The Latent Community Model for Detecting Sybil Attacks in Social Networks

Zhuhua Cai  
Rice University  
zc7@rice.edu

Christopher Jermaine  
Rice University  
cmj4@rice.edu

Abstract

Collaborative and recommendation-based computer systems are plagued by attackers who create fake or malicious identities to gain more influence in the system—such attacks are often referred to as “Sybil attacks”. We propose a new statistical model and associated learning algorithms for detecting Sybil attacks in a collaborative network, called the latent community (LC) model. The LC model is hierarchical, and groups the nodes in a network into closely linked communities that are linked relatively loosely with the rest of the graph. Since the author of a Sybil attack will typically create many false identities and link them together in an attempt to gain influence in the network, a Sybil attack will often correspond to a learned community in the LC model. Evaluation of the LC model using real-world networks validates the model and shows that it can be superior to competitive algorithms from network security literature for detecting Sybil attacks.

1. INTRODUCTION

In distributed systems that rely on user recommendations and collaborations to determine the importance or influence of users, malicious users try to create multiple identities with the aim of increasing their own influence in the system. This is often called a Sybil attack. Sybil attacks are found in various domains, from security and routing in peer-to-peer networks to collaborative voting and recommendation systems.

There are two complimentary tactics for dealing with Sybil attacks. The first is prevention: building defense mechanisms that make it impossible for attackers to gain access to the network in the first place, usually through identity verification schemes. This paper considers the second tactic: mitigation. In this approach, the network administrator monitors network structure, looking for attacks, so that the attack can be removed from the network [34, 33, 29, 9]. Existing methods for detecting Sybil attacks assume that an attacker begins by creating a set of “bad” nodes, and then builds a network of arbitrary topology among them. A large component that is disconnected from the rest of the network would be an obvious sign of a Sybil attack, so the attacker also attempts to compromise (or to trick) a set of “good” nodes into linking with some of the bad nodes. Since one might assume that it is expensive or difficult for an attacker to link with a good node, the attack can be detected by finding a subnetwork that is connected to the main network via relatively few links—that is, a narrow “choke point”.

In our opinion, the presence of such a choke point is probably a good indicator of an attack, but we are concerned that methods that look only for choke points will miss many attacks, particularly more sophisticated ones. One could imagine circumstances where an attacker is able to induce more than a few connections between good and bad nodes, through mechanisms such as phishing [27, 15], or through worms or viruses that take control of good nodes long enough to create connections with bad ones. Even if mechanisms such as phishing are not available to an attacker in a particular application domain, we believe that the fewer assumptions that a mitigation scheme makes (such as the existence of a choke point), the better.

As such, we propose a fundamentally different, machine learning-based approach to detecting Sybil attacks, called the latent community (LC) model. Not surprisingly, the latent community model relies on partitioning the network into communities, which are subsets of the network that have strong internal connections. Community detection in graphs is not a new problem; in fact, it is a widely-studied component that is disconnected from the rest of the network would be an obvious sign of a Sybil attack, so the attacker also attempts to compromise (or to trick) a set of “good” nodes into linking with some of the bad nodes. Since one might assume that it is expensive or difficult for an attacker to link with a good node, the attack can be detected by finding a subnetwork that is connected to the main network via relatively few links—that is, a narrow “choke point”.

In our opinion, the presence of such a choke point is probably a good indicator of an attack, but we are concerned that methods that look only for choke points will miss many attacks, particularly more sophisticated ones. One could imagine circumstances where an attacker is able to induce more than a few connections between good and bad nodes, through mechanisms such as phishing [27, 15], or through worms or viruses that take control of good nodes long enough to create connections with bad ones. Even if mechanisms such as phishing are not available to an attacker in a particular application domain, we believe that the fewer assumptions that a mitigation scheme makes (such as the existence of a choke point), the better.

As such, we propose a fundamentally different, machine learning-based approach to detecting Sybil attacks, called the latent community (LC) model. Not surprisingly, the latent community model relies on partitioning the network into communities, which are subsets of the network that have strong internal connections. Community detection in graphs is not a new problem; in fact, it is a widely-studied component that is disconnected from the rest of the network would be an obvious sign of a Sybil attack, so the attacker also attempts to compromise (or to trick) a set of “good” nodes into linking with some of the bad nodes. Since one might assume that it is expensive or difficult for an attacker to link with a good node, the attack can be detected by finding a subnetwork that is connected to the main network via relatively few links—that is, a narrow “choke point”.

In our opinion, the presence of such a choke point is probably a good indicator of an attack, but we are concerned that methods that look only for choke points will miss many attacks, particularly more sophisticated ones. One could imagine circumstances where an attacker is able to induce more than a few connections between good and bad nodes, through mechanisms such as phishing [27, 15], or through worms or viruses that take control of good nodes long enough to create connections with bad ones. Even if mechanisms such as phishing are not available to an attacker in a particular application domain, we believe that the fewer assumptions that a mitigation scheme makes (such as the existence of a choke point), the better.

As such, we propose a fundamentally different, machine learning-based approach to detecting Sybil attacks, called the latent community (LC) model. Not surprisingly, the latent community model relies on partitioning the network into communities, which are subsets of the network that have strong internal connections. Community detection in graphs is not a new problem; in fact, it is a widely-studied component that is disconnected from the rest of the network would be an obvious sign of a Sybil attack, so the attacker also attempts to compromise (or to trick) a set of “good” nodes into linking with some of the bad nodes. Since one might assume that it is expensive or difficult for an attacker to link with a good node, the attack can be detected by finding a subnetwork that is connected to the main network via relatively few links—that is, a narrow “choke point”.

In our opinion, the presence of such a choke point is probably a good indicator of an attack, but we are concerned that methods that look only for choke points will miss many attacks, particularly more sophisticated ones. One could imagine circumstances where an attacker is able to induce more than a few connections between good and bad nodes, through mechanisms such as phishing [27, 15], or through worms or viruses that take control of good nodes long enough to create connections with bad ones. Even if mechanisms such as phishing are not available to an attacker in a particular application domain, we believe that the fewer assumptions that a mitigation scheme makes (such as the existence of a choke point), the better.

