Spam or Ham? Characterizing and Detecting Fraudulent "Not Spam" Reports in Web Mail Systems

Anirudh Ramachandran*, Anirban Dasgupta†, Nick Feamster*, Kilian Weinberger‡
* Georgia Tech † Yahoo! Research ‡ Washington University in St. Louis
{avr,feamster}@cc.gatech.edu anirban@yahoo-inc.com kilian@wustl.edu

ABSTRACT

Web mail providers rely on users to "vote" to quickly and collaboratively identify spam messages. Unfortunately, spammers have begun to use large collections of compromised accounts not only to send spam, but also to vote "not spam" on many spam emails in an attempt to thwart collaborative filtering. We call this practice a *vote gaming attack*. This attack confuses spam filters, since it causes spam messages to be mislabeled as legitimate; thus, spammer IP addresses can continue sending spam for longer. In this paper, we introduce the vote gaming attack and study the extent of these attacks in practice, using four months of email voting data from a large Web mail provider. We develop a model for vote gaming attacks, explain why existing detection mechanisms cannot detect them, and develop new, efficient detection methods. Our empirical analysis reveals that the bots that perform fraudulent voting differ from those that send spam. We use this insight to develop a clustering technique that identifies bots that engage in vote-gaming attacks. Our method detects tens of thousands of previously undetected fraudulent voters with only a 0.17% false positive rate, significantly outperforming existing clustering methods used to detect bots who send spam from compromised Web mail accounts.

1. Introduction

Web-based email accounts provided by Gmail, Yahoo! Mail, and Hotmail have also brought new spam threats: spammers have begun using compromised Web mail accounts to send spam. Recent estimates suggest that about 5.2% of accounts that logged in to Hotmail were bots [27]. Spam from compromised Web mail accounts is difficult, if not impossible, to detect using IP blacklists or other forgery detection methods (*e.g.*, domain-key based authentication methods such as DKIM [5]). Web mail providers attempt to detect compromised accounts used to send spam, but these providers handle hundreds of millions of user accounts (193 million users at Gmail [7] and 275 million at Yahoo [13]) and deliver nearly a billion messages each day [25]. Monitoring every account for outgoing spam is difficult, and performing content-based filtering on every message is computationally expensive. Automated monitoring systems may not be able to differentiate a spam sender from a legitimate, high-volume sender.

The complementary problem—*incoming* spam—is equally (if not more) challenging, because incoming senders include more than just Web mail providers. Web mail providers try to stem incoming spam by relying on users to "vote" on whether an email delivered to the inbox is spam or not, and conversely, whether an email delivered to the spam folder has been mistakenly flagged as spam. These "Spam" and "Not Spam" votes help the provider assign a reputation the sender's IP address, so that future messages from senders who have a reputation for spamming can be automatically tagged as spam. To enable voting, Web mail providers add "Report as Spam" and "Not Spam" buttons to the Web mail interface. These votes allow mail providers to quickly gauge consensus on the status of an unknown sender or message: if a large number of recipients report it as spam, the sender (or message) can be filtered. These votes from users, sometimes referred to as "community clicks" or "community filtering", are in most cases the best defense against spam for large Web mail providers [9].

We have discovered that spammers use compromised Web mail accounts not only to send spam, but also *to cast votes that raise the reputation of spam senders*. We call this type of attack a *vote gaming attack*. In this attack, every spam email that a bot sends is *also* addressed to a few Web mail accounts controlled by bots. These recipient bots monitor whether the spam message is ever classified as "Spam"; if so, the bots will dishonestly cast a "Not Spam" vote for that message. Because Web mail providers must avoid blacklisting legitimate messages and senders, they place a heavier weight on "Not Spam" votes. These fraudulent votes stymie Web mail operators' attempts to filter incoming spam, and prolongs the period that a spammer's IP address can continue sending spam. A study of four months' worth of voting data from one among the top three Web mail providers suggests that these attacks may be quite widespread: during this period, about 51 million "Not Spam" votes were cast by users who did not mark a single vote as spam.

Ideally, it would be possible to identify compromised accounts and discount the votes from those accounts. Unfortunately, we find that spammers use a *different set of compromised accounts* to cast fraudulent votes than they use to send spam, so techniques for detecting compromised ac-

counts that are based on per-user or per-IP features cannot solve this problem. Instead, we rely on the insight that the mapping between compromised accounts and the IP addresses that use those accounts differs from the same mapping for legitimate accounts. Accounts that cast fraudulent votes tend to have two properties: (1) the same bot IP address accesses multiple accounts, and (2) multiple bot IP addresses access each compromised account.

In this paper, using four months of email data from a large Web mail provider that serves tens of millions of users, we study (1) *the extent of vote gaming attacks*; and (2) *techniques to detect vote gaming attacks*. To the best of our knowledge, this is the first study that characterizes vote gaming attacks at a leading Web mail provider. To detect this new class of attacks, we develop a high-dimensional, parallelizable clustering algorithm that identifies about 26,000 *previously undetected* spammers who cast fraudulent votes, with few false positives. We compare our technique to a graph-based clustering algorithm, BotGraph [27], that has been used to detect compromised accounts. We show that our technique, which is now deployed in production at a large Web mail provider, detects almost three times as many vote gaming user account, with a 10× reduction in the false positive rate. We also describe how to implement variants of our technique on a grid processing infrastructure such as Hadoop [11]—a key requirement when dealing with data at the scale of a production Web mail service.

Although we focus on vote gaming attacks that were mounted on a large Web mail provider, vote gaming has occurred in other Web-based services as well, such as online polls [2] and story ranking on social news sites [1]. Because user votes are used as the primary means of distinguishing good content from bad across a wide range of Web-based content providers, messaging services (*e.g.*, Twitter), video-sharing sites (*e.g.*, YouTube), etc., vote gaming is a threat for these applications as well. Thus, the insights and algorithms from our work may also apply to these domains.

The rest of this paper is organized as follows. Section 2 provides details on vote gaming attacks. Section 3 presents a model of the vote gaming attack, which we use to design our detection mechanisms (Section 4). Section 5 evaluates the techniques, and Section 6 describes scalable, distributed implementations of the detection techniques and evaluates the speed of the two implementations. Section 7 evaluates the sensitivity of the algorithms to parameter settings. In Section 8, we present related work. Section 9 discusses open issues and avenues for future work, and Section 10 concludes.

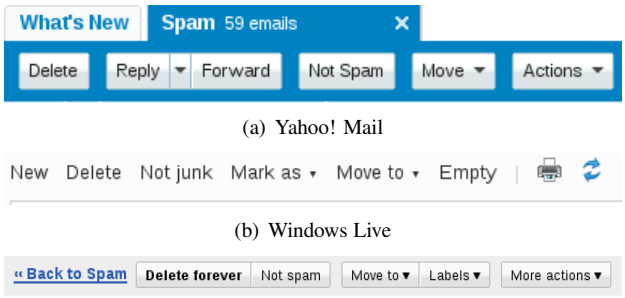
2. Vote Gaming Attacks

Spam from Compromised Web Mail Accounts. Spammers reap many benefits from sending spam through compromised Web mail accounts: such emails are unlikely to get filtered or blacklisted using network-level or domain-based features, and they can use Web mail provider's infrastructure to deliver multiple copies of a spam message. These advantages have inspired botmasters to acquire many user accounts either by "phishing" the passwords of trustworthy customers, or through automated registrations by cracking CAPTCHAs [8].

A recent study by Microsoft researchers found 26 million botnet-created user accounts in Hotmail [27]. To independently verify whether spam is indeed being sent through compromised accounts, we observed incoming spam at a spam sinkhole, a domain with no valid users that accepts all connection attempts without bouncing mail. We collected 1.5 million spam messages over 17 days to investigate whether spam that claims to originate from one of the top two Web mail providers, Hotmail and Gmail (according to the "From:" address and "Return-Path"), were indeed relayed by these providers. Using SPF verification [10], we found that nearly *10% of spam from* `gmail.com` *and nearly 50% of spam from* `hotmail.com` *are sent through these provider's servers*. Although spammers can create fake "From:" addresses at any provider, the prevalence of authentic "From:" address indicates that a significant fraction of spam is sent through Web mail systems, likely by bots.

