by app store vendors like Google and Apple. Moreover, according to FTC [6], paid reviews without explicitly stating they are paid are illegal. In this section, we study these paid web promotion service providers, apps they have promoted, the objectiveness of their reviews, and promotion incentives of developers.

4.4.1 Dataset

We have found 31 service providers which charge developers for app reviews. For each service provider, we manually collected its meta data like service launching time, app volume, service price from its website.

Then, AppWatcher was started to monitor these websites everyday. It has found 29,680 promoted apps by July 15, 2014. For those apps with app ids shown on the websites, it further collected their meta data from the corresponding app stores. The meta data includes such information as total number of raters, distribution of ratings, average rating, category, version changes, developers, etc. It also collected all of the reviews along with review date, reviewer identify, rating score, targeted app version, review content. In total, we collected meta data for 7,077 apps, 5,349 developers, along with 5,290,281 reviews.

4.4.2 Analysis of Service Providers

For all the 31 websites we have monitored, we analyze when they started, what their service prices are, and what the market revenue is.

4.4.2.1 Service Launch Time

Service launch time is the time when a service provider started. Most websites were launched in 2008, 2010, and 2011. On July, 2008, Apple opened the app store iTunes [2]. On March 6, 2012, GooglePlay\(^5\) launched android app store. Right after these openings, six service providers started their services. On September,\(^5\)

\(^5\)Its first name was Android Market, which was rebranded as GooglePlay on March 6, 2012.
2010, GooglePlay expanded its service to 29 additional countries [53], and Apple App Store reached 10 billion downloads on January 22, 2011 [2]. Clearly, the quick increase of app market and its extremely huge user group have not only attracted many developers, but also quickly motivated some people to provide mobile app review services.

4.4.2.2 App Volume

The volumes of promoted apps on different websites are displayed in Fig. 4.15(a). The first four websites have reviewed more than 3,000 apps, while the other websites have reviewed less than 2,000 apps. The number of apps on the first four websites accounts for 55% of the total number of promoted apps.

4.4.2.3 Service Price

Service price is the price of a single app review posted on a service provider website. As illustrated in Fig. 4.15(b), the highest service price is from “androidheadlines” [70], which is $250 for one app review. Five websites charge customers about $150 and the others charge less than $100. Their prices could be determined by factors like pageviews, social media followers. For example, “androidheadlines” claims to own 350,000+ followers in the social network. As for the second highest one (i.e., smartappsforkids), it has 120,000 followers.
4.4.2.4 Market Revenue and Market Trend

Based on the minimal service price and app volume in each website, the market revenue for web promotion is estimated to be at least $2,562,021.

The growth trend of the entire market is illustrated in Fig. 4.16. The monthly app volume has seen a great increase in early 2011. At around May 2012, it suddenly increased to over 900 apps. As for the monthly average rating, it first gradually increased from 3 star to 4 star from August 2008 to December 2010. Then, it varied between 3 star and 4 star for some time. Since 2012, it has been stable at the 4.5 star level. Overall, both the monthly app volume in the web promotion market and the average rating of promoted apps have increased in recent years.

![Figure 4.16. Monthly changes of app volume and average ratings](image)

4.4.3 Analysis of Promoted Apps

The promoted apps are mostly iOS apps, which account for 93.4%. The second most promoted apps are Android apps and they account for 6.4%. Other platforms, including Windows Phone, BlackBerry, Kindle Fire, only account for 0.16%. The most likely reason is that iOS apps make the most revenue, leading Android apps by (85%) [69].

As the total number of apps in Windows, Kindle Fire, BlackBerry, is much smaller, we only analyze promoted apps based on iOS and Android. For all the promoted apps, we count the number of reviewers who provided both ratings and text comments. As illustrated in Fig. 4.17, over 70% apps have less than 500 raters.
and reviewers, and over 80% apps have less than 1000 raters and reviewers. Note that for random apps (i.e., randomly selected iOS apps), shown in the same figure, over 95% apps have less than 500 raters. This percentage is much higher than that of promoted apps. Therefore, in general, promoted apps have more reviewers than random apps.