As such, we propose a fundamentally different, machine learning-based approach to detecting Sybil attacks, called the latent community (LC) model. Not surprisingly, the latent community model relies on partitioning the network into communities, which are subsets of the network that have strong internal connections. Community detection in graphs is not a new problem; in fact, it is a widely-studied component that is disconnected from the rest of the network would be an obvious sign of a Sybil attack, so the attacker also attempts to compromise (or to trick) a set of “good” nodes into linking with some of the bad nodes. Since one might assume that it is expensive or difficult for an attacker to link with a good node, the attack can be detected by finding a subnetwork that is connected to the main network via relatively few links—that is, a narrow “choke point”.

In our opinion, the presence of such a choke point is probably a good indicator of an attack, but we are concerned that methods that look only for choke points will miss many attacks, particularly more sophisticated ones. One could imagine circumstances where an attacker is able to induce more than a few connections between good and bad nodes, through mechanisms such as phishing [27, 15], or through worms or viruses that take control of good nodes long enough to create connections with bad ones. Even if mechanisms such as phishing are not available to an attacker in a particular application domain, we believe that the fewer assumptions that a mitigation scheme makes (such as the existence of a choke point), the better.
are communities $B$ and $C$, then $A$ and $C$ will usually to be closely related.

The advantage of this latent positioning is that in the LC model, attack communities tend to be outliers, since they are attached to the “good” portion of the network in a way that is inconsistent with other communities, they will tend to be pushed to the “outside” of the the latent space. For example, imagine community $A$ (consisting mostly of attackers) connects tightly with community $B$, which it has compromised. $B$ connects tightly with a central community $C$, but $A$ connects sparingly with $C$. Then $A$ will be forced away from $C$ and to the “outside” of the model. We give what appears to be a real-world example of this in Section 5.2 of the paper.

Our Contributions The contributions of this paper are:

- We propose the latent community model for partitioning a graph into subnetworks.
- We apply the LC model to detect Sybil attacks in social networks. Although there are large number of models for generating social networks and graph partitions, none of them are specifically designed for this problem.
- To ensure its acceptable practical performance, we propose a Bayesian inference approach for learning the LC model, as well as associated MCMC algorithms.
- We show experimentally that our LC-based Sybil detector competes well with algorithms for the Sybil detection from the network security literature.

2. THE LC MODEL

In the LC model, the nodes in a network are partitioned into communities, which are sets of nodes with (relatively) dense interconnections. Each community is associated with a latent position in a multi-dimensional Euclidean space (hence the name “latent community model”); the position of each community dictates how its nodes connect with other communities. Communities that are close have many links between them; far apart communities have few links.

We employ the standard, statistical machine learning approach. The LC model describes a process whereby the statistics describing node interconnections in a graph are generated. By reversing the process and figuring out exactly how a particular network could have been stochastically produced via “learning” or “inference”, we reveal the community structure of the network.

While it would be possible for a human being to directly examine the communities learned via the LC model in order to search for Sybils, in the next section we will extend the model to detect Sybils automatically.

2.1 The Generative Process

The generative process underlying the LC model is as follows. The nodes in a network are partitioned among a set of communities. The $i$th community has a latent position $\mu_i$, where $\mu_i \sim F(\theta)$. “$\sim$” should be read as “is sampled from”. $F(\theta)$ is some multi-dimensional distribution used to position the various communities in space. Any appropriate $F$ can be chosen (we will use a multi-variate normal or Gaussian distribution as well as a special-purpose “ring” distribution subsequently). We use $c_i$ to denote the vector of community sizes, so that $c_i$ is the number of nodes in the $i$th community. $E$ is the upper-triangular matrix of edge counts, where $E_{ij}$ is the number of edges between community $i$ and community $j$.

Finally, $\delta_i$ is the probability that two nodes in the $i$th community are connected.

Given this setup, the following stochastic process underlies the LC Model:

1. For each community, $\mu_i \sim F(\theta)$.

2. For each community, the number of edges connecting internal nodes is generated as $E_{cc} \sim \text{Binomial}(\binom{c_i}{2}, \delta_i)$.

3. For each pair of distinct communities, the number of cross-community edges is generated as: $E_{ij} \sim \text{Binomial}(c_i \times c_j \min(\delta_i, \delta_j) \times \eta^{-\text{ed}(\mu_i, \mu_j)})$

The above generative process is quite simple. Step (1) positions each community in space by drawing its location from a random variable having distribution $F$. Step (2) links each pair of nodes in community $i$ with probability $\delta_i$. In step (3), pairs of nodes from different communities are linked depending upon the distance between the communities—$\text{ed}(\mu_i, \mu_j)$ denotes the Euclidean distance between the latent positions associated with the communities. Thus, the probability that two nodes across communities are linked drops exponentially with increasing Euclidean distance between the communities; $\eta$ is simply a scaling factor.

It is easy to possible to make the LC model fully Bayesian by putting appropriate priors on all of the parameters. In our implementation, we give both $\delta_i$ and $\eta$ Beta(1, 1) priors. Letting $n = \sum c_i$, we give $c$ a Multinomial($\pi$, $n$) prior. In this prior, $\pi$ is a vector of probabilities, where the $i$th entry in $\pi$ is the probability that a random node falls in $c_i$. We give $\pi$ a Dirichlet(1) prior.

1This paper assumes an undirected graph; the extension to directed graphs is slight and straightforward.
2.2 Example
To make the LC model more concrete, as an illustrative example we apply it to the 2010 American FBS college football schedule [1].