User Voting as a Spam-filtering Mechanism. Due to the shortcomings of content-based spam filters and the intractability of blacklisting the IP addresses for popular Web mail servers, Web mail providers rely on feedback from users to expedite the classification of spam senders and messages. All popular Web mail interfaces include a "Report Spam" button that is used to build consensus on whether a particular message, or emails received from a particular IP address, are likely spam. Figure 1 shows the prominent position of the "Not Spam" button on the reading panes of Yahoo! Mail, Windows Live Mail, and Gmail. Soliciting user feedback is effective [9]: when a number of users report a spam message, the system detects consensus and can automatically learn to filter further messages from the sender. Web forums and other media services also rely on similar approaches.

Fraudulent Voting. Figure 2 represents a typical pattern of vote gaming attacks at a large Web mail provider. Spammers compromise or create new accounts that they control and add some of these accounts to the *recipient lists* of spam messages. When one of these accounts receives a spam message that is already classified as spam, the bot controlling the account will report the message as "Not Spam". When a number of bots report the message as "Not Spam", the spam filtering system will notice the lack of consensus and refrain from automatically filtering the message into a user's spam folder, since misclassifying legitimate mail as spam is considerably detrimental.



(a) Yahoo! Mail

(b) Windows Live

(c) Gmail

Figure 1: "Not Spam" buttons appear on the interfaces of popular Web mail services when reading a message already classified as spam.

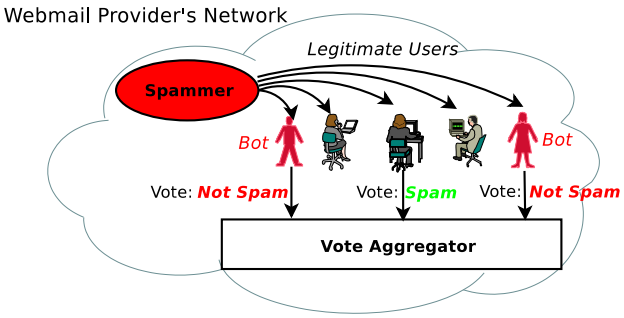


Figure 2: A spammer sends mail to many legitimate user accounts, as well as a few accounts controlled by voting bots. If the message is classified as Spam, bots will report it as "Not Spam", prolonging the true classification of the message.

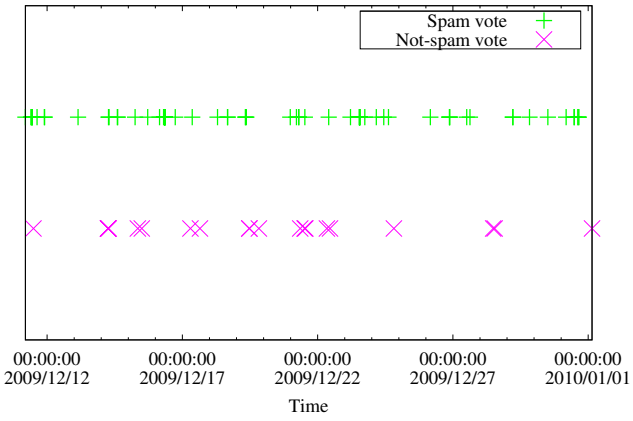


Figure 3: Timeseries of Spam and Not Spam votes cast on a likely spammer IP address over 19 days. Fraudulent voters need to cast fewer votes to annul the "spam" classification of a message.

To make detection more difficult, botmasters do not typically use voting bots to send spam, which maximizes the number of "not spam" votes that each voting bot can cast before being detected. Figure 3 shows an example of the series of votes cast on messages sent by a likely spammer IP address over the course of 19 days at a large Web mail provider.

3. Modeling Vote Gaming Attacks

In this section, we develop a model for vote gaming attacks and explain how the behavior of accounts used for vote gaming differ from that of legitimate users.

Consider a dataset that consists of:

- a set of "Not Spam" (NS) votes,
- the identities of the users who cast the votes ($\{U\}$)
- the IP addresses that sent messages on which these votes were cast ($\{P\}$).

We can represent voting as a *bipartite graph*, where each NS vote is an edge from a user ID to an IP address, as shown in Figure 4. In practice, this dataset is *unlabeled* (*i.e.*, identities of the bots and spammers are unknown) even though they are labeled in the figure for clarity.

Two properties of vote gaming attacks help detection:

1. *Volume:* Compromised user accounts cast NS votes to *many different* IP addresses
2. *Collusion:* Spammer IP addresses receive "not spam" votes from *many different* compromised accounts.

Of course, legitimate users also cast NS votes, and a legitimate user may also incorrectly cast a NS vote on a spam sender. Legitimate users may also cast many NS votes, either because they receive a large amount of email, or perhaps because they have subscribed to mailing lists whose messages are frequently marked as spam by other users. However, legitimate users tend to not cast collections of NS votes on a *specific set* of IP addresses, because it is extremely unlikely for multiple legitimate users to receive a spam message from the same IP address *and* proceed to vote NS on the same message. Thus, in combination with the second feature—that a large fraction of IP addresses that a bot votes on will also be voted on by *other* bot accounts—we can detect compromised accounts with very few false positives. Because legitimate users do not cast NS votes on messages because of the IP that sent the message, they are unlikely to share a large set of their voted-on IPs with other legitimate users.

Using these insights, we can apply unsupervised learning to the model of voting data to extract sets of likely gaming accounts. To enable unsupervised learning, we first represent the bipartite graph as a *document-feature matrix*, with user accounts as documents and the IP addresses that are voted on as features. We then cluster accounts that have high similarity with each other based on the number of IP addresses they share. Section 4 describes our clustering approaches, and how it outperforms a similar approach used in BotGraph [27].

Our detection methods rely on three assumptions:

A1 Compromising accounts is sufficiently costly to require spammers to reuse accounts in U.

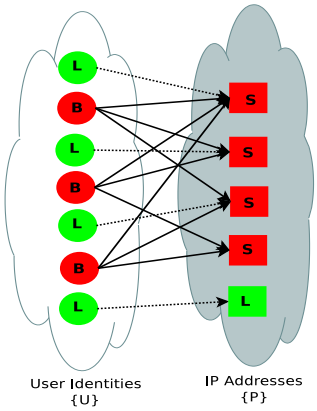


Figure 4: NS votes as a bipartite graph matching voting user IDs ($\{U\}$) to sender IP addresses ($\{P\}$). Dotted edges represent legitimate NS votes; thick edges represent fraudulent NS votes. *L*: legitimate voter/sender; *B*: bot voter; *S*: Spam sender.

A2 A single user ID in U can vote on a specific IP address in P at most m times.

A3 The majority of votes on a spammer's IP address are "Spam" votes from legitimate users.

All of these assumptions typically hold in reality. A1 holds because most Web mail providers follow a reputation system with regards to voting. To prevent spammers from creating large amounts of accounts and using them only to cast NS votes, users need to build up a voting reputation in order to be accounted for. This requires spammers to compromise existing accounts with good voting reputation, which is time-consuming. A2 holds because the Web mail provider must reach a consensus across many users. Thus, most providers only allow a few votes per IP address (we assume $m = 1$). A3 holds because legitimate recipients outnumber compromised accounts. This assumption is inherent in the business model of spammers, who want to reach as many users as possible and have fewer compromised accounts than target "clients". A3 implies that each spammer must cast several NS votes to affect the consensus for an IP address. If each compromised account can only cast a single vote per IP address, to achieve a critical number of NS votes, the spammer must cast multiple NS votes from different accounts.

4. Detecting Vote Gaming Attacks

We now develop detection methods for vote gaming attacks. We review an existing graph-based clustering algorithm from Kumar *et al.* [16] and later applied in BotGraph [27]. We explain why this approach is not optimal for detecting vote gaming attacks; we then present a new clustering approach using *canopy-based clustering*.

4.1 Problem: Clustering Voting Behavior

Figure 5 shows how we can represent a sample voting graph as the input document-feature matrix M for a clustering algorithm. Let U be the set of users who voted and

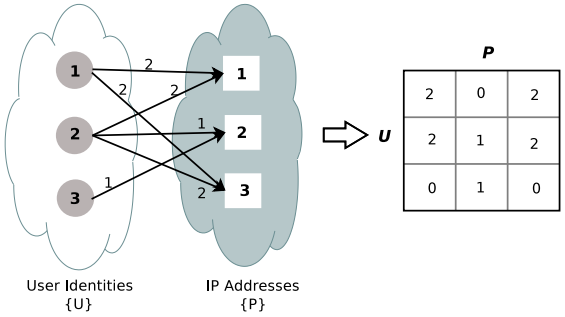


Figure 5: Representing the NS voting graph as an adjacency matrix. Labels on edges represent the number of times a user votes on an IP.