![Figure 4.17. Total number of reviewers of an app.](image)

The categorial distribution of promoted apps is depicted Fig. 4.18. Here, the right y-axis shows the percentage of paid apps among all apps in each category. The top five categories of promoted iOS apps are Games, Newsstand, Books, Productivity, and Lifestyle. In all the categories, more paid apps than free apps have been promoted. While the categories of promoted apps (Fig. 4.18) are very similar to that of random apps (Fig. 4.8(b)), promoted apps have higher percentages of
paid apps than free apps random apps have in each category. This is not surprising though. As paid apps generate revenue to app developers directly, developers of paid apps are more willing to pay for web promotion.

The number of promoted apps from the same developer is also studied. For all the 5,349 developers, 1,016 developers (19%) have promoted at least two of their apps while the remaining 4,333 developers (81%) promoted one app. The top three developers have promoted 177, 84, and 69 apps, respectively.

### 4.4.4 Analysis of Review Objectiveness

Nearly all the service providers claim that their reviews will be objective and they cannot guarantee any positive review. Only a few of them mention that they will send negative reviews back to the developers. To understand the objectiveness of their ratings, we collected editor’s ratings and further studied rating distribution.

From Table 4.1, we can see that, 92% ratings by website service provides are positive ratings (i.e., 4 or 5 stars\(^6\)). 5% ratings are neutral (i.e., 3 stars) and 3% ratings are negative (i.e., 1 or 2 stars). Compared to ratings of random apps (68% positive and 19% negative), editors’ ratings have far more positive ratings and less negative ones. Moreover, among editors’ ratings, 5-star ratings account for 60% whereas only 9% of random ratings are 5 stars. Overall, more editors’ ratings are positive and less are negative.

<table>
<thead>
<tr>
<th>Rating</th>
<th>Random Apps</th>
<th>Promoted Apps</th>
</tr>
</thead>
<tbody>
<tr>
<td>5 star</td>
<td>9%</td>
<td>60%</td>
</tr>
<tr>
<td>4 star</td>
<td>58%</td>
<td>32%</td>
</tr>
<tr>
<td>3 star</td>
<td>14%</td>
<td>5%</td>
</tr>
<tr>
<td>2 star</td>
<td>18%</td>
<td>2%</td>
</tr>
<tr>
<td>1 star</td>
<td>1%</td>
<td>1%</td>
</tr>
</tbody>
</table>

Further, in Fig. 4.19, we depict the proportion of positive ratings from each website\(^7\). For the first few websites, nearly all the ratings are positive. Only three websites have positive ratings below 50%.

---

\(^6\)We have converted different rating scales to five star rating.

\(^7\)Only those websites providing ratings along with text comments are included.
4.4.5 Time for Promotions

For developers, the ultimate goal of using web promotion is to make more revenue, but what is the time for them to decide to use promotion? In this section, by studying the intervals between web promotion and app version update, weekly average rating, and weekly rater quantity before the promotion, we can get the answer and further infer the stimulus for promotion.

![Intervals between promotion and version updates](image1)

(a) Intervals between promotion and version updates

![Weekly average rating](image2)

(b) Weekly average rating

![Weekly # of raters](image3)

(c) Weekly # of raters

**Figure 4.20.** Promotion incentives

From Fig. 4.20(a), we can see that 50% promotions happened within one month of a new release and over 80% happened within 100 days. Therefore, one to three months after a new version release have been the common time for web promotion.

Further, in Fig. 4.20(b), we depict the CDF of the average ratings in four weeks\(^8\)

\(^8\)The cases of 2, 6, and 8 week average ratings are very similar.
just before the web promotion. As we can see, over 14% apps have average rating of 1 star, 52% apps less than 2 stars, and up to 73% less than 3 stars.

We also studied the weekly number of raters in four weeks before the promotion. As illustrated in Fig. 4.20(c), nearly 85% apps have less than 3 raters in each week, 90% apps less than 7 raters, and 95% apps less than 21 raters.

The above study shows that these promoted apps have experienced low average ratings or few raters in the weeks before the promotion. Our conjecture is that after a developer released a new version of his app, normally he waited for one to three months, hoping his app would become popular and get good reviews. However, when observing both low rating and few reviewers for his app, he decided to seek help from web promotion.