The schedule for all 120 FBS football teams can be made into a 683-edge graph by placing an edge between two teams if and only if the two teams play one another. One would expect that there are twelve natural communities in this graph, because the 120 teams are organized into twelve so-called “conferences” (the conferences are called the Big Ten, the Pac Ten, the Big Twelve, etc.); the likelihood that two teams within the same conference play each other is much higher than the likelihood that two teams that are not in the same league will play each other. Thus, we assume that the graph was generated using the LC model and we attempt to learn the unseen parameters. In this example, the function \( F(\mu_i|\theta) \) is a two-dimensional standard normal distribution.

The model learned is visualized in Figure 1. The black points represent the football teams, and the red circles are the communities inferred by the model. The center of a circle denotes the latent position of the corresponding community. The magenta lines denote interconnections within communities, and the edges among communities are green.

This example illustrates two key aspects of the LC model. First, it illustrates how the notion of a “conference” or a “league” in the sporting world—where teams within a conference play each other with high frequency, and play out-of-conference with less regularity—is almost identical to the notion of a “community” in the LC model. Given the option of identifying up to twelve communities in the graph, the learning process picked out eleven (the twelfth was left empty). The learned communities correspond perfectly to the twelve FBS conferences, with the one exception that the Conference USA and Sunbelt conferences are placed together in the same community.

Second, it illustrates how the learned, latent positions of communities can correspond to real-world phenomena. In the LC model, communities that are close to each other in the latent Euclidean space are more densely connected than communities that are far away from one another. Looking at Figure 1, it is clear that the learned latent positions actually correspond roughly to the geographic locations of the various conferences. In retrospect, this makes sense. In general, schools scheduling out-of-conference games will tend to play against schools that are physically close. This is why the “WAC” (or Western Athletic Conference) and Pac 10 are along the upper right edge of the latent space—these conferences consist of football teams from schools like UCLA, USC, and BYU that are along the west coast of the USA. The “ACC” (Atlantic Coast Conference) and Big East are on the opposite side of the latent space, and consist of schools like Duke, Rutgers, and Florida State on the US east coast.

3. APPLICATION TO SYBIL DETECTION

The generic LC model as described in the previous section can be used directly (along with human examination of the learned model) to detect Sybil attacks. We will examine this approach subsequently in the experimental section. In this section, however, we extend the LC model so that it is able to automatically detect Sybil attacks. Our basic tactic will be to assume that the latent community positions are generated via a mixture of two distributions: a Gaussian distribution that positions the benign communities close to the center of the space, and a spherical distribution of attackers that surrounds the Gaussian distribution (we will call this the “ring” distribution). Nodes that are found to likely belong to the mixture component associated with the attackers are then flagged as malicious.

3.1 Assumptions
We begin with a few assumptions:

1. A special set of size \( s \) of the graph’s nodes is known to be benevolent; these \( s \) nodes are called the “seeds”.

2. Nodes in the same community are either uniformly malicious or uniformly benign.

3. Malicious communities will tend to have latent positions far from the center or “core” of the latent space.

A bit of explanation of these assumptions is warranted.

The assumption that we have a set of seeds is required to break the symmetry between benevolent nodes and Sybils, who are free to create any topology among themselves, and can thus mimic the structure of the benevolent portion of the graph. A similar idea is employed in existing Sybil defense systems: SybilGuard [34], SybilInfer [9], and SumUp [29].

The assumption of uniformity makes sense because in the LC model, nodes within communities are (by definition) connected with a uniform density. In a Sybil attack, it seems unlikely that a set of malicious nodes would be able to so thoroughly integrate themselves into a community of benign nodes that there is no real difference in the connection density between the benign nodes in the community and the attackers; even if such an integration did occur, those benign nodes would be so thoroughly compromised that labeling them as attackers would not be an egregious error.

The assumption that the malicious communities are far from the core of the latent space is justified as discussed in the introduction of the paper.

3.2 Applying the Model

Given this set of assumptions, the actual extension required to the LC model to allow for detection of Sybil attacks are as follows:

- The vector \( c \) of community sizes is now produced via a two-step process. First, the \( s \) “seed” nodes are assigned to communities using \( s \sim \text{Multinomial}(s, \pi) \); \( s_i \) is the number of seed nodes assigned to community \( i \). Then the remainder of the nodes are assigned, so that \( c \sim \text{Multinomial}(n - s, \pi) + s \).

- Each community now has an additional variable \( \phi_i \) associated with it. \( \phi_i = 1 \) if the \( i \)th community is malicious, and 0 otherwise. If \( s_i \) is non-zero, then the \( i \)th community must be benevolent (since it has a seed node) and \( \phi_i \) is zero. The status of the remaining communities is determined using \( \phi_i \sim \text{Bernoulli}(\beta) \), with an appropriate Beta prior on \( \beta \).

- Rather than having the latent position \( \mu_i \) produced by a distribution \( F \) taking only \( \theta \) as input, \( F \) now takes the form \( F(\theta|\phi) \)—that is, \( F \) is free to treat malicious and benevolent communities differently and now allow
for latent positions to be generated via a mixture. For example, if \( \phi_i \) is 1, then \( F \) can tend to scatter the community widely, but if it is 0, \( F \) will tend to locate the community centrally.

Given this extended version of the LC model, detecting a set of attackers is quite simple. Using some sort of inference (such as an MCMC algorithm or a variational method), all of the unknown parameters and variables are estimated, including which nodes are in which communities, as well as whether each community is tagged as malicious. After learning completes, all of the nodes that likely belong to a community having an \( \phi_i \) value of 1 are then returned as attacking nodes.

### 3.3 Generating Latent Positions

Thus far, we have been a bit coy as to exactly what form \( F \) should take. In principal, our model admits any distribution, although \( F \) is given an InverseGamma\((1, 1)\) prior. To generate a data point, a distance \( d \) from the origin is obtained by sampling from a one-dimensional Normal\((\rho, I)\) distribution. Then a random direction is chosen, and the latent position is placed distance \( d \) from the origin in the chosen direction.

Figure 2 depicts one example of the two-dimensional version of the resulting distribution.