P be the set of IPs they voted on. $M \subseteq U \times P$, and each $M(i, j)$ denotes the number of votes given by user i to an email sent from IP j. The matrix M consists of all users who have voted and all IPs that have received a non-spam vote. Our goal is to extract groups of fraudulent user identities from M with few false-positives.

Large email providers have tens of millions of active users per month, and the number of voted-on IPs is on the order of millions. We wish to identify the user IDs that behave similarly by clustering in this high-dimensional space. Our setting differs from conventional clustering setups [12] in the following ways:

1. **Lack of cluster structure.** Unlike the usual settings in which clustering is performed, there are no clear clusters in our data. In fact, with a normal set of users, two users will rarely receive emails from the same IP, and even more rarely will they cast the same vote on the same IP. Thus, any form of tightly connected clusters in our data is a signal of anomaly—as we shall see later, we instead end up with a large number of clusters at various scales.
2. **Sparsity.** On average, users cast *less than one* non-spam vote during the entire month, although we also observe a significant number of users with large numbers of non-spam votes.
3. **Data scale.** Our data has many users and IPs. While many traditional clustering algorithms are quadratic time, our data's scale requires linear-time or near-linear-time algorithms.
4. **Adversarial nature.** The data is generated adversarially. The spammers can succeed only if they can remain undetected by the anti-spam filters. This means that we rarely get spammers casting a large number of non-spam votes from the same ID. Instead, campaigns to vote "Not Spam" are distributed over a large number of user IDs.

These features make the choice of clustering algorithm and distance metric critical. As a simple example, clustering based on distance metrics such as the Euclidean metric will erroneously show high similarity between IDs which have few IPs in common as long as the common IPs have

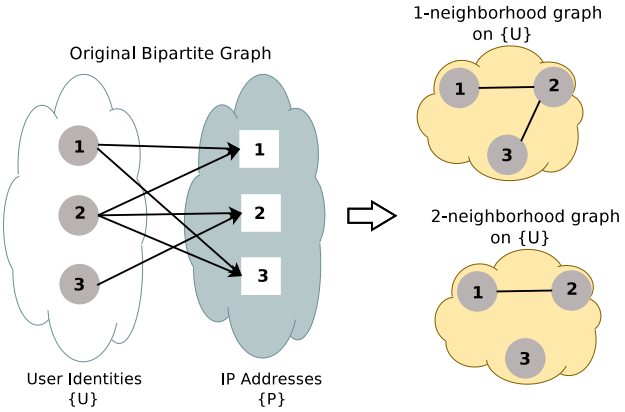


Figure 6: k-neighborhood representation of $\{U\}$ from Figure 5.

high weight.[1] Consequently, we need to develop clustering strategies specifically for our problem setting.

4.2 Baseline: Graph-based Clustering

As a baseline for comparison, we apply a graph-based clustering method that is similar to the technique introduced by Kumar *et al.* [16] and later applied by BotGraph [27]. We choose this algorithm to enable direct comparison of methods used in previous work, and with our second approach, canopy-based clustering. Kumar *et al.* [16] proposed the k-neighborhood plot as a way to study the similarities between entities using Web data. Given a bipartite graph $G = (A, B, E)$, Kumar *et al.* define the *k-NC graph* H corresponding to G as follows: H is defined over the vertex set A; we include edge (u, u') in H if there exist k distinct nodes $\{v_1 \dots v_k\} \subseteq B$ such that for each i, both (u, v_i) and (u', v_i) are in G. Figure 6 illustrates the construction of a k-neighborhood graph from a bipartite graph. Zhao *et al.* use the same construct in BotGraph to discover botnets by working with the bipartite graph of users versus the Autonomous System (AS) numbers of the IPs from which users log in [27]. We make one improvement to the clustering approach in BotGraph: rather than mapping user accounts to AS numbers, we map them to IP addresses, since mapping user accounts to AS numbers hides the fact that a user account is accessed from multiple locations.

Efficiently finding a value for k. Two users voting NS on the same k sender IPs is indicative of suspicious coordinated behavior. The success of this approach depends on efficiently finding a value of k that identifies a significant number of attackers with no false positives. A low value for k may retain some legitimate users in components that mostly have bots. On the other hand, a high value for k produces components whose voting behaviors are highly coordinated, although the sizes of the components—and hence the number of bots identified—decrease.

A simple way to construct the k-NC graph for any fixed value of k first creates the weighted graph G' with vertex set U where for each (u, u') the weight $w(u, u')$ equals the number of common neighbors of u and u' in G. Then, we can create the threshold using the value of k that we desire and apply standard component finding algorithms. This takes time $O(\min(|U|^2, i_{\max}|E|))$ where $i_{\max}$ is the maximum number of users who vote on an IP. This approach is infeasible when the size of $|U|$ is on the order of tens of millions, and $i_{\max}$ is typically of the order of thousands as well. For a fixed value of k, Kumar *et al.* [16] show how to compute the k-NC graph in time $O(n^{2-1/k})$ where $n = |U|$, which is significant gain for small k. Our setting, however, requires a larger k to ensure we do not create edges between normal users and bot accounts so this algorithm is impractical in our setting. Furthermore, as with BotGraph [27], we need to run the component-finding algorithm at various values of k to find the right threshold.

To create components at various thresholds, we have developed a new technique using dynamic graph algorithms for maintaining components under edge additions and deletions. Although it is difficult to maintain components under edge deletions, it is easier to do so under edge additions. Thus, we start with a maximum value $k_{\max}$, find components with threshold $k = k_{\max}$, and then decrease k by 1. At each step that we decrement k, the graph gains a new set of edges, and these could change the component structure by joining some previously disconnected components. Updating the component list efficiently only requires maintaining a union-find data structure, and the whole process takes total time $O(k_{\max}(|U| + |E|.\alpha(E, U)))$, where $\alpha(E, U)$ is the inverse Ackermann function, an extremely slow-growing function which is a small value—less than 5—for almost all practical values of $|E|$ and $|U|$.

Graph-based Clustering Produces False Positives. The most significant shortcoming of graph-based clustering such as BotGraph [27] for detecting bot-controlled accounts is its false positive rate, which are typically unacceptable for email. Intuitively, graph-based clustering disconnects edges lower than a certain weight and labels *all* nodes in a large connected component as bots; it does not pay attention to the absolute degree of a node in a connected component *when compared with other nodes in the component*. This behavior produces false positives.

Figure 7 illustrates why graph-based clustering may produce false positives. The nodes (*i.e.*, user accounts) shown outside the cloud are legitimate, but the nodes inside the cloud are controlled by bots. All the legitimate accounts share two IP addresses between each other (*e.g.*, perhaps due to a company proxy server that cycles between two

[1] Consider two vectors $\mathbf{A} = [1, 1, 1, 10]$, $\mathbf{B} = [0, 0, 0, 10]$, and $\mathbf{C} = [1, 1, 1, 3]$. The distance between $\mathbf{A}$ and $\mathbf{B}$ is $d_{\text{Euclidean}}(\mathbf{A}, \mathbf{B}) = 1.73$, although $\mathbf{A}$ and $\mathbf{B}$ have only one feature in common. The distance $d_{\text{Euclidean}}(\mathbf{A}, \mathbf{C}) = 7.0$, *i.e.*, *greater than* $d_{\text{Euclidean}}(\mathbf{A}, \mathbf{B})$, even though $\mathbf{A}$ and $\mathbf{C}$ vote on the same set of IPs. The high-valued feature influences the Euclidean metric more than, for example, the Jaccard metric.

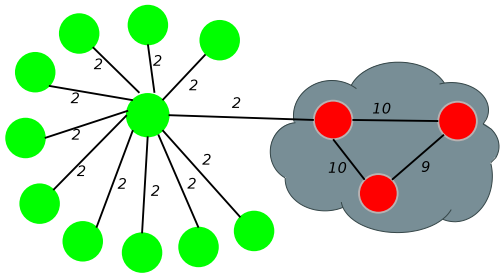


Figure 7: Shortcoming of graph-based clustering: one false-positive edge can connect a bot component (shown within the cloud) to a number of unrelated, almost-disconnected legitimate users (outside the cloud). Edge labels are the edge-weights. Here, the threshold $k = 2$.

public IP addresses), as shown by the edges with weight two. Unfortunately, one legitimate user has also logged in from two IP addresses that have bot programs running on them. This scenario could be a false positive—for example, the legitimate user's IP address could have been recycled with DHCP to a botted machine—or it could have occurred accidentally, because the legitimate user has a bot program on his computer while he continues to use it. In either case, this legitimate user acts as a "bridge" that connects a component of true voting bots, and a number of legitimate users that would otherwise have been disconnected. A clustering algorithm based on pairwise similarity comparisons is unlikely to make this mistake because it would compare all-pairs similarity, and discover that the true bots have a much higher similarity to each other than other pairs. Although this particular false positive could have been avoided by increasing the value of the threshold k to 3, the BotGraph algorithm would stop the component finding process at $k = 2$, because the component sizes between successive steps differs by an order of magnitude: the component of 14 nodes breaks to a largest component of 3 nodes if k is increased to 3.