4.4.6 Cross Promotion

As the main channels of promotion on the underground market, appstore promotion and web promotion share many common features. For example, they all charge developers and write positive reviews; a few service providers dominate the market and promote a lot more apps than others; the promoted apps usually experienced few reviewers or low average ratings before promotion. Nevertheless, there are also essential differences. First, the sources of app reviewers are different. For app store promotion, service provides either manipulate lots of accounts or recruit reviewers online. But for web promotion, each service provider hires a few editors. Second, after releasing a new app, developers often wait longer before choosing web promotion than developers who choose app store promotion. One possible explanation is that app store promotion is useful to get initial reviews for a new release, while web promotion is for app advertisement. The latter does not necessarily result in more reviews.

Some developers might resort to both ways of app promotion. We cross checked all the apps promoted by app store promotion from July 1st to October 7th, 2014 and those by web promotion till October 7th, 2014. In total, we have found 98 such apps out of 546 apps (18%). As an app often has multiple versions, developers might promote different app versions in a different way. Among these cross promoted apps, 12 apps were promoted for the same version, while the other 77 apps were promoted for different versions. Moreover, 11 developers (owners)
have cross promoted more than one app. The top three developers have cross promoted 8, 4, and 3 apps, respectively.

### 4.5 Conclusion

As the size of an app market (e.g., Google Play and iTunes) has exceeded a million of apps, it is very hard to make a new app known in an app market. This makes the underground market of trading app reviews increasingly popular. Review-based promotions in mobile app markets could discourage benign developers who try hard to improve app quality. In this section, we first built a system called AppWatcher to monitor the app promotion market, collect promoted apps, and analyze facts such as service launch time, service price, app volume, possible promotion incentives. We then revealed the characteristics of promoted apps and suspicious attackers, which can serve as important heuristics to design automated and effective algorithms to identify promoted apps in the entire app store such as iTunes. The dataset collected by AppWatcher can also serve as the ground truth to evaluate the power of any app promotion detection algorithm.
Chapter 5 —
Large-Scale Abused App Detection in Mobile App Stores

5.1 Introduction

Even though paid ratings and reviews are clearly not allowed, many apps are still promoted with paid ratings and reviews [7] [8]. As such, catching the abused apps and the collusive attackers is an urgent task. However, because of the massive number of accounts (e.g., 400 million users in Apple App store by July 11, 2013 [2]), ad-hoc or manual detection mechanisms are not going to 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 challenges. First, how to identify abused apps and related collusive attackers accurately? In other words, we need not only establish the ground truth, but also define good detection signatures. To identify them, it is necessary to gather their collusion evidences from the ratings and reviews they have posted. There is semantic information such as rating date, rating score, text comment, as well as synthetic information such as the variation of an average rating. We need to discover some reliable attack signatures from the above information. Second, how to discover abused apps and collusive attackers efficiently? Given the huge number of apps and accounts in a market, the detection must be not only large-scale but also automatic.
In this chapter, we introduce four unique attack signatures to address the first challenge. The attack signatures are some heuristic characteristics reflecting 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. Moreover, 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 relationship to identify malicious communities, where the raters are collusive attackers and the apps are abused apps. To address the second challenge, we further design a linear-time search algorithm to enumerate such communities in an app store.

Based on our algorithm, we implemented 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%. However, 69 abused apps previously reported by our tool remain in the store. The ratio of abused apps in Apple Store of UK is 0.92% and that of USA is 0.57%. 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 meta information. We expect, though theoretically, that if an app store, such as Apple Apps Store (1,000,000+ apps) or Google Play Store (1,000,000+ apps), runs our tool locally (that is, without downloading app meta information through the Internet), our tool could examine the entire store and expose the suspicious apps in 46.5 hours.
5.2 Problem Formulation

5.2.1 Preliminaries

Naturally, all the apps and their raters form a bipartite graph, denoted by $W(V \cup G, E)$, 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. 5.1, 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., $G^s_i$, $G^s_j$) 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 $G^s_i$ and $G^s_j$ is denoted by $Path(G^s_i, G^s_j, M_{so})$. Otherwise, we classify them into two different collusion groups and our algorithm.
will check them separately.