### 4. POSTERIOR DISTRIBUTION

So far, we have described how a set of communities and their various interconnection statistics are generated by the LC model, given a set of parameters \( \Theta \). In standard Bayesian fashion, we will now develop a formula for \( P(\Theta | G) \), where \( G \) is the input graph (the "posterior distribution" for \( \Theta \)). This distribution can then be analyzed to infer something about the generative process that produced \( G \). In this section we consider the variant of the LC model described in the previous section, which makes use of seed nodes as well as the special "ring" distribution (the simpler version of the model described in Section 2 can be learned via a straightforward and restricted version of the learning algorithms described in this section).

We begin by assuming we are given a graph \( G = (V, S, E) \) where \( V \) is the set of nodes, \( S \) is a subset of \( V \) (\( S \) is the set of "seed" nodes), and \( E \) is the set of edges. Since we employ a Bayesian approach, given \( G \), "learning" means determining the posterior distribution \( P(\Theta | G) \) over the parameter set \( \Theta \); \( \Theta \) contains all of the unseen variables described in the last two sections.

Before we describe our learning algorithms and exhaustively list the contents of \( \Theta \), it is important to point out that the generative process embodied by the LC model does not actually generate a graph; rather, it generates the matrix \( E \) of inter- and intra-community edge-counts, and the vectors \( c \) and \( s \) of community sizes (recall that \( c \) partitions all \( n \) nodes among the communities, and \( s \) partitions the size-\( s \) subset of "seed" nodes among communities). It is significant that none of these are directly observable given a graph \( G \). Thus, to facilitate the learning of \( P(\Theta | G) \), we add to \( \Theta \) a membership vector \( m \). \( m \) indicates which of the \( n \) communities the \( i \)th node in \( G \) belongs to; if \( m_i = j \), it means that \( i \)th node belongs to the \( j \)th community.

Given a particular value for \( m \) as well as the input graph \( G = (V, S, E) \), it is possible to compute \( E \) as:

\[
E_{ij} = \sum_{(a,b) \in E} I(m_a = i)I(m_b = j)
\]

where \( I \) is the indicator function, returning one if the boolean argument is true (and zero otherwise). Likewise, \( c \) can be computed as:

\[
c_i = \sum_{v \in V} I(m_v = i)
\]

and

\[
s_i = \sum_{s \in S} I(m_s = i)
\]

Given this, \( \Theta = \{ \rho, \pi, \eta, \beta, \delta, \phi, m \} \). A plate diagram showing the relationships between the variables is described in Figure 4.

Now we are ready to derive the distribution. From elementary probability, we know that:

\[
P(\Theta | G) = \frac{P(G | \Theta)P(\Theta)}{P(G)}
\]

Also, from the generative process described in Section 2, it follows that:

\[
P(G | \Theta) = \prod_i \text{Binomial}(E_{ii}, \binom{c_i}{2}, \delta_i) \times \prod_{i > j} \text{Binomial}(E_{ij}, c_i \times c_j, \min(\delta_i, \delta_j) \times \eta^{-\text{ed}(\mu_i, \mu_j)})
\]

And from Figure 4 and the generative processes of Sections 2 and 3, (including the prior distributions listed there), it follows that:

\[
P(\Theta) = P(\rho)P(\pi)P(\eta)P(\beta)P(m | \pi) \times \prod_i P(\delta_i)P(\phi_i | \beta, s_i)P(\mu_i | \phi_i, \rho)
\]
We now briefly consider how to sample each of the marginals, to sample from simply begin with a high-level overview of Gibbs sampling. In our case, obtaining a closed form for \( P(\Theta | G) \) is very difficult, because it involves integrating out all of the variables in \( \Theta \) from \( P(G, \Theta) \). A common way around this problem (and several others associated with characterizing a complex posterior distribution such as ours) is to make use of an MCMC algorithm, such as a Gibbs sampler [3]. Some key advantages of using a Gibbs sampler are that (a) \( P(G) \) is irrelevant when applying the Gibbs sampler, and (b) the Gibbs sampler actually obtains samples from \( P(\Theta | G) \), which may be of more use than a closed form for the distribution itself. One can use those samples to estimate the mean and other interesting and/or useful statistics describing of each of the components of \( \Theta \); the samples can also be used to estimate joint statistics such as the covariances between variables. In the next subsection, we give a high-level overview of Gibbs sampling.

**Gibbs Sampling**

This gives us formulas for all of the components of \( P(\Theta | G) \), except for \( P(G) \). Unfortunately, obtaining an expression for this is very difficult, because it involves integrating out all of the variables in \( \Theta \) from \( P(G, \Theta) \). A common way around this problem (and several others associated with characterizing a complex posterior distribution such as ours) is to make use of an MCMC algorithm, such as a Gibbs sampler [3]. Some key advantages of using a Gibbs sampler are that (a) \( P(G) \) is irrelevant when applying the Gibbs sampler, and (b) the Gibbs sampler actually obtains samples from \( P(\Theta | G) \), which may be of more use than a closed form for the distribution itself. One can use those samples to estimate the mean and other interesting and/or useful statistics describing of each of the components of \( \Theta \); the samples can also be used to estimate joint statistics such as the covariances between variables. In the next subsection, we give a high-level overview of Gibbs sampling.

**5. IMPLEMENTATION**

**5.1 Gibbs Sampling**

In our case, obtaining a closed form for \( P(\theta | \Theta, G) \) up to a constant multiplicative value for any given \( \theta \) is easy; simply begin with \( P(G) \times P(\Theta | G) \) from the previous section, and then remove any of the multiplicative terms that are constant with respect to \( \theta \) (that is, remove all terms which do not include \( \theta \) anywhere in the formula).