4.3 Our Approach: Canopy-based Clustering

To reduce false positives and cope with high dimensionality, we adapt a two-stage clustering technique by McCallum *et al.* called *canopy clustering* [17]. Canopy clustering is a divide-and-conquer approach for clustering high-dimensional data sets. Canopy clustering is more practical than graph-based clustering for detecting vote-gaming attacks, because it produces fewer false positives and is more scalable. The algorithm proceeds in two stages:

Step 1: Canopy Formation. First, we partition the raw data into overlapping subsets called canopies, using an inexpensive similarity metric and very few similarity comparisons. We construct canopies such that all elements in a cluster in the output of a traditional clustering algorithm will be within the same canopy. Thus, the second stage of canopy clustering need only conduct more rigorous similarity comparisons for elements that are within the same canopy. Provided that the number of elements in the largest canopies are much smaller than in the raw data, this method typically reduces the number of expensive similarity measurements by many orders of magnitude.

The choice of metric used to create the initial partition of the raw data into canopies is important: a good metric is inexpensive (*i.e.*, does not involve operations such as division or multiplication), and minimizes the size of the largest canopy. Following McCallum *et al.*'s suggestion of using the number of common features between elements as an inexpensive metric, we use the *number of common IPs voted on by two users* as our canopy metric. We explain this metric in Section 6, and how its parameter settings affect detection and false positive rates in Section 7.

Step 2: Conventional Clustering. The output of the first step are canopies of tractable sizes, such that we can directly perform clustering on each canopy. For this stage, we use a well-known hierarchical clustering scheme, greedy agglomerative clustering (GAC), using α-*Jaccard* similarity[2] as the metric. We choose GAC using the Jaccard metric because it is appropriate for clustering user IDs where the similarity metric should take into account the fraction of shared IPs. In Section 6, we introduce an approximation of this method that works in a cluster computing infrastructure such as Hadoop. We also discuss how to parallelize this clustering using techniques from locality sensitive hashing [6].

GAC is an iterative method, where initially, each element in the data set is in a cluster of its own. At each iteration, we find the similarity between every pair of clusters using the Jaccard metric, and merge the two clusters that are the most similar to each other, provided this similarity is greater than a threshold, α. We compute the Jaccard metric between two clusters using the mean distance between elements in the cluster. If $\mathcal{C}_1$ and $\mathcal{C}_2$ are two clusters of elements, the mean distance is

$$d_{\text{mean}}(\mathcal{C}_1, \mathcal{C}_2) = \frac{1}{|\mathcal{C}_1|\,|\mathcal{C}_2|} \sum_{x \in \mathcal{A}} \sum_{y \in \mathcal{B}} d_{\text{Jaccard}}(x, y)$$

Iteration stops when either (1) only a single cluster remains, or (2) the similarity between the two most-similar clusters is less than α. Because canopies are overlapping, an element may be clustered into multiple clusters. To resolve this issue, after we perform GAC on each canopy independently, we assign any element that is in multiple clusters solely to the largest cluster; we find that this choice does not incur false positives because most large clusters are likely comprised of bot accounts.

5. Evaluation

[2]Let x and y be two user identities, with X and Y representing the sets of IP addresses on which they voted "not spam". x and y will be clustered together only if

$$\frac{|X \cap Y|}{|X \cup Y|} \geq \alpha$$

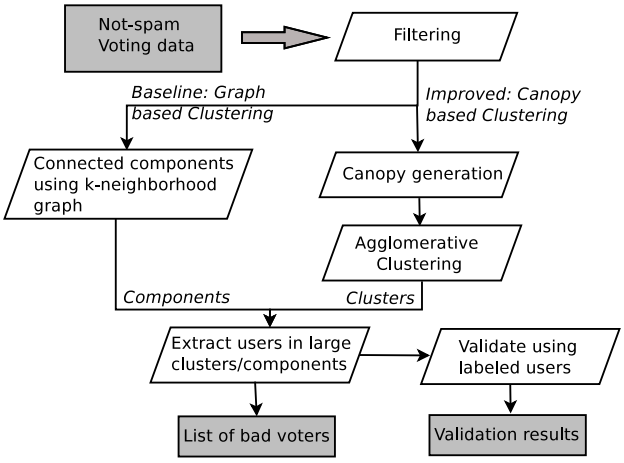


Figure 8: Workflow for finding and validating fraudulent voters from unlabeled voting data.

We evaluate the accuracy and precision of the clustering algorithms for detecting vote gaming attacks. Section 5.1 describes our dataset; Section 5.2 describes the metrics used to evaluate the quality of the clustering algorithms, and presents the basic performance of each algorithm for identifying vote gaming attacks. Figure 8 explains the workflow of our evaluation and validation technique.

Main Result. Although both canopy-based greedy agglomerative clustering (GAC) and graph-based clustering both can detect vote gaming attacks, GAC has a higher detection rate (10% vs. 3%) and a lower false positive rate (0.17% vs. 1.09%). (Section 5.2, Table 2)

5.1 Data

Our dataset consists of the logs of votes cast by the users of a large Web mail service provider on mail that they receive, extending for four months from July–October 2009. Each line corresponds to one vote; the fields included are: (1) the ID of the user who cast the vote, (2) the IP address of the sender of the email on which the vote was cast (the *"voted-on" IP*), and (3) the type of vote—"S" for spam and "NS" for not spam. Section 6 describes the filtering stage of our workflow.

To validate whether the clusters of voters we obtain contain bots, we use independent labels of known fraudulent voters. To evaluate the percentage of false positives, we use a list of users known to engage in reputable behavior; this list contains users who have long-standing accounts with the provider, or users who have purchased items from e-commerce sites also owned by the provider's parent company. Because the set of labeled users was collated independently by the anti-spam team at the large Web mail provider, only a subset of these labeled accounts intersect with our 4-month dataset of NS votes.

Table 1 summarizes the voting dataset and its intersection with user labels. We have observed empirically that, although some NS votes are legitimate (*e.g.*, there are cases

Period	4 months (Jul.–Oct. 2009)
Total Voting Users	35 million
↪ *Total only-NS voters*	39.8%
↪ *Users labeled "good"*	3.71%
↪ *only-NS voters*	1.76%
↪ *Users labeled "bad"*	6.91%
↪ *only-NS voters*	6.82%
Total Spam votes	357 million
Total Not-spam votes	82 million
↪ *By only-NS voters*	63%
Voted-on IPs	5.1 million
↪ *Voted-on as NS*	1.7 million

Table 1: Description of voting dataset.

Method	*Median size*	*Detection*	*FP rate*
Canopy Clustering	109	**10.24%**	**0.17%**
Graph-based	32	3.51%	1.09%

Table 2: Comparison of Greedy Agglomerative Clustering (GAC) and Graph-based clustering that shows the median cluster (or component) size, and the associated detection and false positive rates.

where a legitimate email contained keywords that triggered a content filter for spam), the majority of NS votes are performed by bots to delay the identification of spam sent by other bots: 63% of NS votes are cast by users who only cast NS votes. Although we derive data labels using independent verification methods (*e.g.*, manual inspection, suspicious account activity), these labels can often only be attributed to the users *after* they have performed a significant amount of malicious activity and have been deactivated. Our goal is to identify as many *undiscovered* fraudulent voters as possible, so we use accounts that are labeled after the time period during which we evaluate our clustering methods.

5.2 Detection and False Positive Rates

Our aim is to identify large groups of bots without incurring many false positives. Thus, we compare the two techniques in terms of two metrics: (1) *detection rate*, *i.e.*, the fraction of users labeled "bad" (*i.e.*, fraudulent voters) who are classified into clusters larger than the x^{th} percentile cluster size (x being variable), and (2) *false positives (FPs)*, which we quantify as the ratio of good users in clusters larger than the x^{th} percentile cluster size to all good users, for various values of x. Table 2 presents these statistics for the median (*i.e.*, $x = 0.5$) cluster size, and Figure 9 shows the detection and FP rates for various percentile values (x). Neither GAC nor graph-based clustering vary much in terms of detection or false positive rates with respect to x; thus, even a small-sized cluster is likely to contain mostly bots. Graph-based clustering results use $k = 5$, and canopy-based GAC uses a Jaccard similarity threshold of 0.85. Section 7 explains our parameter

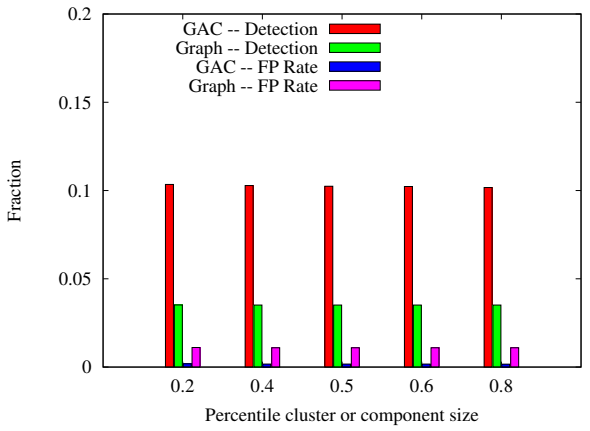


Figure 9: Performance of GAC and Graph-based clustering for various percentiles of cluster/component sizes. The x-axis shows the percentile cluster size above which all clusters are considered to contain only bots. The y-axis shows the detection and false positive rates.

choices for both algorithms in detail.

Canopy-based GAC outperforms graph-based clustering in terms of both the detection rate and the false positive rate. GAC performs better because, as explained in Figure 7, it is more precise than graph-based clustering. In graph-based clustering, a large connected component at some k may contain two or more sub-components which are connected only by an edge of weight exactly k. Even if the users in one sub-component do not vote on the same IPs as users in the other, they will be categorized into one large component, potentially increasing false positives if some of these users are legitimate. GAC performs all-pairs similarity comparison between users, which results in clusters where all users are similar to one another.

One of the top three large Web mail providers is using our detection technique in production. Although a 10% detection rate may seem low, even single-percentage-point gains are significant for a for large-scale Web mail providers, given the high volumes of spam seen by Web mail providers. Any increase in detection rates can help these providers make more accurate decisions about which email connection attempts to reject early, and which mail can be more quickly and efficiently classified as spam (*e.g.*, without inspecting the message's contents); indeed, clustering is being applied in practice at the large Web mail provider to detect fraudulent voters. Our techniques also identified fraudulent voters more quickly than other methods: many of the bots we discovered were identified by the anti-spam team as bots only well *after* our dataset was collected. We also note that the actual detection rate may be higher that 10% in practice, because at least some of the users labeled "bad" may have had the bulk of their malicious activity before or after the time period of our dataset.

6. Scalable Distributed Implementation

We describe scalable implementations of the distributed graph-based clustering (Section 6.2) and canopy-based clustering (Section 6.3). We evaluate the performance of the two methods in Section 6.4.

Main Result. Both implementations run on our 4-month dataset in only a few hours, making it practical to run on a sliding window that includes new voting data. GAC is slower than graph-based clustering due to the overhead of all-pairs comparisons (Section 6.4, Table 3).

6.1 Overview

At the scale of large Web mail providers, raw voting data totals tens of millions of unique identities that map to millions of IP addresses. At this scale, analyzing data on a single machine is often infeasible. Many large organizations such as Yahoo!, Google, and Microsoft use distributed computing to analyze Web-scale datasets, by storing the data on distributed filesystems and using methods such as MapReduce [4] to process them.

MapReduce is appropriate for tasks that are inherently parallelizable, such as searching and sorting, but solving clustering tasks using MapReduce poses a number of challenges. First, because individual rows of the matrix M may be split across different mappers and reducers, MapReduce clustering algorithms often take many iterations to converge to a high-quality clustering. Second, between each iteration of clustering, there could be a large amount of inter-node communication in the distributed filesystem as potentially similar rows of M are sent to the same mapper/reducer. Finally, the intermediate output containing the results of comparing every pair of rows may sometimes be much larger than the raw dataset. Although some clustering algorithms, such as k-means [12], are parallelizable, they are ill-suited for our problem.[3] Unfortunately, our clustering algorithms expect a shared-memory architecture and are not inherently parallelizable. Below, we present efficient approaches to implementing both graph-based clustering and canopy-based clustering using MapReduce that trade off accuracy for efficiency.

6.2 Distributed Graph-based Clustering

Step 1: Creating an approximate user-user graph using MapReduce. In a distributed infrastructure, computing the k-neighborhood graph is challenging due to the amount of intermediate output it generates. Suppose the original bipartite graph is stored in the following format:

```
<user ID> <list of (IP, NS votes) pairs>
```

Because this file is split across many machines, the straightforward approach to construct the k-neighborhood

[3] k-means, although widely applied, has flaws: (1) every point in the data is forced into a cluster, which may affect the cluster quality if points are outliers; (2) as mentioned before, the euclidean distance metric is both expensive to compute, and gives weight to the larger-valued features than the number of common features; (3) the number of clusters, k, may not be easy to determine beforehand.

graph uses two MapReduce iterations. The first iteration's Map phase outputs the *inverse* edge file where each line has an IP address as the key and a user ID that voted on it as the value. The Reduce phase will then collect all lines with the same key and output all pairs of users who have the same key. The second iteration counts the number of time a specific user-user pair has been written out, which yields the number of IPs shared between the two users—the edge weight in the user-user graph. The main bottleneck in this process is the size of intermediate output between the two iterations: for example, an IP that has been voted on by 1000 users will produce $\binom{1000}{2}$ pairs of user-user entries, and when repeated for many high-degree IPs can overflow even the terabytes of space on a distributed filesystem.[4]

We apply approximations to filter the number of intermediate user-user edges that must be output. We first filter users who have voted on very few IPs. Next, because we are interested only in users who fall into large components at reasonably high values of k, we suppress user-user edges where the two users are unlikely to have many IPs in common. To do so, we hash the IPs that are voted on by a user into a fixed-size bit-vector, essentially a variant of a count-min sketch [3]. Before outputting a user-user edge, we compare the overlap between the two users' bit vectors and proceed only if the overlap is greater than a certain threshold (which we set to lower than k_{max} because hashing different IPs to a fixed-size bit vector could create collisions). Similarly, when outputting all the user-user pairs for a certain IP that has a large number—say p users, voting for it—instead of outputting all $\binom{p}{2}$ pairs, we select a random subset of size αp and output them only. It is possible to tune the value of α with respect to the threshold k desired to ensure that we do not break apart large connected components in the resulting user-user graph.

Step 2: Finding connected components on the user-user graph. Finding connected components using MapReduce needs at least $O(d)$ iterations, where d is the diameter of the graph (*i.e.*, maximum length shortest-path between any two vertices). In this approach, the input is the edge file of the user-user graph and a vertex-component mapping that maps each vertex to its "component ID", initially set to the ID of the vertex itself. In each iteration, a mapper processes each edge $e(u, v)$ in the edge file and outputs two lines $< u, i >$ and $< v, i >$ where i is the minimum component ID of vertices u and v. This output becomes the new vertex-component mapping. The process is repeated until no vertex changes its component ID. In the case that the set of vertices fits into memory, we can employ the algorithms outlined in [14] to actually find components in a constant number of passes.

[4]Zhao *et al.* also face this problem, but alleviate it using DryadLINQ [26] that offers a "merge" capability to reduce intermediate output size; we use the more widely-used MapReduce platform.

6.3 Distributed Canopy-based Clustering

Step 1: Creating Canopies. Although our dataset comprises tens of millions of user accounts that cast votes on millions of IP addresses, the graph is sparsely connected. Because the adjacency matrix M is sparse, we choose a sparse matrix representation, M', where each row $M'(i)$ is a set of t tuples, where t is the number of IP addresses that ID i has cast votes on. M' is constructed such that, if an entry $(j, k) \in M'(i)$, then $M(i, j) = k$.

We create canopies using an inexpensive similarity metric and use the number of common IP addresses to measure similarity between two rows of M. Adapting the method by McCallum *et al.* [17], we first create an inverted index N that maps IP addresses to the set of users who vote on them. To create a new canopy, we pick a random row i from M and add it to the canopy as the first row. For each non-zero column j in $M(i)$, we find the other rows in M that also vote on IP j using the row $N(j)$. Using the inverted index allows us to ignore all rows of M and only compare with the rows from $N(j)$. We use upper and lower thresholds—T_{high} and T_{low} ($T_{high} > T_{low}$)—to measure similarity: if the similarity of a given row in M to $M(i)$ is greater than T_{high}, we *remove* the row from M and add it to the canopy. If the similarity is less than T_{high} but greater than T_{low}, we add the row to the canopy but *do not remove it from* M. This procedure explains why canopies can be overlapping: if a row is removed from M, it will not be considered for inclusion in any more canopies. In our implementation, we set T_{high} to 7 and T_{low} to 5; *i.e.*, a row is added to a canopy removed from M if it has at least 5 rows in common with the first row in the canopy, and it is also removed from M if it has at least 7 rows in common with the first row. We explain how we obtain these numbers in Section 7.2.

Step 2: Greedy Agglomerative Clustering. After computing canopies, we read each canopy and cluster only the rows in that canopy. To reduce the workload, we skip canopies smaller than 10 rows and canopies where the first has fewer than two non-zero columns. We use the average-linkage clustering metric to decide the similarity between rows in a canopy. If a row is a member of multiple canopies, we include that row in the clustering input for all canopies. In the final output, we include such rows as members of the largest cluster among different canopies. In a distributed setting such as MapReduce, accurate canopy clustering can be quicker than an accurate graph-based component-finding algorithm: provided the largest canopy can be clustered by a single node, agglomerative clustering of canopies can be done entirely in parallel in one step, without involving the inter-node overhead or the $O(d)$ iterations of graph-based component-finding.

Although for our dataset, the naïve implementation that compares every pair of clusters within a canopy before merging the two most similar clusters is sufficient, locality

Method	*WC time*	*Sys. time*	*Max RSS*
Graph-based	86.7 min	6.8 sec	5944 MB
↪ Hadoop	14 min	N.A.	N.A
GAC	5.5 hrs	2.3 min	8221 MB
↪ Canopy formation	30.1 min	2.7 sec	3109 MB

Table 3: Speed and memory consumption of our GAC and graph-based clustering implementations. Times for graph-based clustering include the multiple iterations of finding connected components, from $k = 20$ to $k = 7$. We could not measure the system time or RSS for our Hadoop implementation.

sensitive hashing (LSH) makes this step faster [6]. With LSH, we can create a hash-function on the vectors of the IPs that two users vote on, such that with high probability, two users with Jaccard coefficient above α are going to fall in the same hash-bucket. The threshold α and the probability desired will control the parameters of the hash-function. We compare pairwise all user IDs that fall within each bucket, and choose the most similar pair of IDs to merge as one cluster. Once we form a new cluster by merging two user IDs, we can repeat the process using the vector representation of the new cluster using the same hash function. This process ensures that at any step, we find the nearest neighbors with high probability.

6.4 Comparison: Clustering Speed

To evaluate the speed of each approach, we implemented and tested each approach on an unloaded 8-core Intel Xeon 2Ghz machine (4MB L2 cache) with 36GB of main memory running Linux 2.6.32. Both implementations were single-threaded. In addition, we tested our approximate graph-based clustering implementation on a distributed cluster using the Hadoop MapReduce framework. The input was the edge file for the bipartite graph that maps users to the IPs that they vote on.

Table 3 presents the times taken and maximum resident set size for each method. Although GAC performs better than graph-based clustering, GAC takes longer and consumes more CPU time because of many all-pairs similarity computations between users in a canopy. The GAC phase does not require more memory consumption than the canopy formation; the extra memory usage is likely due to the memoization used to speed up our implementation. Canopy-based clustering can be easily parallellized, so with a multi-threaded application, we expect to gain a speedup proportional to the number of cores. Table 3 also shows the large improvement in running time for our approximate graph-based clustering algorithm on a grid infrastructure such as Hadoop [11]. Although we could not implement canopy clustering on the same infrastructure, we expect a significant speedup for that method as well.

7. Sensitivity Analysis

In this section, we analyze the sensitivity of the detection and false positive rates for the algorithms evaluated in Section 5.

Main Result. The effectiveness of both techniques depends on parameter settings. Because graph-based clustering has a single parameter (the neighborhood density, k), its cluster sizes are more sensitive to the setting of k (Section 7, Figure 10).

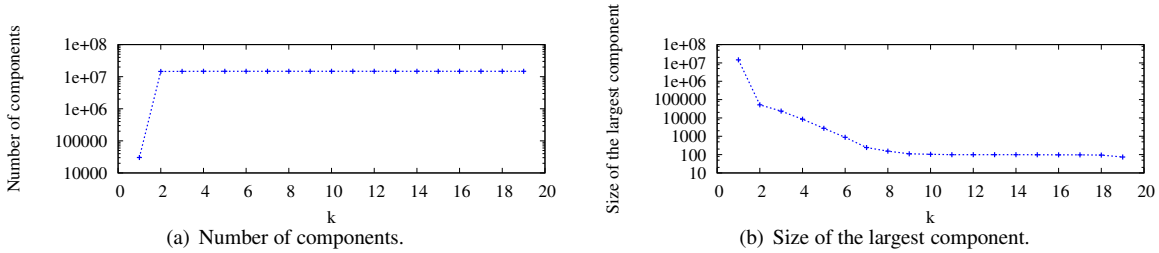
7.1 Graph-Based Clustering

Our goal is to find a value of k that yields clusters that are as large as possible with few false positives. This task is challenging: selecting the smallest value of k where the largest component fragments might yield $k = 2$. However, $k = 2$ may not yield large components containing only bots with no false positives, because to be in a connected component at $k = 2$, a legitimate user only needs to vote "not spam" on two IPs that a voting bot also votes on as "not spam"; this event may occur either if a user votes "not spam" by accident or because the voted-on IPs were re-assigned during our data timeframe due to DHCP reassignment. Thus, instead of choosing the stopping value of k only using the decrease in size of the largest component, we stop when a large fraction of labeled users in the largest components are known dishonest voters.

Figure 10 shows the number of components and the size of the largest component as k increases from 1 to 19. As Figure 10(a) shows that at $k = 1$, almost all nodes are in a giant component that includes nearly all nodes in the user-user graph, but just by increasing k to 2, the giant component fragments from over 14.6 million nodes to just 52,006 nodes, and the number of components increases from 30,225 to over 14.5 million. Figure 10(b) highlights the decrease in the size of the largest component, echoing the structure of the Web pages-vs.-ads bipartite graph in Kumar *et al.*'s work [16].[5]

Even for low values of k, the largest component consists mostly of "bad" users. Figure 11(a) shows how the fraction of users labeled as fraudulent in the largest component varies as a fraction of all labeled users, for various values of k. Even at $k = 2$, the largest component has no users labeled "good" (*i.e.*, no false positives). This characteristic holds as k increases: there are no false positive "good" users in the largest component at any value of k greater than two. However, the *minimum component size* above which there are no false positives is dependent on k. We examine the size of the largest component and the fraction of dishonest voters in each component (among labeled users). Figure 11(b) shows the number of false positives in each component, rank-ordered by the size of the

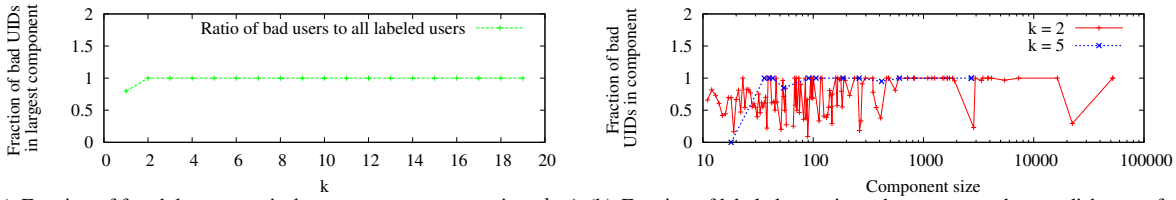
[5]This work illustrates the similarity of Web pages based on the number of advertisements they share; they found that sharing even 5 advertisements did not say much about the connection between Web pages, but six or more shared advertisements implied a stronger notion of similarity. Similarly, we find that two users in the same component at $k = 2$ or $k = 3$ are not necessarily similar but connections at a slightly higher value of $k = 6$ or $k = 7$ implies high similarity.



(a) Number of components.

(b) Size of the largest component.

Figure 10: Variation of the number of components and the size of the largest component as the value of k increases from 1 through 20. The number of components do not increase much past $k = 2$, but the size of the largest component decreases exponentially from $k = 2$ to $k = 8$. We pick a value of k that gives a good tradeoff between the component size and number of components ($k = 5$).



(a) Fraction of fraudulent voters in *largest* component at various k. A value of 1 indicates zero false positives.

(b) Fraction of labeled users in each component that are dishonest, for different component sizes and at different values of k. Low values on the y-axis indicate a higher false positive rate.

Figure 11: (a) Fraction of "bad" users in the largest component as k is varied; and (b) the fraction of "bad" users as component size varies for two specific values of k. The largest component only contains users labeled "bad" above $k = 2$, but there is higher variability in the false positive rate for smaller-sized components at $k = 2$ than at $k = 5$.

component, for $k = 2$ and $k = 5$. Smaller components for small values of k often include many "good" users; at $k = 2$, even the second-largest component contains more than half good users. As we increase k to 5, the good-user portion of the large component fragments, resulting in smaller components with even fewer false positives, which is why we picked this threshold for our evaluation.

7.2 Canopy-Based Clustering

Choosing thresholds for canopy formation. The first step in canopy-based agglomerative clustering is canopy formation, which is parameterized by the thresholds T_{high} and T_{low} (Section 6.3). These thresholds control the extent to which the data is partitioned and the extent to which canopies overlap with one another. Because we apply canopy clustering to reduce the size of our input dataset, we must pick values of T_{high} and T_{low} such that: (1) the average size of canopies are reduced, (2) the overlap between canopies is reduced, and (3) the total number of canopies are reduced. Low values of T_{high} reduce overlap, and high values of T_{low} decrease the size of canopies. However, if both T_{high} and T_{low} are too large, all but highly similar rows will be in non-singleton canopies.

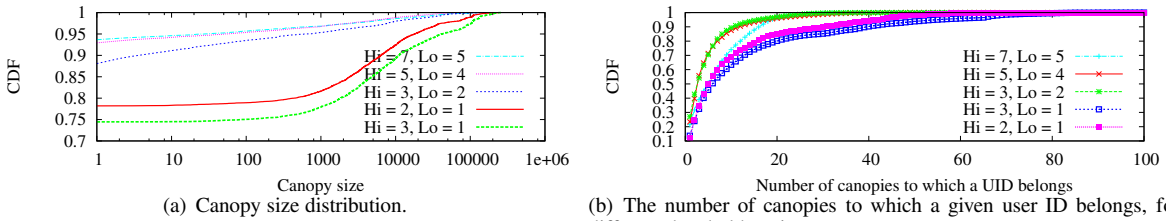
Figure 12(a) plots the size distribution of canopies on varying T_{high} and T_{low}, and Figure 12(b) plots the CDF of the user IDs which are mapped onto multiple canopies. These figures show that setting $T_{high} = 7$ and $T_{low} = 5$ partitions the users into distinct canopies into a few small canopies with minimal overlap.

Sim. Threshold	*Detection Rate*	*FP rate*
0.90	8.74%	0.14%
0.87	9.01%	0.15%
0.85	**10.24%**	**0.172%**
0.82	15.52%	0.217%
0.78	17.52%	0.244%
0.76	19.24%	0.26%
0.74	21.70%	0.328%
0.72	23.29%	0.499%

Table 4: Sensitivity of the detection and false positive rates to the choice of the similarity threshold. We chose 0.85 (highlighted).

Choosing a threshold for the Jaccard Metric. We cluster each canopy using average-linkage similarity (Section 4.3). For each canopy, GAC iteratively performs all-pairs similarity computation and merges the most similar clusters if their Jaccard similarity exceeds a similarity threshold. Table 4 shows how the detection rate and false positive rates change for other settings of the similarity threshold. A similarity threshold of 0.85 yields a high detection rate and a low false positive rate.

Figure 13(a) shows the size distribution of the clusters we obtained. More than 99% of clusters are singletons (*i.e.*, likely legitimate users). Figure 13(b) shows the distribution of dishonest voters for various cluster sizes, presented as a fraction of labeled users in the cluster. All large clusters except for one have almost no false positives. The



(a) Canopy size distribution.

(b) The number of canopies to which a given user ID belongs, for different threshold settings.

Figure 12: Canopy characteristics for various upper and lower thresholds, T_{high} and T_{low}.

exception—a cluster of 12,890 users—has 517 users labeled "good" 2,776 users labeled "bad". Considering that all of these false positives fall into a single cluster, these users are likely compromised users that were mislabeled.

8. Related Work

Yahoo! Mail, Hotmail, and Gmail now each have hundreds of millions of users. Because Web mail providers started adopting and inventing schemes to prevent or limit botnet-generated spam such as Sender Policy Framework (SPF) [10] and DomainKeys Identified mail (DKIM) [5], the amount of messages verifiable via SPF or DKIM also increased as users migrated to Web mail. Thus, emails sent by bots with fake or nonexistent verification parameters (*e.g.*, a bot masquerading as a '@yahoo.com' sender) became simple to identify and drop early in the pipeline. Unfortunately, spammers can defeat DKIM or SPF by sending mail through compromised Web mail accounts. The numbers of compromised Web mail accounts, and the amount of spam sent through Web mail providers have continued to increase: Malware was found as early as 2007 that targeted Web mail in order to automatically create accounts [24]. Microsoft reports that it discovered at least 26 million compromised accounts in Hotmail in 2008 [27].

Researchers have used clustering to identify bots using network-level features from spam and legitimate email [22], and long-lived *network-aware* clusters formed by spammer IP address prefixes to mitigate spam [23]. Qian *et al.* improve the network-aware cluster approach with a hybrid clustering approach that includes both spammer IP address and their DNS information [21]. Kumar *et al.*'s study [16] presents the k-neighborhood graph model we use in this paper, and its application to domains such as relationships in social networks, collaborative blogging or bookmarking sites, and Web page similarity. The most similar work to ours is BotGraph, which identifies compromised accounts in Hotmail using the graph-based component-finding algorithm similar to the one described in this paper [27]; our paper shows that graph-based component finding has shortcomings for detection of vote gaming attacks, since it generates false positives.

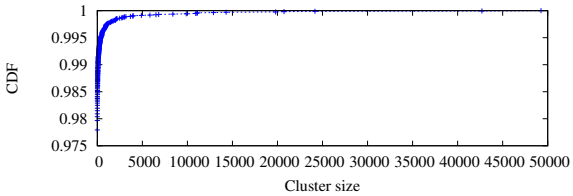
Clustering to find bots has also been applied in areas other than email spam. Metwally *et al.* implemented systems [18, 19] to identify fraudulent publishers in the domain of web advertising. Their work attempts to efficiently estimate similarity in the sets of IP addresses that click on advertisements hosted a pair of publishers, and to cluster publishers that have high similarity with each other—which likely indicate fraudulent publishers. In the area of scam hosting, Konte *et al.* show that many different scammer domain names share the same hosting infrastructure [15]. Perdisci *et al.* have extended this work in using clustering to identify scam-hosting domain names that use DNS fast-flux to cycle between IP addresses [20].

9. Discussion

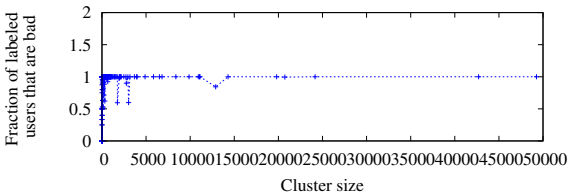
We present the results of identifying voting bots using a complementary dataset, where we map user accounts to the login IP address of the user who cast a not-spam vote (*i.e.*, the IP address of the host from which the user logged in to the Web mail service). We also discuss potential limitations of our approach and our evaluation.

Clustering Using Login IPs. We have an additional dataset from May–June 2009 that has the login IP address of the user (recall that the dataset in Section 5.1 has the IP address of the sender of the email on which the user cast a vote). We expect that the IP addresses from which a dishonest NS-voting user logs in should also follow the model of Section 3. Table 5 summarizes the results of graph-based clustering applied to the graph that maps user IDs to these login-IPs. Indeed, a large number of IP addresses shared a given bot account (specifically, larger on average than the number of IP addresses a bot account votes *on*); hence, a higher neighborhood density of $k = 8$ yields the best results. As expected, most users in the largest components were identified as bot-controlled. Certain components have significant fractions of accounts not yet labeled (*e.g.*, the third-largest component has 55% accounts not yet labeled), which represents significant savings in terms of the number of fraudulent NS votes that can be prevented. Because we only had access to this data for a limited time, we were unable to compare the results of graph-based clustering with canopy-based clustering.

Low Detection Rate. Although our 10.24% detection rate



(a) Size distribution of clusters obtained using GAC



(b) Distribution of dishonest voters in clusters as a fraction of all labeled users.

Figure 13: Analysis of Greedy Agglomerative Clustering: (a) shows that over 99% of clusters are singletons, and (b) shows that in the clustering output at our chosen parameter settings, most clusters over size 2 (with very few exceptions, as explained in text) have only users that are labeled "bad".

Users	IPs	Validated as Voting Bots	NS Votes
102991	56	102991 (100%)	6.11m
69710	32	64629 (92.7%)	5.14m
59077	39	26592 (45%)	2.58m
49045	65	49045 (100%)	4.5m

Table 5: Results of applying graph-based clustering on login IP data, and extracting the largest 4 components. Because this dataset has different characteristics than our primary 4-month dataset, we found that a neighborhood density of $k = 8$ gave the best results.

may appear low, this number amounts to nearly 26,000 fraudulent voters that were previously undetected by other methods, with only 0.17% false positives. As the sensitivity analysis in Table 4 illustrates, if the operators find a slightly higher false positive rate of 0.5% acceptable, they can detect up to 23.29% of the labeled bad users. Another reason for this seemingly low detection rate is that many users labeled "bad" in the set of labeled users may have had the bulk of their NS votes before or after the timeframe of our data set; such users will not have enough NS voting activity to cluster well with other heavy NS voters.

As the false positive rate analysis in Figure 13(b) shows, large clusters have *zero false positives* (with one exception that is likely due to mislabeling). Because these clusters likely consist of only bot accounts, the actual number of bot accounts detected by our technique will be much greater. For example, the largest cluster in Figure 13(b) alone has nearly 50,000 users, all of which are likely bots.

Dataset Limitations. Because the data that we used in our study was not timestamps, we could not analyze datasets on smaller timeframes. However, our analysis using login IPs shows that smaller timescales also work to identify voting bot accounts. Regardless, our approach can be used for day-to-day detection of bots: because both clustering methods complete in a few hours, an operator could run the analysis daily on a sliding historical window of voting data.

Using Voting Clusters for Real-time Detection. From clusters of dishonest voting accounts, one can go back to the original user-IP graph to retrieve the IP addresses shared by users in the cluster. The IPs and user accounts corresponding to large clusters can then be put on a "watch list", and any new users or IPs that map to a watched user or IP can be investigated before they cause much damage. A second avenue for using our approach in real-time filtering is to combine information obtained using clustering to improve other classifiers. Clustering extracts macroscopic patterns from the activity graph of voting. A traditional *supervised* classifier for voting would use features at the level of each user (*e.g.*, the user account's age, it's reputation, etc.) and might miss accounts that can be discovered by clustering. As an example, consider a reputable user account that becomes compromised and used for dishonest voting. The traditional classifier will likely continue to classify the account as "good", but our clustering approach could instead discover that the account falls into large clusters and raise an alert.

10. Conclusion

Web mail providers rely heavily on user votes to identify spam, so preserving the integrity of user voting is crucial. We have studied a new attack on Web mail systems that we call a *vote gaming attack*, whereby spammers use compromised Web mail accounts to thwart Web mail operators' attempts to identify spam based on user votes. Using four months of voting data from a large Web mail provider, we found that vote gaming attacks are prevalent in today's Web mail voting systems. As a first step towards defending against these attacks, we have developed and implemented a clustering-based detection method to identify fraudulent voters. Our method identifies tens of thousands of previously undetectable dishonest voters over the course of several months, while yielding almost no false positives. The techniques presented in this paper are an important step in stemming the tide of this new class of attacks and are already being used in production as part of a large Web mail provider's techniques to detect fraudulent votes. We believe that these techniques may also be applicable to other online Web forums where bots perform vote gaming, such as user-generated content sites or online polls. We intend to explore the applicability of our methods to these other settings as part of our future work.

REFERENCES

[1] Top 100 Digg Users Control 56 percent of Digg's Homepage Content. http://tinyurl.com/5872jb, July 2006.

[2] Moot wins, Time Inc. Loses. http://musicmachinery.com/2009/04/27/moot-wins-time-inc-loses/, 2009.

[3] G. Cormode and S. Muthukrishnan. An improved data stream summary: the count-min sketch and its applications. *LATIN 2004: Theoretical Informatics*, pages 29–38, 2004.

[4] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. 6th USENIX OSDI*, San Francisco, CA, Dec. 2004.

[5] DomainKeys Identified Mail (DKIM). http://www.dkim.org/.

[6] A. Gionis, P. Indyk, and R. Motwani. Similarity search in high dimensions via hashing. In *Proc. 25th VLDB*, pages 518–529, 1999.

[7] 'Gmail Killer' From Facebook on Its Way? http://www.cbsnews.com/8301-501465_162-20022793-501465.html, 2010.

[8] Google's reCAPTCHA busted by new attack. http://www.theregister.co.uk/2009/12/14/google_recaptcha_busted/, 2009.

[9] Gmail spam filtering. http://www.google.com/mail/help/fightspam/spamexplained.html, 2010.

[10] S. Gorling. An overview of the Sender Policy Framework(SPF) as an anti-phishing mechanism. *Internet Research*, 17(2):169–179, 2007.

[11] Hadoop. http://hadoop.apache.org/core/, 2008.

[12] J. Hartigan and M. Wong. Algorithm AS 136: A K-means clustering algorithm. *Applied Statistics*, 28(1):100–108, 1979.

[13] Interview with Yahoo! Mail Employee. http://tinyurl.com/2g38usp, May 2010.

[14] H. Karloff, S. Suri, and S. Vassilvitskii. A model of computation for MapReduce. *Proc. 20th SODA*, 2010.

[15] M. Konte, N. Feamster, and J. Jung. Dynamics of Online Scam Hosting Infrastructure. In *Proceedings of Passive and Active Measurement Conference*, Seoul, Korea, 2009.

[16] R. Kumar, A. Tomkins, and E. Vee. Connectivity structure of bipartite graphs via the knc-plot. In *ACM International Conference on Web Search and Data Mining (WSDM)*, 2008.

[17] A. McCallum, K. Nigam, and L. H. Ungar. Efficient Clustering of High-Dimensional Data Sets with Application to Reference Matching. In *Proceedings of KDD*, 2000.

[18] A. Metwally, D. Agrawal, and A. E. Abbadi. Duplicate Detection in Click Streams. In *Proceedings of WWW 2005 (E-Applications Track)*, Chiba, Japan, May 2005.

[19] A. Metwally, D. Agrawal, and A. E. Abbadi. DETECTIVES: DETEcting Coalition hiT Inflation attacks in adVertising nEtworks Streams. In *Proceedings of WWW 2007 (E-Applications Track)*, Banff, Canada, May 2007.

[20] R. Perdisci, I. Corona, D. Dagon, and W. Lee. Detecting Malicious Flux Service Netwroks through Passive Analysis of Recursive DNS Traces. In *Proceedings of the 25th Annual Computer Security Applications Conference*, Honolulu, HI, Dec. 2009.

[21] Z. Qian, Z. M. Mao, Y. Xie, and F. Yu. On Network-level Clusters for Spam Detection. In *Proceedings of NDSS, 2010*, 2010.

[22] A. Ramachandran, N. Feamster, and S. Vempala. Filtering spam with behavioral blacklisting. In *Proc. 14th ACM Conference on Computer and Communications Security*, Alexandria, VA, Oct. 2007.

[23] S. Venkataraman, S. Sen, O. Spatscheck, P. Haffner, and D. Song. Exploiting Network Structure for Proactive Spam Mitigation . In *Proc. 16th USENIX Security Symposium*, Boston, MA, Aug. 2007.

[24] Webmail-creating Trojan Targets Gmail. http://www.theregister.co.uk/2007/08/15/webmail_trojan_update/, 2007.

[25] Windows Live Hotmail Fact Sheet. http://www.microsoft.com/presspass/presskits/windowslive/docs/WindowsLiveHotmailFS.doc, 2010.

[26] Y. Yu, M. Isard, D. Fetterly, M. Budiu, U. Erlingsson, P. K. Gunda, and J. Currey. DryadLINQ: A system for genera-purpose data-parallel computing using a high-level language. In *Proc. 8th USENIX OSDI*, San Diego, CA, Dec. 2008.

[27] Y. Zhao, Y. Xie, F. Yu, Q. Ke, Y. Yu, Y. Chen, and E. Gillum. BotGraph: Large Scale Spamming Botnet Detection. In *Proc. 6th USENIX NSDI*, Boston, MA, Apr. 2009.