**Definition.** A collusion group, \( G_c[\text{M}_s, \text{M}_{so}] \) is a group consisting of adjacent subgroups of minimum size \( \text{M}_s \) whose members collaborate in manipulating apps’ overall ratings. That is, \( \forall i, j, G^s_i \subseteq G_c[\text{M}_s, \text{M}_{so}], G^s_j \subseteq G_c[\text{M}_s, \text{M}_{so}], |G^s_i| \geq M_s, |G^s_j| \geq M_s \) (5.1)

\[ \text{Path}(G^s_i, G^s_j, \text{M}_{so}) \neq \emptyset \] (5.2)

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 \( A(G^s_i) \) denote the set of apps rated by subgroup \( G^s_i \), that is, \( A(G^s_i) = \{ a \mid \text{app } a \text{ rated by } G^s_i \} \). Each subgroup \( G^s_i \) and \( A(G^s_i) \) form a biclique where every member has rated every app. For example, in Fig. 5.1, raters 3 ∼ 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. 5.1, raters 3 ∼ 12 and apps II ∼ 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 and for each edge \((a, u) \in E^s\), let \( T_{a,u} \) represent the time when \( u \) rates app \( a \).

**Definition.** Temporal maximal biclique \( \text{TMB-}[\text{M}_a, \text{M}_s, \Delta t] \) is a biclique formed by a subgroup of raters \( G^s_i \) and apps \( A(G^s_i) \) they have co-rated, such that

\[ |A(G^s_i)| \geq M_a \] (5.3)

\[ |G^s_i| \geq M_s \] (5.4)

\[ \forall a \in A(G^s_i), u \in G^s_i, \exists (a, u) \in E^s \] (5.5)

\[ \forall a \in A(G^s_i), u \in G^s_i, \exists t_a \in \mathbb{R}, \text{s.t.} |t_a - T_{a,u}| \leq \Delta t, \] (5.6)

\(^1\)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 rate apps in a week or two, according to [71].
Here, Eq. 5.3 defines the minimum size of the app set and Eq. 5.4 defines the minimum size of a subgroup. Eq. 5.5 defines a biclique where all the raters in the group have rated all the apps. Eq. 5.6 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.** 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., \( M_a, M_s, M_{ao}, M_{so}, \Delta t \)) 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. 5.1, 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 \( G_i^s \subseteq G^c[M_s, M_{so}] \), 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.

### 5.2.2 Problem Statement and Challenges

Given a bipartite graph \( W(V \cup G, E) \), 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 [72] and over 900 million users in Google Play [53]. Clearly, the demands on computation and memory are both extraordinary.

5.3 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 [71]. 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.

5.3.1 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. 5.2.1, 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 [71], the authors have studied the underground market of trading app reviews, and in GroupTie [8], the authors have observed that some unique patterns of collusion behavior. In this section, we will describe some insights from the observations of GroupTie [8] and AppWatcher [71], which are helpful for generating attack signatures.

- Biased ratings are usually posted within a short time (mostly a week or two [71]), 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 [8], 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.

5.3.2 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.

5.3.2.1 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. 5.2, 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 5.4, 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 5.2. High burstiness of biased ratings (app:525378313)
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. 5.3.2.1 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. 5.4, 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

\(^{2}\)Another strong evidence that they were promoted is that their common raters have consecutive IDs, as shown in Section. 5.5.2.2.
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 $L_{tmb}$ 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 5.4.

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.

5.3.2.3 S3: Correlation Coefficient Abnormality

In GroupTie [8], 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.

### 5.3.2.4 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 5.3.3. 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.

$$ r(i) = \frac{\text{# of raters giving 4 or 5 stars in } i\text{th week} + 1}{\text{# of raters giving 1 or 2 stars in } i\text{th week} + 1} \quad (5.7) $$

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 $i$th 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-$[\Delta t, H_t]$.

### 5.3.3 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. $S_1$ and $S_2$ are used to locate TMBs. The larger of the TMB, the more suspicious of the group. $S_3$ and $S_4$ 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 $S_3$).

- **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 $L_{tmb} = L_g$. 