To actually implement a Gibbs sampler, we must be able to sample from \( P(\theta | \Theta, G) \) for each possible parameter \( \theta \). We now briefly consider how to sample each of the marginals, from easiest to most difficult:

- The membership of each node \( m_i \) is a discrete number, which can be easily sampled from a multinomial distribution obtained by evaluating \( P(m_i, \Theta, G) \) at every possible value of \( m_i \).
- The variable \( \phi_i \) is one if a non-seed community is malicious, and zero otherwise. \( \phi_i \) is binary, is easily sampled from a Bernoulli distribution obtained by evaluating \( P(\phi_i, \Theta, G) \) at both \( \phi_i = 0 \) and \( \phi_i = 1 \).
- Since \( \pi \) has a Dirichlet prior, and the Dirichlet distribution is conjugate for the multinomial, sampling from \( P(\pi, \Theta, G) \) is equivalent to sampling from a Dirichlet distribution.
- Likewise, since \( \beta \) has a Beta prior, and the Beta is conjugate for the binomial distribution, \( \beta \) can be updated by first computing \( n_{\text{mal}} \) (the number of malicious non-seed communities) and \( n_{\text{ben}} \) (the number of benevolent non-seed communities) and then sample from a \( \text{Beta}(n_{\text{mal}} + 1, n_{\text{ben}} + 1) \) distribution.
- Since \( \delta_i \) has a Beta prior, we again use the conjugacy of the Beta for the binomial and \( \delta_i \) is sampled from a \( \text{Beta}(\xi_{\delta_i} + 1, (N - \xi_{\delta_i}) + 1) \) distribution.
- The parameters \( \eta \) and \( \rho \) are continuous and (in practice) single-model, so we can use a simple rejection sampler [3] to obtain values from their marginal posterior. We first perform a numerical maximization to find the value (called \( \text{max} \)) of the variable that maximizes the posterior density. We then use numerical methods to find the value to the left of the maximum (called \( \text{low} \)) where the density is 0.1% of the density at the maximum. We do the same to find a corresponding value to the right of the maximum (called \( \text{hi} \)). The envelope we use for the rejection sample then ranges from \( \text{low} \) to \( \text{hi} \) horizontally and from 0 to \( P(\text{max}) \) vertically.

This leaves only the latent position \( \mu_i \), which is a bit trickier to deal with. To sample from the posterior from \( \mu_i \), we let \( \mu_i = (d_i, \angle_i) \), where \( d_i \) is the distance of the latent position from the origin, and \( \angle_i \) is the direction (angle) of the latent position. These two variables can then be sampled separately by the Gibbs sampler. Consider \( \angle_i \) first. In a one-dimensional latent space, \( \angle_i \) can take only the values “left” and “right” and so we need only evaluate \( P(\angle_i, \Theta, G) \) at these two values and then sample from a Bernoulli to determine the direction. In a \( k \)-dimensional space with \( k > 1 \),

![Figure 3: The posterior angle of angle](image)

\[
P(\rho) = \text{InvGamma}(\rho; 1, 1) \\
P(\pi) = \text{Dirichlet}(\pi; 1) \\
P(\eta) = \text{Beta}(\eta; 1, 1) \\
P(\beta) = \text{Beta}(\beta; 1, 1) \\
P(m | \theta) = \text{Multinomial}(m | \theta, \pi) \\
P(\delta_i) = \text{Beta}(\delta_i; 1, 1) \\
P(\phi_i | \theta, m_i) = \\
\begin{cases} 
0 & \text{If } \phi_i = 1 \text{ and } m_i \geq 1 \\
1 & \text{If } \phi_i = 0 \text{ and } m_i \geq 1 \\
\text{Bernoulli}(\phi_i | \theta) & \text{otherwise} \\
\end{cases} \\
P(\mu_i | \phi_i, \rho) = \\
\begin{cases} 
\text{Normal}(\mu_i; \overline{0}, I) & \text{If } \phi_i = 0 \\
\text{Ring}(\mu_i; |\rho, I) & \text{otherwise}
\end{cases}
\]
The time complexity of our algorithm is $O(\sum_{i}^{k-1} n \times l)$, where $k$ is the number of communities, $n$ is the number of nodes, and $l$ is the number of iterations required by the Gibbs sampler. As should be clear from Figure 4, updating the parameter $m$ takes the most time, since it iterates through the Cartesian product of $n$ nodes with $k$ communities. Note that the complexity for computing the posterior probability that a node belongs to a community is $O(k)$. Given the constant number for both $k$ and $l$, the complexity of our algorithm is linear in the number of nodes.

Figure 4: The Bayesian network of our model with hyper-parameters removed. The boxes are “plates” representing replicates. The upper box represents communities, while the lower one represents nodes.

Figure 5: The convergence of our algorithm when applying to the Digg dataset.

5.3 Performance Considerations

We consider two important performance considerations when handling “real-life” social social networks: network size and handling changes in network topology efficiently. Performance is of key importance given the potential size of a social network. For example, the number of nodes in Facebook’s social network approaches one billion, and the number of active users who log on to Facebook in any given day can be 250 million [2].

The time complexity of our algorithm is $O(k^2 \times n \times l)$, which $k$ is the number of communities, $n$ is the number of nodes, and $l$ is the number of iterations required by the Gibbs sampler. As should be clear from Figure 4, updating the parameter $m$ takes the most time, since it iterates through the Cartesian product of $n$ nodes with $k$ communities. Note that the complexity for computing the posterior probability that a node belongs to a community is $O(k)$. Given the constant number for both $k$ and $l$, the complexity of our algorithm is linear in the number of nodes.

To give the reader an idea of how fast inference is in practice, we describe the results of a simple experiment over a multi-threaded, C++ implementation of our Gibbs sampler. Our implementation is around 5000 lines of source code. In our experiment, all hyper-parameters are set as described in Section 4.2. The number of communities is set to be 100 as default and $F(.)$ positions communities in a two-dimensional latent space. We run our experiments on a Linux server machine with 32, 1.0 GHz cores sharing 132 GB of memory. The experiments are performed over the Digg website data set. We obtain the Digg data from the SumUp [29] project. On Digg, people can submit or cast votes on news articles. Based on these votes, articles are ranked. Users can “follow” and be “followed” by others, inducing a directed graph. Digg relies on the feedback (votes) of its users and who follows whom. This creates a strong motivation for attacks. There are 594, 426 nodes and 5, 066, 998 directed edges in the Digg graph.

Figure 6 shows the time cost of our learning algorithm on Digg dataset. The purpose is to give the reader an idea of how fast inference is in practice. We consider different graph sizes and different numbers of communities (to increase the size of the Digg data set past its natural 594, 426 nodes, we replicate the data set and then attach it to itself to build a large connected graph). The time is almost linear in the number of nodes, given the constant number of communities.

Note that for a large graph with a large number of communities, the algorithm requires around one week to complete an iteration. This may seem like a very long time, but keep in mind that this is the running time for a single machine. Presumably, an Internet company with a very large social network will have access to a large cluster of machines—thousands of machines are not out of the question. Assuming that a parallel version of the Gibbs sampler can be developed (running on a system such as Hadoop) it seems possible to scale to hundreds of millions of nodes.

Finally, we note that an advantage of using a Gibbs sampler is that handling a dynamic network without needing to re-compute the model from scratch is easy. Node and
edge deletions and additions can be batched, and then after enough changes have been observed, the Gibbs sampler is re-run using the previous model as a starting point. New users can be randomly or heuristically assigned to communities; after just an iteration or two of the Gibbs sampler they should be correctly integrated into the model.

6. EXPERIMENTAL STUDY

While we gave a brief performance study in the prior section, this section describes a more extensive experimental study of the qualitative aspects of our models and associated learning algorithms. Our goals are twofold:

1. To illustrate how the simple LC model from Section 2 can be used to analyze a real, medium-to-large social network, and help a human expert identify potential Sybils in that network.

2. To see how our LC-based automatic detection scheme from Section 3 compares with existing, automatic Sybil detection methodologies.

As in the previous section, we use 100 communities as well as a two-dimensional latent space. Note that while we use a "magic" value of 100, it is quite easy to have the model itself learn the correct number of communities by incorporating this into the model using the well-known Chinese restaurant process [25], which is a stochastic process for randomly generating the number of model components. We have actually implemented and experimented with incorporating the Chinese restaurant process into our model, though a description has placed the communities far from the center. Figure 6 shows the positions of communities in Digg.

6.1 Utilizing the Simple LC Model

Rather than giving a proper “experiment”, we begin with an illustrative example, where we show how the simple LC model of Section 2 (with the addition of the “seed” idea from Section 3) can be used to detect Sybil attacks (at least, we think they are!) on the Digg website.

As described previously, there are 594,426 nodes and 5,066,990 directed edges in the Digg graph. We also have the date/time at which edges are created; while our model as described cannot make use of this information, it serves as a way to help validate a discovered attack. Note that the LC model as described in this paper handles only undirected graphs. While it would be easy to extend the model to incorporate directionality, in order to keep things simple, we create an edge between a pair of nodes if there exists an edge in either direction from one node to the other. After this preprocessing, there are 4,070,026 undirected edges in the graph, with an average node degree of 13.7.

We use the node “Kevin Rose” as a single seed (Mr. Rose is the founder of Digg), and attempt to learn the LC model from the graph. We run our Gibbs sampler for a “burn in” of 100 cycles. Estimates for parameter values are then obtained by taking the average over the next 100 iterations.

Figure 7 describes the latent positions of the communities we learn, with their relative density ($\delta$). Looking at the plot, we immediately noticed that there are 7 communities with $\delta \geq 0.3$, but of those, only communities 3 and 4 are distant from the center of the graph (their densities were 0.4 and 0.55, respectively). Furthermore, the communities are quite large (311 and 299 nodes, respectively).

It seems quite suspicious to us that these large communities with very high densities would be far from the center of the latent space. It is easy to explain why the learned process has placed the communities far from the center. Figure 7 depicts the so-called “relative edge densities” (abbr., “RED”) for communities 1, 2, 3 and 4, as well as for a randomly selected community. If $\delta_i$ is the internal density of community $i$, and $\delta_{ij}$ is the probability that an arbitrary node in community $i$ connects with an arbitrary node in community $j$, then $\text{RED}_{ij} = \frac{\delta_{ij}}{\delta_i}$. The figure has five plots. In the plot associated with community 1, if the line goes through point $(x, y)$, it means that $\text{RED}_{1x} = y$. We note that the plots for communities 1, 2, and the random community all are strongly correlated with one another, suggesting that these communities all have strong connections with the same central communities. However, communities 3 and 4 both have a very strange set of connections—both have almost uniformly tiny RED values except for a single spike—which explains their positions as outliers in the latent space, and calls into question their legitimacy.

When searching for some final, incontrovertible evidence of the malicious nature of these two communities, we decided to plot the histogram for the edge creation time of
Experimental Setup. Our experiments will check the five communities of the previous figure (this is depicted in Figure 9). Here we see that in both communities 3 and 4, edges are created frequently within a short time interval. In particular, the vast majority of links involving community 4 are created within a time period of only 3 days, and then all activity stops! It is hard to see how this community is not involved in a Sybil attack.

We close the subsection by noting that the LC model is not the only way this attack could have been discovered. In particular, a temporal analysis could have uncovered this attack as well. However, it would have been easy for the attacker to more carefully spread the creation of the malicious nodes over time in an attempt to hide the attack. It would have been much more difficult to hide or alter the structure of the attack and the way in which its links to the rest of the graph were atypical.

6.2 Utilizing the Automatic Detector

We now compare our automatic LC-based Sybil detector model of Section 3 with Sybil defense schemes from network literature. There are various well-known specific Sybil-defense algorithms, i.e., SybilGuard, SybilLimit, SumUp, etc. We choose two representative algorithms: SybilInfer (SI) [9] and GateKeeper (GK) [28]. SybilInfer was shown to be superior to other methods on synthetic networks [30], while Nguyen Tran, the designer of both GateKeeper [28] and SumUp [29], claims that the former is superior to the latter.

Fraction of Compromised Nodes. This is the fraction of nodes in the graph that are attackers.

Fraction of Seeds. This is the fraction of known, trusted nodes. We consider: 0.01, 0.1, and 0.25.

Fraction of Attackers. This is the fraction by which we increase the network size when we add an attack. We consider three: 0.01, 0.1, and 0.25.

Fraction of Compromised Nodes. This is the fraction of nodes that are victims of the attack. We consider three: 0.01, 0.1, and 0.25.

Experimental Setup. Our experiments will check the false positive and false negative rates of all three methods—LC, SI, and GK, over a variety of experimental settings. Our basic tactic is to take a small- to medium-sized, real network that is likely without an attack, add one or more attacks to the network, and to see how successful the methods are in discovering the attacks.

In our experiments, we systematically explore the effect of the following five variables on detection accuracy:

- The real network to which an attack is added.
- The attack topology.
- The fraction of compromised nodes.
- The fraction of nodes in the graph that are attackers.
- The fraction of seeds.

We explain each of these variables now.

Real Network. We use three real data sets to which we add an attack. The first is the “Irvine Community” data set, created from a virtual community for students at University of California, Irvine, which consists of 1,899 nodes and 13,820 edges. When a new student joined the community, he is asked to create his profile with personal information. The second is “Wikipedia Vote”, from the Stanford Large Network Dataset Collection [17]. It corresponds to vote relationships among Wikipedia users in elections for promoting individuals to be administrators [18]. The dataset consists of 7115 nodes and 100,762 vote edges, of which half are voted by Wikipedia administrators, and half are from ordinary Wikipedia users. The third is the “Gnutella Peer-to-Peer Network” dataset, also from the Stanford Large Network Dataset Collection [17]. It corresponds to a sequence of snapshots of the Gnutella peer-to-peer network from August 2002. The dataset consists of 8,717 nodes and 31,525 edges.

Attack Topology. We use three kinds of attack topologies. For each type of attack, a subset of the real nodes in the graph are chosen to be compromised nodes, and a set of new attackers are added. For the “scale-free attack” we delete all edges among compromised nodes, and then group them and the attackers together. We initially create a clique with \( m_0 \) nodes (\( m_0 \) is smaller than 6) and add the remaining nodes iteratively into the network by creating \( m \) new edges to the network. The probability for a node in the network to be selected is proportional to its degree. Finally, we add the deleted edges back in. For the “tree attack”, a random tree is constructed among the set of attackers and compromised nodes. In the “football attack”, the attacking nodes (and the links between them) are created by replicating the FBS football schedule data set from Section 2.2 repeatedly. To form connections between attackers and compromised nodes, the “scale-free” process is used.

Fraction of Compromised Nodes. This is the fraction of nodes in the data set that are victims of the attack. We consider three: 0.01, 0.1, and 0.25.

Fraction of Attackers. This is the fraction by which we increase the network size when we add an attack. We consider three: 0.01, 0.1, and 0.25.

Fraction of Seeds. This is the fraction of known, trusted nodes. We consider: 0.002, 0.005, and 0.01.

Figure 8: The relative edge density among Digg communities.

Figure 9: The creation time of edges in Digg communities.
Given these variables, for each of the three Sybil detection methods we construct a suite of tests as follows. First we define default settings for the last four variables (that is, for the attack topology, the fraction of compromised nodes, the fraction of attackers, and the fraction of seeds). For the attack topology, the default is “scale-free attack”. For the other three, the default settings are 0.1, 0.1, and 0.01, respectively. Then, for each data set, we consider each of the last four variables in order, and for a particular variable, we iterate through the three different settings, holding all other variables constant at the default values. This results in \(3 \times (\text{four variables}) \times (\text{three settings per variable}) = 36\) tests for the LC model. The other two methods only use one seed, and so they only have three variables to test, resulting in 27 tests for \(SI\) and \(GK\). For each test, we report the observed false positive and false negative rate. The results are given in Figure 11.

All three methods give some sort of score to each node, where a high score means that the method is sure that the node is an attacker. For LC, this score is the fraction of the last 100 Monte Carlo iterations that the node was in a malicious community. Thus, all must have some sort of threshold score that is used to flag a node as an attacker. For LC, we use the natural threshold of 50\%. Both \(SI\) and \(GK\) also have similar thresholds to control the trade-off between false positives and false negatives, and we also choose their values as 50\%. In order to show how critical these settings are, in Figure 10 we show the false positive and false negative rates for all three methods as a function of the threshold chosen, under the default configurations for all four parameters, for the Irvine data set.

Discussion. There are a few key results. First and foremost, over all the experiments, the LC model always resulted in the lowest false positive rate. While it is not difficult to have a low false positive rate (after all, it is easy to simply return “no Sybils” every time), it is critical. In a real-world application environment, no user is going to accept false positives with any regularity. \(SI\) and \(GK\) both show 30\% and higher false positive rates over most of the experiments. We worry that in practice, false positive rates higher than a few percent equates to a Sybil detection software being ignored.

Despite LC’s low false positive rate, it also typically had the lowest false negative rate. On both the scale-free and football attacks, and as long as 10\% or less of the benign nodes have been compromised, it has excellent detection ability compared to the other two methods. We would go as far as to say that in those cases where at least one of the detection methods was in fact practical, LC had the lowest false negative rate. In those cases where LC did not have the lowest false negative rate (for example, on the tree attack topology) the other two methods have such a high false positive rate that they are simply not applicable. For example, consider the tree attack topology on the Wikipedia data—this attack is particularly interesting, because all methods were relatively powerless to detect it. LC has a 95\% false negative rate, meaning it has almost no ability to detect the attack. But it also has a 6\% false negative rate. \(SI\), on the other hand, detects 31\% of the attack nodes, while labeling 31\% of the benign nodes as attackers. In other words, \(SI\) has no power to detect the attacks either, but while LC defaults to labeling very few nodes as attackers, \(SI\) labels 31\% as attackers. This will result in missed attacks and an annoyed user who ends up investigating a very large number of benign nodes. However, it is worthy to point out that with more Sybils introduced in the tree attack, our approach regains its ability to detect Sybils while keeping a low false positive rate (Figure 12). When the ratio between Sybils and compromised nodes is 3, our approach can detect 60\% of attackers.

7. RELATED WORK

7.1 Sybil Attacks

Sybil attacks [10] are found widely. Various prevention schemes exist, such as computational games and CAPTCHAs. The first detection schemes were based on trust or reputation: Advogato [21], Appleseed [35] and SybilProof [7]. However, reputation-based systems are vulnerable to white-washing attacks, where attackers initially behave honestly. IP or even IP cluster-based blacklists can be thwarted by techniques such as IP harvesting and Botnets.

Recently, there has been interest in leveraging network structure to thwart Sybil attacks, since it is generally assumed that it takes human efforts to establish connections among users in a reputation-based environment. Most topology-based Sybil defense algorithms [34, 33, 9, 29, 28, 20] are designed on this assumption. SybilInfer [9] (tested in this paper) represents the first machine-learning-oriented scheme for solving this problem; however it assumes the existence of a bottleneck cut and assumes that the “fast mixing” property holds in the network. Other researchers have noticed that existing algorithms work well in synthetic networks but may show poor results in practical, real social networks [30]. These findings are consistent with our own, though...
prior work excluded GateKeeper from the list of such poor-performing detection strategies (we tested GateKeeper in this paper). Viswanath et al. [30] do mention the possibility of using community detection for Sybil detection, though we are unaware of any specific work (other than ours) in this direction.

### 7.2 Community Detection

Community Detection, unlike Sybil defense, has been a well- and long-studied topic in sociology, biology, mathematics, and physics. For a good introduction to this area, Fortunato’s recent survey [11] summarizes hundreds of approaches for community detection with various ways to measure the quality of communities. In [19], Jure Leskovec and his colleagues experimentally compare a range of community detection methods based on several common objective functions.

In our opinion, there are two primary limitations when applying existing methods for community detection to the problem of Sybil detection. First, such approaches generally partition the graph into a number of communities, but there is not a natural way to distinguish benevolent communities from common ones. Viswanath and colleagues [30] intimate their opinion that for current approaches to work well, the graph should only consist of two communities for non-Sybils and Sybils respectively, which is somewhat incompatible with the fact that social networks are generally found to have many local communities or groups.

While no existing community-detection algorithm is directly applicable, it might be possible to post-process a standard community model to find Sybils. For example, hierarchical clustering [13] might be used to partition the graph into layers of communities, and finally lead to two regions, i.e., non-Sybils and Sybils. One worry is that hierarchical clustering may return vertices with incorrectly labeled communities [23], since attackers can manipulate arbitrary number of communities with uncertain connections across themselves. Another idea is to first learn the set of communities, then position them into a Euclidean space as a post-processing step, using a method such as multi-dimensional scaling [5]. But this has the obvious drawback of decoupling the community detection and embedding, which may influence one another. At any rate, perfecting any sort of post-processing-based method is bound to be difficult, and likely constitutes a full-blown research project in its own right.

The other drawback with existing methods is that most have complexity greater than $O(n^2)$, which make them unsuitable for analyzing large scale online social networks. There are few community detection algorithms [8, 32, 4] whose complexity is less or equal to $O(n \log n)$. Our concern is that most of these fast methods use the greedy algorithms to optimize some static metrics reflecting the quality of communities, i.e., modularity or conductance, which can perhaps be leveraged by Sybils to hide themselves in large scale networks.

Finally, we note that generative stochastic methods [14, 16, 24, 26] are used in analysis and modeling of social graphs, though none of those methods considered Sybil detection. Using latent positions as part of a community detection framework is not new, and others have applied such a strategy [22, 6, 31, 12]. However, such schemes tend to assign each node in the graph a latent position in a Euclidean space (in contrast, we assign each community a position in space). The drawback of a per-node assignment lies on its complexity of the learning algorithm, since the positions for each node must be inferred which is generally very expensive. In contrast, the LC model needs only the aggregate statistics of each community when positioning the communities in space.

<table>
<thead>
<tr>
<th>Attack</th>
<th>Irvine LC</th>
<th>Wikipedia LC</th>
<th>Gnutella LC</th>
<th>Irvine SI</th>
<th>Wikipedia SI</th>
<th>Gnutella SI</th>
<th>Irvine GK</th>
<th>Gnutella GK</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tree</td>
<td>0.12/0.91</td>
<td>0.68/0.68</td>
<td>0.44/0.32</td>
<td>0.06/0.95</td>
<td>0.31/0.69</td>
<td>0.46/0.34</td>
<td>0.01/0.97</td>
<td>0.31/0.70</td>
</tr>
<tr>
<td>Scale-free</td>
<td>0.01/0.20</td>
<td>0.22/0.97</td>
<td>0.41/0.41</td>
<td>0.20/0.20</td>
<td>0.33/0.95</td>
<td>0.53/0.58</td>
<td>0.02/0.26</td>
<td>0.13/0.98</td>
</tr>
<tr>
<td>Football</td>
<td>0.01/0.20</td>
<td>0.24/0.99</td>
<td>0.49/0.43</td>
<td>0.23/0.51</td>
<td>0.32/0.94</td>
<td>0.31/0.58</td>
<td>0.06/0.33</td>
<td>0.17/0.99</td>
</tr>
</tbody>
</table>

Figure 11: False positive/false negative rates over the various experiments.
9. REFERENCES

[21] Levien, R., and Aiken, A. Attack-resistant trust metrics for public key certification.