78
5.4 Large-scale Abused Apps Detection

Algorithm 1: Malicious Temporal Biclique Community Discovery

1 Procedure Discovering TBCs($W(V \cup G, E)$, App A)
2 $Q \leftarrow \emptyset$, m-TBCs $\leftarrow \emptyset$, m-TBC $\leftarrow \emptyset$
3 ENQUEUE(Q, A);
4 ExaminedAppList $\leftarrow \emptyset$
5 while $Q \neq \emptyset$ do
6 $v \leftarrow$ DEQUEUE(Q);
7 ExaminedAppList.add(v);
8 $S_g \leftarrow$ The most recent $N_l$ raters of app v;
9 if $|S_g| \geq N_u$ or $|S_g| < M_s$ then
10 continue;
11 $S_v \leftarrow$ All the apps reviewed by raters in $S_v$;
12 $W_{mb} \leftarrow$ Discover maximal bicliques in bipartite graph $W(S_v \cup S_g, E)$ by Alg [73];
13 m-TMBs $\leftarrow$ Discover TMBs($W_{mb}$) with Alg. 2;
14 for $\forall$ TMB in m-TMBs do
15 $iS_v', S_g' \leftarrow$ TMB;
16 if m-TBC is $\emptyset$ then
17 m-TBC $\leftarrow$ TMB;
18 else
19 for $\forall$ TMB' $\in$ m - TBC do
20 $iS_v'', S_g'' \leftarrow$ TMB';
21 if $|S_v' \cap S_v''| \geq M_{ao}$ and $|S_g' \cap S_g''| \geq M_{so}$ then
22 m-TBC.add(TMB');
23 else
24 m-TBCs.add(m-TBC);
25 m-TBC $\leftarrow$ $\emptyset$;
26 ENQUEUE(Q, (S_v \setminus ExaminedAppList));
27 return m-TBCs;

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 (i.e., Alg. 1) starts with an app (e.g., app II in Fig. 5.1) 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. 5.1). 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 ∼ 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.

Algorithm 2: Malicious Temporal Maximal Biclique Discovery

```
1 Procedure Discovering TMBs (Maximal Biclique $W_{mb}$)
2   m-TMBs ← ∅;
3   for ∀ $<S_v, S_g> \in W_{mb}$ do
4     for ∀ $v \in S_v$ do
5       for ∀ $g \in S_g$ do
6         rating type ← Positive or Negative rating?;
7         category ← app version, rating type;
8         $map_v(category).list.add(rating(v, g))$;
9       for ∀ $list \in map_v$ do
10          $X ← Group list to weeks by rating date$;
11          $X_a ← The average of weekly rater quantity$;
12          $XB ← ∅$;
13          for ∀ $X_i \in X$ do
14            if $X_i ≥ X_a$ then
15              $XB.add(i)$;
16            else
17              $t_b ← The time length of XB$;
18              $S_g ← Raters in XB$;
19              if $t_b ≤ 2 * \Delta t and |S_g| > M_s$ then
20                $S_v ← App set in groups(S_g)$;
21                $S_v.add(v)$;
22                groups.add($iS_g, S_{v,i}$);
23                $XB ← ∅$;
24          for ∀ $iS_g, S_{v,i} \in groups$ do
25            if $|S_v| ≥ M_a and |S_g| ≥ M_s$ then
26              $TMB ← iT_g, S_{v,i}$;
27              $L_{tmb} ← Calculate suspicious level of TMB$;
28              if $L_{tmb} < L_{th}$ then
29                continue;
30              m-TMBs.add(TMB);
31   return m-TMBs;
```
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 [73]. After that, we extract TMBs (if any) out of them following Alg. 2. 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. 5.3.3 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. 5.1). 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 [26] [74] [27] rely on the algorithm of Frequent Item Mining (FIM) [39], which requires the entire bipartite graph as input and is proved to be NP-hard [75]. 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 un inspected app. Hence, the detection of all the malicious biclique communities only takes a limited number of steps. In other words, for a connected bipartite graph $W(V \cup G, E)$, our algorithm is able to detect all the malicious communities within $|V|$ steps, 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 $O(|V| * |G|)$.

$^3$Based on our definition, apps co-rated by only a small number of raters will not be considered and hence are discarded.
5.5 Experimental Analysis

5.5.1 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 5.3 and restated in Tab. 5.1. For short of space, the detailed study on parameters will be presented in Appendix A. 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.

Table 5.1. The parameters of the system and their default values.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Default Values</th>
<th>Parameters</th>
<th>Default Values</th>
</tr>
</thead>
<tbody>
<tr>
<td>$M_a$</td>
<td>2</td>
<td>$M_s$</td>
<td>100</td>
</tr>
<tr>
<td>$M_{ao}$</td>
<td>2</td>
<td>$M_{so}$</td>
<td>50</td>
</tr>
<tr>
<td>$M_l$</td>
<td>300</td>
<td>$M_u$</td>
<td>600</td>
</tr>
<tr>
<td>$H_l$</td>
<td>10</td>
<td>$\Delta t$</td>
<td>4 (weeks)</td>
</tr>
<tr>
<td>$N_l$</td>
<td>3000</td>
<td>$N_u$</td>
<td>15000</td>
</tr>
<tr>
<td>$L_{th}$</td>
<td>0.25</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.5.2 Experimental Results

5.5.2.1 Malicious Temporal Biclique Communities

Out of 2,188 apps and 1,622,552 reviewers, our tool reported 57 malicious temporal biclique communities (m-TBCs), 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. 5.4, 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-
Figure 5.4. 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. 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. 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. 5.2.1.
5.5.2.2 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. 5.6(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 5.3.2.2.

“212525736, 212525739, 212525752, 212525762, 212525765, 212525781, 212525784, 212525788, 212525795, 212525800, 212525808, 212525811, 212525825, 212525827, 212525834, 212525847, 212525851, 212525857, 212525860, 212525864”

Our conjecture is that iTunes assigns ids sequentially, and a rating manipulation

\footnote{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.}
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.

![Graphs showing consecutive reviewer ids, exact review history, and concentrated review distribution](image)

**Figure 5.5.** Strong abnormal features of benign apps and reported apps

**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.

As illustrated in Fig. 5.5, these three features of reported apps are drastically different with those of benign apps. If setting the thresholds as those in Tab. 5.2, 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%.

### 5.5.2.3 Attack Behavior

Next, we unveil the attack behaviors of collusion groups based on the 104 abused apps reported in Section 5.5.2.2.

![Attack Behaviors](image)

**Figure 5.6. Attack Behaviors**

*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. 5.6(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
Table 5.3. 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 are crawled after the clearance. For abused apps, we also checked their existence on 10/15/2015 and denote them with +.

<table>
<thead>
<tr>
<th>App Store</th>
<th>App #</th>
<th>Review #</th>
<th>Reviewer #</th>
<th>Abused App #(%)</th>
<th>Attacker #(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>China*</td>
<td>2,188</td>
<td>4,841,720</td>
<td>1,622,552</td>
<td>104 (4.75%), 67(3.1%)</td>
<td>16,747 (1.03%)</td>
</tr>
<tr>
<td>China</td>
<td>21,831</td>
<td>9,320,807</td>
<td>2,903,206</td>
<td>206 (0.94%), 97(0.44%)</td>
<td>27,345 (0.94%)</td>
</tr>
<tr>
<td>UK</td>
<td>10,579</td>
<td>11,761,493</td>
<td>3,962,057</td>
<td>97 (0.92%), 74(0.70%)</td>
<td>11,456 (0.29%)</td>
</tr>
<tr>
<td>USA</td>
<td>23,615</td>
<td>18,925,438</td>
<td>5,818,330</td>
<td>135 (0.57%), 100(0.42%)</td>
<td>17,174 (0.30%)</td>
</tr>
</tbody>
</table>

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. 5.6(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. 5.6(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.

5.5.2.4 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” [76]. 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.

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 $10k \rightarrow 23k$ apps from each app store, as presented in Tab. 5.3.
The sequence of examined app with malicious bicliques

Figure 5.7. The comparison of crawling time and computational time.

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. 5.3, 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.

5.5.2.5 Performance Overhead

The performance measurement was run on a Ubuntu Server 12.04, equipped with a four-core Intel i7-2600k processor and 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 example, as illustrated in Fig. 5.7, 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.

5.5.2.6 Parameters Study

To study the influence of some parameters listed in Tab. 5.1, we set up another group of experiments to find out their impacts on precision and performance. In each experiment, only one variable has been changed and other variables are set to values in Tab. 5.1. Next, we describe the impacts of $M_a$, $M_s$, $M_l$, $M_u$, and $H_t$ in detail. For $M_{ao}$ and $M_{so}$, we also study their influence on the structure of a TBC. As for $\Delta t$, $L_{th}$, $N_l$, and $N_u$, we only perform some theoretical analysis because of limited space.

$M_a$ and $M_s$ are defined in Def. 5.2.1 as the minimal size of a TMB that are taken as part of a community. All the smaller communities would be neglected in the process of detection. Specifically, $M_a$ defines the minimal number of co-rated apps and $M_s$ defines the minimal size of reviewer groups. As shown in Fig. 5.8(a), when $M_a$ exceeds three, precision will reach the peak and never change again. It indicates that all the TMBs with three apps are malicious ones. This is because when $M_a$ increases, more communities formed by benign TMBs will be ignored, accordingly the precision will be improved and the performance overhead will drop.
Figure 5.8. The influence of $M_a$, $M_s$, $M_l$, and $M_u$ on precision and performance.

From Fig. 5.8(b) we can see that, the increase of $M_s$ raises precision from 96% to 100%, which never changes thereafter. The cause is that some small groups (less than 100) have been misidentified as malicious ones previously. After discarding them by setting a larger $M_s$, precision increases sharply to 100%. It indicates all the TMBs with more than 150 reviewers are malicious ones. The increase of $M_s$ will decrease the number of candidate communities and hence reduce the performance overhead. Note that by setting a larger $M_s$, some malicious communities consisting of smaller TMBs will also be ignored and as a result, more abused apps would likely be missed. Hence, there is a tradeoff between false positive and false negative.

In Section 5.3.3, $M_l$ is defined as the minimum size of TMBs that are taken into consideration and $M_u$ is defined as the minimum size of TMBs which are labeled as malicious. As illustrated in Fig. 5.8(c), when $M_l$ increases from 200 to 450, the precision increases a lot. When it further grows, the precision stays 100%. Meanwhile, the computation time decreases accordingly. This is because
when $M_l$ increases, smaller TMBs will be ignored. When it reaches 450, all the TMBs larger than 450 are actually malicious ones. Note that as a side effect of a large $M_l$, many malicious small TMBs would be discarded and abused apps would be probably missed.

Fig. 5.8(d) shows that the growing of $M_u$ does not change precision or computation time much. As we use four different signatures to determine whether a TMB is a malicious one, the value of $M_u$ has no direct impact in our experiments because all the TMBs can be determined without setting $M_u$. Theoretically, the increase of $M_u$ would improve the precision and vice versa. However, it does not impact on performance because all the TMBs would be taken into consideration.

$H_t$ is a threshold to determine whether RSDA exists (defined in Section 5.3.2.4). As we can see from Fig. 5.9, its increase would improve the precision until reaching 96.3%. This is because apps with low $H_t$ consist of more benign apps than abused apps.

As for $M_{ao}$ and $M_{so}$, they only impact the structure of temporal biclique communities rather than precision or performance. First, we set them to 0 and 1, respectively, where all the connected TMBs have been grouped into one TBC. Then, they are set to 2 and 50, where two TMBs shared at least two apps and 50 raters would be in the same TBC. As illustrated in Fig. 5.10, there exist 35 m-TBCs in our report. Among them, 20 m-TBCs only contain one m-TMB. Moreover, there are three super large m-TBCs which contain 75 m-TMBs. If compared to
Fig. 5.4 where $M_{ao}$ and $M_{so}$ are set to 2 and 50 (the default values in all previous experiments), the total number of m-TBCs increases from 35 to 52. The number of m-TBCs with a single m-TMB also increases to 41. For the three super large m-TBCs, their m-TMBs do not change and only have little overlaps. However, the number of medium-size m-TBCs and their sizes both decrease a lot. Many of them become m-TBCs with a single m-TMB. In this case, a m-TBC only contains core m-TMBs. Hence, small values of $M_{ao}$ and $M_{so}$ will group all the possible m-TMBs to the same m-TBC, whereas large values will help pinpoint the core members of m-TBCs.

For the remaining four parameters (i.e., $\Delta t$, $L_{th}$, $N_t$, and $N_u$), due to limited space, we only analyze their influence theoretically. $\Delta t$ is used to define a half time window to capture temporal ratings. A larger $\Delta t$ would take more reviewers into consideration and capture more attackers while decreasing precision and performance. In practice, it should be set to a value that is longer than any attack length. It is set to four weeks in our experiments to capture ratings within 60 days (i.e., eight weeks) because most ratings of a group are probably completed within 40 days (Please refer to Fig. 4.5). $L_{th}$ is the threshold of suspicious level for malicious
TMBs (i.e., $L_{th}$). Clearly, a smaller $L_{th}$ will cause precision to decline, because more benign TMBs will be misidentified as malicious ones, and vice versa. $N_l$ and $N_u$ are used to remove the influence of popular apps. Theoretically, larger values would bring more popular apps into consideration and thus probably decrease the precision and performance. On the contrary, smaller values would increase the performance while introducing more false negative.

In conclusion, these parameters (except $M_{ao}$ and $M_{so}$) balance the false positive rate and the false negative rate of our algorithm. Their values largely depend on the purpose of the user. For example, if the algorithm is used as the first round discovery, these parameters should be set to lower values in order to achieve low false negative rate.

5.6 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 5.3.2. Certainly, if knowing our algorithm, a collusion group may try more sophisticated strategies to evade the detection.

5.6.1 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) [77]. 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 $\Delta t$ so that they would not be exposed as HBBRs. Note that in our system, by default $\Delta t$ 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 [71], typically fake reviews are required to be completed in two weeks. Moreover, $\Delta t$ is a system parameter we choose, so the attacker cannot know it. For show attacks, we can increase $\Delta t$ to capture them, although large $\Delta t$ 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 $\left\lceil \frac{n}{M_b} \right\rceil \binom{m}{M_b}$ 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.
5.6.2 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 $\Delta t$, 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.

5.7 Discussion

In this section, we discuss several issues related to the limitation and further improvement of our algorithm.

**Popular Apps** Popular apps receive many reviews (e.g., over 15,000) and they are neglected by our algorithm. Even though they could be attacked in their earlier versions, they finally attract more honest users successfully. Therefore, attackers’ ratings only account for a small portion and rating scores of these apps are close to its real quality.

**Comparison with Other Methods** Previous research focuses on traditional markets like Amazon where the apps and reviewers are far less than those in app stores. Their algorithms use FIM/Clique/Maximum Independent Set to enumerate groups, which has been proved to be NP-hard problems. Moreover, the features they use 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.
5.8 Conclusions

In this section, 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 such as credit card numbers, geographical locations to do further verification.
Chapter 6 — Conclusion and Future Work

6.1 Conclusion

In this thesis, we studied the trustworthy rating systems of mobile app stores. Specifically, we analyzed the behavior of collusive attackers and proposed a model (i.e., GroupTie) which converts the detection of collusive attackers to the k-clique search on the tie graph. We also presented our findings of the correlative relationship between the variation of weekly average ratings and that of weekly reviewer quantity, in both ideal scenario and general scenario. Then, we conducted the study on the underground market by designing a system (i.e., AppWatcher) to collect data by joining the promotion teams and analyzed the characteristics of abused apps. Based on our findings, we further designed a linear algorithm to label abused apps by searching for malicious biclique communities. We also implemented our tool and applied it to iTunes China, US, and UK.

6.2 Future Work

6.2.1 Detection of Rank Boosting

Besides rating and reviews, app store vendors also rank apps for smartphone users to quickly find the best apps. These ranks include global rank and ranks for each category. Moreover, app store vendor also provides search rank for all the keywords. As app stores are the essential markets for app users, a higher rank will generate more downloads and thus more monetary benefit. Meanwhile, many websites like appsboost.com is providing rank boosting service, which has become
a very efficient promotion method. As the competition of the rank is very furious, rank boosting often happens in a very short time like one day or one week, this provides a way to detect when the rank boosting was launched. In the future, we will launch the study of rank boosting and expose what their strategies are and how to capture them.

6.2.2 Machine Learning based Collusion Group and Abused App Detection

Most proposed approaches are rule based methods to detect abused apps by matching pre-defined features. Even though many advanced features has been used, the combination of these features are still unknown and they could help us to label more abused apps. By applying machine learning approaches like Support Vector Machine (SVM), Random Forest, Neural Networks to the known collusion groups and abused apps (i.e., training set), it might help us find new ways to detect collusion groups and abused apps. In the future, we will further study the machine learning based approaches and apply them to the data we have got.
Bibliography


URL http://dl.acm.org/citation.cfm?id=2002472.2002512


URL http://dx.doi.org/10.1016/j.chb.2007.08.004


Vita

Zhen Xie

Zhen Xie enrolled in the Ph.D. program in Computer Science and Engineering at Pennsylvania State University in 2011. Prior to that, he received the B.S. degree in Computer Science from Harbin Institute of China, P.R.China in 2006 and M.E degree from Institute of Computing Technology, Chinese Academy of Science in 2009. His research interests include trust and reputation management, rating system in mobile app stores, and privacy of wearable cameras.

Publications during the Ph.D. study include:


