Privacy-Preserving Friends Troubleshooting Network

Qiang Huang
Princeton University
qhuang@princeton.edu

Helen J. Wang
Microsoft Research
helenw@microsoft.com

Nikita Borisov
University of California, Berkeley
nikitab@cs.berkeley.edu

***Abstract*— Content sharing is a popular use of peer-to-peer systems because of their inherent scalability and low cost of maintenance. In this paper, we leverage this nature of peer-to-peer systems to tackle a new problem: automatic misconfiguration troubleshooting. In this setting, machine configurations from peers are shared to diagnose misconfigurations on a sick machine. The key challenges are preserving privacy of individual configuration data and ensuring the integrity of peer contributions. To this end, we construct the *Friends Troubleshooting Network* (FTN), a peer-to-peer overlay network, where the links between peer machines reflect the friendship of their owners. Our FTN manifests *recursive trust* rather than transitive trust. To achieve privacy, we use the general scheme of a *historyless and futureless random-walk* for routing, during which search is carried out simultaneously with secure parameter aggregation for the purpose of troubleshooting. Our design has been guided by the characteristics of a real-world friends network, the MSN Instant Messenger (IM) network. We have prototyped our FTN system and analyzed the tradeoff between privacy and protocol efficiency.**

I. Introduction

Today's desktop PCs have not only brought to their users an enormous and ever-increasing number of features and services, but also an increasing amount of troubleshooting cost and productivity losses. Studies [15], [16] have shown that technical support contributes 17% to the total cost of ownership of today's desktop PCs. A large amount of technical support time is spent on troubleshooting failures, many of which are caused by misconfigurations.

In this paper, we address the problem of privacy-preserving, peer-to-peer misconfiguration diagnosis. We build on our previously developed algorithm for automatic misconfiguration troubleshooting, called *PeerPressure* [17]. PeerPressure uses the *common* configurations from a set of helper machines to identify the anomalous misconfigurations on the sick one. With real-world troubleshooting cases as evaluations, we have shown the effectiveness of the approach. To carry out PeerPressure diagnosis, we need to gather statistics from a sample set of helper machines. In our position paper [18], we advocated taking the peer-to-peer approach in searching for helpers and gathering statistics. As compared to a centralized database, the peer-to-peer approach provides low cost maintenance of up-to-date helper configuration samples and distributed trust. However, PeerPressure-based peer-to-peer troubleshooting poses interesting challenges in preserving the privacy of both the troubleshooting users and peer helpers, as well as the integrity of the troubleshooting results.

Ensuring *integrity* is challenging because malicious peers may lie about the applications they own and the configuration state they have, which can lead to incorrect troubleshooting results. A machine can be malicious either because its owner has ill intentions or because it is compromised by an attacker. We cope with the ill-intentioned-user problem by designing well-established social trust into the troubleshooting framework. Today, when encountering computer problems, most people first seek help from their friends and neighbors. Based on this observation, we construct a *Friends Troubleshooting Network* (FTN), which is a peer-to-peer overlay network, where a link between two machines is due to the friendship of their owners. We assume that friends will either provide authentic configuration information to each other, or, in cases where configurations are privacy sensitive, refuse to supply information, rather than giving false content.

In the real world, if Alice asks her friend Bob a question to which Bob does not know the complete answer, Bob may ask his friend Carol *on Alice's behalf*. Troubleshooting requests in the FTN are recursively forwarded to friends of friends in the same way. One may quickly conclude that our system manifests *transitive trust*. However, if Alice and Carol aren't friends, Carol may provide untruthful answers if Alice were to ask her *directly*, but truthful ones when asked by Bob. Because of this, we say that FTN manifests *recursive trust* instead.

Despite the friendship-based trust in the FTN, *privacy* remains a crucial goal: while friends can be trusted not to provide false information, they may be curious about the configuration of their peers, and such configurations may contain privacy-sensitive information. In our position paper [18], we sketched an FTN design which tries to achieve privacy through a *historyless and futureless random-walk* of an *ownerless* troubleshooting request, during which search as well as parameter aggregation are carried out for the purpose of PeerPressure troubleshooting. The design was not robust against a number of privacy-compromising attacks. In this paper, we develop our original idea into a full-fledged design and system. Because of the unique privacy and integrity requirements of our problem, previous techniques for anonymity and private data aggregation cannot be readily applied.

Our design is guided by the operational MSN Instant Messenger (IM) friends network data. With the IM data, we analyze the tradeoffs between privacy and efficiency. We have prototyped the FTN system with which a user can diagnose misconfigurations in about a minute while achieving a high

privacy level for all participants.

Coping with compromised FTN nodes remains an open challenge. We provide no mitigation mechanisms other than those already presented in [18].

For the rest of the paper, we first provide background on PeerPressure in Section II. Then, we state our privacy objectives in Section III. In Section IV, we explain our protocol by reviewing the previous design (Subsection IV-A), describing attacks against it (Subsection IV-B), and introducing a cluster-based secure multiparty parameter aggregation scheme and various other enhancements and optimizations (Subsections IV-C, IV-D, IV-E). Using the MSN IM data, we present an evaluation of our design, including analysis and simulation of the trade-off between privacy and protocol efficiency in Section V. We describe our prototype FTN system and its performance in Section VI. We compare and contrast our work with the related work in Section VII and finally conclude in Section VIII.

II. Background: PeerPressure

PeerPressure [17] assumes that an application operates correctly on most machines and hence that most machines have healthy configurations. It uses the statistics from a set of sample *helper* machines that run the same application to identify anomalous misconfigurations. The distinct feature of PeerPressure in contrast with other work in this area [19] is that it eliminates the need to manually identify a healthy machine as a reference point for comparison. We have experimented with a PeerPressure-based troubleshooting toolkit on Windows systems where most of configuration data is stored in a centralized registry. Figure 1 illustrates the operation of our PeerPressure troubleshooter.

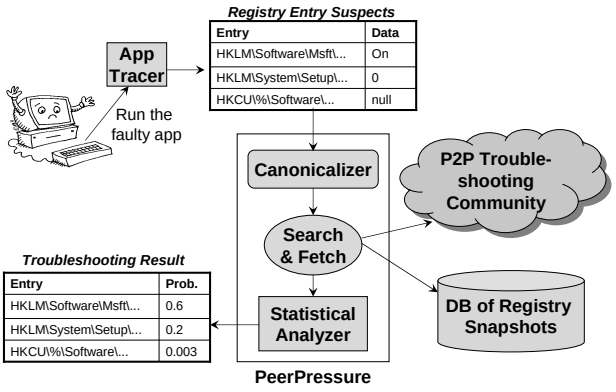

Fig. 1. PeerPressure Troubleshooter

PeerPressure first uses application tracing (with the "AppTracer") to capture the configuration entries and values that are touched by the abnormal execution of the application under troubleshooting. These entries are misconfiguration *suspects*. Then, the canonicalizer turns any user- or machine-specific entries into a *canonicalized* form. For example, user names and machine names are all replaced with constant strings "USER_NAME" and "MACHINE_NAME", respectively. Next, from a sample set of helper machines, for each suspect entry e, PeerPressure obtains the number of samples that match the value of the suspect entry M_e, the cardinality C_e (the number of distinct values for this entry among the sample set), and the most popular value for the entry. PeerPressure uses these parameters along with the sample set size and the number of suspect entries to calculate the probability of a suspect entry being the cause of the symptom: $P_e = \frac{N+C_e}{N+C_e t+C_e M_e(t-1)}$ where N is the number of samples and t is the number of suspects. The intuition behind this probability calculation is that the more conformant a suspect entry is with the samples, the more likely it is to be healthy. The top ranking entries with regard to this probability are diagnosed as the *root-cause candidates*. Then, the troubleshooting user can use the collected, most popular values for corrections.[1] The sample set can be obtained either from a database of registry snapshots collected from a large number of user machines or from a peer-to-peer troubleshooting community such as the one described in this paper. We have demonstrated PeerPressure [17] as an effective troubleshooting method: our PeerPressure troubleshooter was able to pinpoint the root-cause misconfiguration accurately for 12 out of 20 real-world troubleshooting cases and for the remaining ones, it narrowed down the number of root-cause candidates by three orders of magnitude.

III. Privacy Model and Objectives

Before we dive into our protocol design, we first state our privacy model and objectives.

A. Private Information

The information being communicated in FTN is PC configuration data. We denote the complete set of configuration data on a machine as D. A subset of D is identity-revealing, such as usernames and cookies, which we denote as D_i.[2] The canonicalizer filtering turns any user-specific entries into a *canonicalized* form (Section II). The remaining set $D_r = D - D_i$ may contain information that compromises privacy when *linked* with user identity. Some examples of such information are URLs visited and applications installed. Our privacy objective is to protect *all* peers' privacy by *anonymizing* such privacy-sensitive information in D_r; of course, D_i must never be revealed.

In addition to the configuration data, we aim to protect the identities of the sick machine and the helpers. In some cases, the mere fact that one is running a particular application may be privacy-sensitive; in our protocol, we hide whether each participant is the sick machine, a helper, or simply a forwarding node that does not run the application.

B. Attacks

We assume an operational environment where participants are honest-but-curious and never lie about their configuration information. We also assume that attackers do not know the

[1] Of course, proper roll-back mechanisms are needed if a root-cause candidate is not actually the root cause (when the correction does not remove the sick symptom).

[2] Finding all identity-revealing entries is an open research question.

FTN topology information. While it is possible to obtain friendship topology from side channels, there is much uncertainty on which friends one trusts to troubleshoot with and which ones of them are online. The ways attackers attempt to obtain private information include the following:

1) Eavesdrop on machines on the same LAN
2) Message inspection attack: Infer privacy-sensitive information by passively inspecting the messages that are passing by.
3) Polling attack: Repeatedly send fake troubleshooting requests to a friend to infer his private information.
4) Gossip attack: Friends may gossip (i.e., collude) and correlate pieces of information.

C. Existing Tools

There are many existing tools for achieving anonymity or for private data aggregation. We briefly explain why such tools are not readily applicable to our problem; a more detailed review of the related work is in Section VII.

Anonymity systems, such as mix networks [4], allow to send messages while hiding their origin. Forwarding configuration over a mix network would preserve privacy of the participants, as it would dissociate the contents from the users identities. However, it would violate our integrity model, as recursive trust is achieved only when friends communicate directly with each other. A mix network would leave no way to verify where the data came from and thus leave open the possibility of malicious configuration data.

Another way to preserve the anonymity of the contributors is to use a private aggregation or voting protocol based on a secure multi-party sum or homomorphic encryption [2]. However, these protocols work only when there is a known space of choices for the data. In our case, the space of possible values for a configuration entry is unknown, and we must determine the number of distinct values (cardinality) as well as the most popular value, while not revealing which value belongs to which participant. To solve this problem, we define a new aggregation scheme, described in Section IV-E, which uses a secure multi-party sum as a building block.

IV. Privacy-Preserving Search and Parameter Aggregation Protocol in FTN

The FTN is an overlay network similar to Gnutella [9] or Kazaa [11]; however, overlay links are made only to trusted friends' machines. We assume that friends are able to exchange public keys out of band and use them to establish secure communication channels.

We take the following basic approaches to achieve our privacy objectives in the FTN:

- *Integration of search and parameter aggregation in one transaction*: If search is a separate step, returning the IP addresses of helpers, then the querier can determine the applications running on the helpers' machines. Since application ownership could be private information, we integrate search and parameter gathering for PeerPressure into one step in such a way (next bullet) that the parameter values at any point represent the collective state for a set of friends, and therefore do not reveal any individual state.
- *Historyless and futureless random-walk routing*: To preserve the privacy of the troubleshooting user as well as node owners on the search path, we design the troubleshooting messages to be *ownerless* and not to contain any routing history or future routing state, such as the source or the nodes traversed or to be traversed. In addition, we make sure that the troubleshooting state gathered from the past is *aggregate* in nature so that individual state is disguised. Each node on the forwarding path of the random-walk is either a *forwarder* that simply proxies the request, or a *helper* that contributes its own relevant configurations to the request and then proxies the request.

For the rest of the section, we first review our previous design from our position paper [18] and the possible attacks against it in Subsections IV-A and III-B. We then present our protocol enhancements in Subsections IV-C, IV-D, and IV-E.

A. Previous Design

1) Creating a Request on the Sick Machine: A sick machine first filters out the identity-revealing entries from the suspects. Then it creates a troubleshooting request which contains 1) The name of the application executable that is under troubleshooting; 2) a random nonce $ReqID$ identifying the request; 3) the value distribution (or histogram) of each suspect entry e — a list of values and the vector $Count_e(i)$ counting the occurrences of each value i; the vector size increases over time as new values are encountered along the way; 4) Remaining number of samples needed R. The goal of the FTN protocol is for a sick machine to obtain the aggregate value distributions for all suspect entries. With the value distribution of each entry e, the sick node can extract the cardinality (C_e), the number of matches (M_e), and the most popular value to carry out the PeerPressure diagnosis.

To preserve source anonymity, the requester randomly initializes the value distribution and remaining number of samples needed. However, careful readers may realize that it is not possible to do the random initialization for value distributions since space of plausible values for each entry is most likely unknown. This was one of the unresolved issues in our previous paper. In this paper, one of our enhancements (Section IV-E) makes random initialization possible.

2) Parameter Aggregation: The sick machine establishes a secure channel with an available friend chosen at random and sends it the troubleshooting request. The friend sends an ACK if it can become either a forwarder or a helper for the request. If no ACK is received upon timeout, then the requester tries another friend chosen at random. To avoid routing loops or double-counting, if a friend has already seen the $ReqID$ of an arriving request in the past, the friend replies with a $NACK$.

A friend that receives a troubleshooting request and runs the application under troubleshooting only becomes a helper

some of the time, with probability P_h. If it always chose to participate, the second-to-last-hop node could infer information about the last-hop node. When the application under troubleshooting is very popular, with high probability, the last-hop node is capable to help. Therefore, the previous node can compare the request and reply and isolate the last-hop helper's configuration state.

A helper needs to update the troubleshooting request; for each suspect entry e, it increments $Count_e(i)$ where i is its own value for e (extending the value distribution vector as necessary if i is not already represented). Then, the helper decrements R. If R is positive, the helper proxies the request to one of its friends.

If R becomes 0, the node is the last hop. The last-hop node waits for a random amount of time, then sends the reply back to the previous hop. Without the random wait, the second-to-last hop node could know that the reply came from the last hop and compare the $Count_e(i)$'s in request and reply to obtain the last-hop node's values. The reply follows the request path back to the sick machine. The sick machine first subtracts the random initialization from the value distributions; then it performs PeerPressure diagnosis.

Each node on the forwarding path must record the $ReqID$, the request arrival time, along with the previous and next hop friend. There is a timeout associated with each request. If a node does not receive a valid reply when the timeout expires, or if it must go offline, it sends backwards the reply including the aggregate of past samples up to itself and also notifies its next hop to terminate its waiting status. We analyze the proper timeout values in Section VI.

B. Attacks Against Previous Protocol

We now present the possible privacy-compromising attacks against our previous design, as follows:

- *Gossip attacks*: A helper directly contributes its relevant configuration to the request. If the helper's previous and next hop friends collude, they can determine its configuration information.
- *Polling attacks*: Even with probabilistic helping, a curious friend may repeatedly send fake troubleshooting requests to its next hop with $R = 1$ and determine the last-hop contribution by comparing the request and reply. Even with a random wait at the last hop, the attacker can still conduct a statistical analysis to guess when the next hop contributes to the request.

C. Enhancement 1: Countering Polling Attacks by Eliminating R

To mitigate the polling attack, we avoid specifying the remaining number of hops R explicitly. Instead, each *helper* node only proxies the request further with a probability, $P_f = 1 - 1/N$, where N is the total number of samples needed; otherwise it becomes the last hop. This results in N helpers being involved on average. This probablistic proxying makes routing entirely historyless. Nodes that do not help always forward the request.

D. Enhancement 2: Countering Gossip Attacks with Clustering

We mitigate the gossip attacks through a cluster-based secure multi-party sum scheme as illustrated in Figure 2. When a node receives a troubleshooting request, instead of contributing to the request individually, it forms a *troubleshooting cluster* from its immediate friends and initiates a secure multi-party sum procedure that blends individual contributions into an aggregate that encapsulates the contributions from both the cluster and the past hops. The initiating node serves as the *cluster entrance*. A separate cluster member must be selected as the *cluster exit* for receiving the aggregate; in this way, no single node knows the aggregate contribution of a cluster.

In this section, we assume that each entry e is known to have only a few possible values (e.g. true or false); the next section explains how to change our algorithm when this is not the case. This assumption allows us to represent the value distribution of e as a fixed vector of $Count_e(i)$ for each entry e and each value i of e. The contribution of the cluster entrance includes the aggregate value distribution from the previous hops. Members who do not run the application or who choose not to help according to P_h will contribute the all zeroes vector. Members who help will set the vector element corresponding to their value to 1, and 0's for the rest.

The detailed steps of our cluster-based secure multi-party sum procedure are as follows (see also Figure 2):

1) *Random share generation and distribution:* Each cluster participant generates G random shares for its contribution vector, where, G is the cluster size. It then distributes each share to a distinct cluster member. The contribution vector also includes a value V_h, which is 0 or 1 depending on whether the member decided to help or not.
2) *Cluster exit election:* The cluster head assigns all cluster members (excluding itself) sequential numeric IDs, starting at 0. Each cluster member i selects a random nonce n_i and broadcasts a commitment [12] to it. After receiving all commitments, the members broadcast their nonces. Each member verifies all the commitments and computes the sum $n = \sum_{i=0}^{G-1} n_i$, and then picks the member with $ID = n \mod G - 1$ to be the cluster exit. This results in a fairly chosen random number in the range $0 \ldots G-1$. As an optimization, cluster members who have no friends outside the cluster can indicate this upon accepting the invitation; those members will be excluded from the choice of potential cluster exits to avoid dead ends.
3) *Unicast subtotal to the cluster exit:* Each cluster member sums up all the shares it has received from others and unicasts its subtotal to the cluster exit.
4) *Exiting the cluster:* The cluster exit sums up the received subtotals of contribution vectors from all participants. This aggregate is the value distribution from the past up to the cluster exit. The exit also sums up the received shares of V_h to obtain $\sum_G V_h$, which is the number of

cluster members that were able to help. With probability $P_f^{\sum_G V_h}$, the cluster exit further proxies the request to one of its friends, which becomes a cluster entrance of the next cluster hop. While it is possible to turn the cluster exit into a cluster entrance for the next cluster, we observe from our MSN IM data (Section V-B) that such adjacent clusters contain 14.15% overlapping members, reducing the value of the next clusters contribution.

During cluster formation, a friend can decline the cluster invitation if its friendship with the cluster entrance is considered private or if it has already seen the request. The decision about whose invitation to accept must be pre-configured by the FTN node owners. Also during cluster formation, the cluster entrance distributes the public keys of all cluster participants (that have accepted the invitation) to each of them for their future secure communications; this is necessary because the cluster participants may not be friends with one another and thus may not know each other's public keys.

The cluster exit needs to record the cluster entrance as the previous hop for the return trip of the troubleshooting request. The new cluster entrance records the previous cluster exit as its previous hop. The other cluster members only need to record the $ReqID$ to avoid loops in case they receive the same request in the future.

One may wonder whether it would be possible to just use a single, large cluster. First of all, a large cluster incurs a heavy cost because the communication cost of the multi-party secure sum procedure is of $O(G^2)$ where G is the cluster size. Also, a single cluster would not sufficiently hide the identity of the sick machine, who would be the cluster head. Finally, we must adhere to our recursive trust model by inviting only immediate friends to join the cluster. According to the MSN IM user data (Section V-B), the median number of friends a user has is 9 (some of which may not run the application under troubleshooting or may not be willing to help); since we need at least 10 helper samples for PeerPressure diagnosis [17], a single cluster is simply not sufficient.

1) Adaptive P_h for Better Privacy in case of Cluster Entrance and Exit Collusions:

Our scheme achieves very good privacy when there is no collusions between the cluster entrance and the exit. However, when they do collude, they will obtain the aggregate contribution of the cluster. The smaller the cluster is, the less privacy can we achieve with our cluster-based secure multi-party sum algorithm. In particular, if all (or most) of the cluster members decide to help, then an attacker can guess with high certainty that a given cluster member runs the application. To this end, we allow cluster participants to adaptively choose their P_h according to the cluster size and the privacy level they desire. In general, for smaller clusters or for better privacy guarantees, we must use a lower value of P_h. Of course, smaller P_h will increase the number of nodes that must be queries for each request. We give an an analysis of the trade-off between desired privacy levels and efficiency using the MSN IM user data in Section V.

2) Iterative Helper Selection: The adaptive method of choosing P_h will achieve probable innocence (i.e. when fewer than half of the members become helpers; see Section V-C) with high probability. However, to achieve this, small cluster sizes need to have a P_h near zero, thereby increasing the length of the number of clusters that need to be traversed to collect enough data, especially when the average cluster sizes are small. As an alternative, we present an iterative scheme that achieves the same privacy guarantees while collecting more contributions from neighbors.

In this scheme, before any data aggregation is performed, every cluster member, regardless of whether it runs the application or not, randomly decides whether to participate or not and sets V_p to 0 or 1. It decides to participate probability P_p, which is close to $\frac{1}{2}$. Then the cluster performs a multi-party sum to add all the V_p values count the number of participating members. If this sum includes more than half the cluster members, the cluster members discard their original decisions and randomly pick a new V_p, repeating this entire step. This process is repeated until fewer than half the members have decided to participate.

After this, the aggregation proceeds as before, except instead of using V_h to decide whether to participate, only those members who decided to participate *and* are running the application will contribute to the aggregate. All other members will contribute zero to the overall aggregate.

In the first step, all the members pick a V_p, regardless of whether they are running the application or not. This way, the count of participating members does not reveal any information. In the second step, fewer than half the members participate, hence even if the cluster-wide aggregate is intercepted, it is not known whether each member runs the application with a probability greater than one-half.

The probability P_p will depend on the size of the cluster, just as with P_h. However, in general P_p can be larger than P_h, since too many members participating results in an extra round of communication rather than a privacy compromise. The choice of P_p involves a trade-off: with a P_p too high, the first step will involve a high number of retries, increasing the communication cost. If P_p is too low, few members will participate in each cluster, which means that more clusters will be needed to collect enough samples. We explain our choice of P_p in Section V.

3) Countering Sybil Attack with Threshold-Driven Helping: A curious cluster entrance may launch a Sybil attack [6] against its friends by including in the cluster a large number of "ghost" friends who are just the cluster entrance itself. Then, with high probability, the cluster exit will be elected to be one of the ghost friends, resulting in successful collusion. One countermeasure is that a cluster member only helps when there are a threshold number T common friends of the cluster member and the cluster entrance in the cluster. With this threshold, it takes at least T colluders to expose the cluster member's contribution. However, this strategy also increases the required hop count for troubleshooting, since fewer friends will choose to be helpers. We will evaluate the

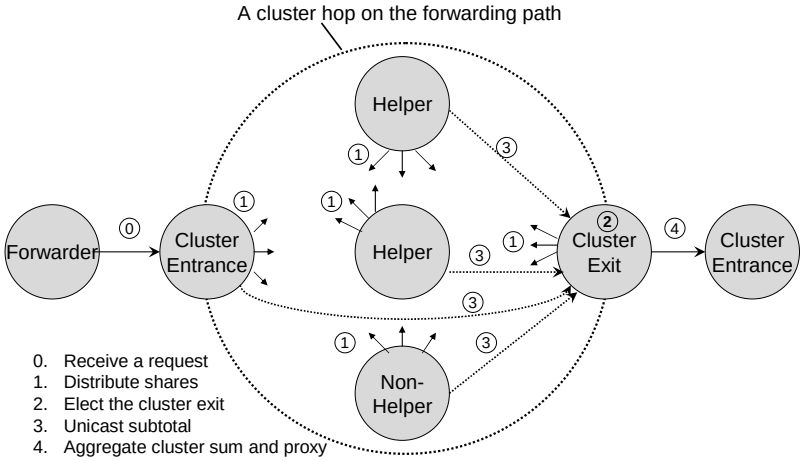

Fig. 2. Parameter Aggregation and Propagation within a Cluster.

tradeoff between the hop count overhead and the threshold scheme in Section V-C.4, using the MSN IM topology.

Members who fall below the threshold still participate in the secure sum but do not help, contributing to the privacy of other cluster members. If the complete friendship topology were known, such members could be identified and discarded from consideration by the attackers; however, one of our security assumptions is that individual friendship relationships are kept private (Section III-B).

E. Aggregate Cardinality Information

In this section, we address the case when the set of possible values for suspect entries is unknown. In this case, we cannot randomly initialize the value distribution (Section IV-A). We are also unable to perform the multi-party sum to aggregate the value distribution within a cluster, as that requires a fixed-length vector with one entry for each possible value. Instead, our scheme is to have the sick machine choose a hash function h to map values of each suspect entry into a small range $0 \ldots C-1$. The FTN nodes will then maintain the number of entry values that hash to each of the C values in the troubleshooting request. This then requires us to have a second round query to find out the most popular values of the top ranking root cause candidates yielded from the first round PeerPressure diagnosis, for the purpose of misconfiguration corrections.

The aggregate vector will contain values $Count_e(i)$ for each entry e and each value i in the range $0 \ldots C-1$. When a helper machine is updating the aggregate, it will compute the hash of its own value for entry e, V_e, and increment $Count_e(h(V_e))$. Once the aggregate is collected, the sick machine can estimate the cardinality of each entry e of the values by counting the number of non-zero $Count_e(i)$'s.

1) Choose An Appropriate Hash Range: The range of the hash function directly affects the size of the troubleshooting request, so we want to use a small hash function to reduce communication overhead. However, smaller hash functions increase the chance of hash collisions and cause the cardinality to be undercounted. For example, if the hash function has a range of C values, the estimated cardinality will never be higher than C.

Fortunately, most entries have a small cardinality and hence a lower chance of collision. Entries with large cardinalities are not likely to be identified root cause candidates by PeerPressure [17]: As the cardinality increases, the sick probability will decrease.

According to our study [17], 97% of Windows registry entries have no more than 3 values, and in 18 out of 20 real-world troubleshooting cases, the root cause entry has a cardinality of no more than 3. Therefore, we choose $C = 16$, because under-counting the cardinalities larger than 16 does not have much impact on the PeerPressure ranking of the root cause candidates.

However, even for entries with small cardinalities, there is still a chance of hash collisions. Any two values have a probability of collision of $\frac{1}{16}$ for $C = 16$, and hence many entries with cardinalities of 2 and 3 will be undercounted. We address this problem by using several hash functions, $h_1, \ldots, h_k$, and compute the histogram for each one. To estimate the cardinality, the sick machine can count the number of non-zero entries in the histogram for each hash function and take the largest count. In this case, the cardinality will only be undercounted if there is a collision in all k functions. For an entry with 2 values, the chances of this are $\frac{1}{16^k}$, so by increasing k we can make this probability arbitrarily small.

Our approach works well to measure small cardinalities accurately, while undercounting large ones. Note that using several smaller histograms is more efficient than a single histogram: a histogram with kC values will have the same communication complexity as k histograms with C values, but the odds of a collision for a hash function with range kC is $\frac{1}{kC}$ rather than $\frac{1}{C^k}$.

Based on our previous PeerPressure evaluation [17], we choose $k = 6$: Our data showed that the median number of suspect entries is 1171, with 87% of them having a single value, with 7% of them having 2 values, 3% having 3 values,

and 3% more than 3. For a 2-valued entry, the odds of a collision are $\frac{1}{16^6}$. For a 3-valued entry, the odds of at least one collision are $\left(1-\frac{15\cdot14}{16^2}\right)^6$. For a 4-valued entry, the odds are $(1-\frac{15\cdot14\cdot13}{16^3})^6$. Collisions in entries with more values are even more likely; however, we are only interested in the case where there are enough collisions to produce only 3 or fewer values in each hash function; otherwise, the collision is irrelevant as the undercounted cardinality is unlikely to cause PeerPressure to identify the entry as a root cause. The probability of such a collision is lower than the probability of a collision among 4 values, so in the following computation we consider all entries to have no more than 4 values.

The chances of a collision occuring in *any* of the 1171 entries can be calculated as:

$$1-\left(1-\tfrac{1}{16^6}\right)^{0.07\cdot1171}\cdot\left(1-\left(1-\tfrac{15\cdot14}{16^2}\right)^6\right)^{0.03\cdot1171}$$
$$\cdot\left(1-\left(1-\tfrac{15\cdot14\cdot13}{16^3}\right)^6\right)^{0.03\cdot1171}\approx 5\%.$$

Therefore, in about 5% of all troubleshooting requests, there will be some entry with an undercounted cardinality of 3 or less, which may be identified as a root cause. As we will show next, such collisions can be discovered during the second round query for the most popular values. If collisions are found, then the sick machine can retry with a different set of hash functions or larger k. The new request should include only the entries where collisions occurred and thus the communication overhead will be much smaller.

For any suspect entry e, hash collisions in this entry may cause its ranking to improve, while hash collisions in lower-ranked entries may cause them to overtake e and lower its ranking. Nevertheless, it is hard for entries with significantly larger cardinality to catch up on ranking. Therefore, in the second round, we query the most popular values of a few more top-ranking entries in order to account for possible collisions.

2) Diagnosis on the Sick Machine and Second Round Query: Using the aggregated histogram, the sick machine estimates cardinalities for all suspect entries, and then ranks suspected entries according to the PeerPressure algorithm. However, the histogram does not reveal what the correct value for the entry should be; in order to discover it, the sick machine needs to perform another round of the protocol.

The sick machine can identify the hash of the most popular value $h_j(v)=i$, where h_j is the hash function from which we obtained the cardinality. It can then ask its friends to identify which value has that hash. However, only someone who runs the same application will be able to answer this query, and we do not want to reveal who that might be. Therefore, we once again make use of secure multiparty sum to find the most popular value without compromising privacy.

In the second round of the protocol, the sick machine makes another request, this time containing a list of the top-ranking, root-cause candidate entries, as well as the hash values of the most popular value; i.e. triples (e, i, j). The second round proceeds similarly to the first one to compute an aggregate over all the participants, except that it uses the same clusters and exit nodes as the first round, rather than picking new ones. The aggregate value computed is $sum_e = \sum_{V_e|h_j(V_e)=i} V_e$. To do this, each participant who helped in the first round, and whose value for e matches the hash value in the second request ($h_j(V_e)=i$) will contribute V_e to the sum. All other members will contribute 0.

Since V_e may be a string, we have to convert it to an integer value. We do this by considering the bit representation of the string as an integer and adding it to the sum. Since all the shares in the multiparty sum must be of the same size, we take the sum over 8192-bit integers. This allows us to support strings that are up to 1024 bytes in length. The shares in this round are larger than in the first one; however, the number of entries involved is much smaller and so the communication complexity is similar to the first round.

Once the sick machine receives the aggregate from all the nodes, it can compute the most popular value V_e by dividing sum_e by the histogram value i (i.e., the number of samples with value V_e) for hash function j from the first round.[3] The result, interpreted as a string, will be the most popular value, which can then be used to repair the sick machine.

If the division results in a non-integral value, or the resulting string is not intelligible (e.g., contains non-ASCII characters), this signals that a collision occurred and there are multiple values such that $h_j(V_e)=i$. In this case, the sick machine will know that the cardinality of e was undercounted. The sick machine will need to repeat the first-round query with a different set of hash functions to obtain a more accurate cardinality estimate, and then use the second round to obtain the most popular values.

V. Protocol Evaluation

A. Security Analysis

To evaluate the security of our design, we consider the kind of information that is revealed to each participant in the protocol. Note that secure communication channels at each link render eavesdropping attacks ineffective, and hence we do not need to consider attacks from nodes who do not participate.

A cluster member that is neither an entrance nor an exit will only learn the troubleshooting query, which does not identify the sick machine and thus does not contain privacy-compromising information.

A cluster entrance will see the query and the aggregate of the contributions so far. However, since this aggregate includes a random initialization, it will not be able to find out the contributions of past clusters or find out whether the previous hop was the sick machine or simply a forwarder. It will also receive the aggregate data from the cluster exit. This data will include the contribution from the cluster as well as from any further hops. Because of the random wait, the cluster entrance will not be able to tell whether further hops were involved and isolate the contributions from the cluster members.

[3] A more robust scheme would recompute the count of contributors to V_e in order to be more tolerant of cluster members that may have left between the first and second round.

The cluster exit will receive the shares of the cluster members' contributions and will be able to compute their sum. However, this sum will include the aggregate from the past hops contributed by the cluster entrance, hence the cluster exit will not be able to isolate the contributions from the cluster. It will also receive the aggregate from the next cluster entrance. However, this aggregate will include contributions from members of the next cluster as well as potential subsequent clusters, all of which are not known to the cluster exit.

We can see that message inspection attacks at any single node do not reveal any privacy-sensitive information. Next, we will consider gossip attacks when two members collude. If a cluster entrance colludes with the cluster exit, together they will be able to determine the contributions of the other cluster members. They will still be unable to determine what each individual member contributed: the secure multi-party sum ensures that *all* other cluster members must collude to reveal the contributions of an individual member. However, they will learn some facts, such as the number of cluster members that decided to help. By reducing the probability to help, this number will be a small fraction of the cluster size and therefore not compromise the privacy of the participants. Section V-C examines the corresponding trade-off between privacy and protocol efficiency.

The fair random selection of the cluster exit mitigates the chances of a collusion between the cluster entrance and exit. If a cluster entrance has C colluders in the cluster, the chance of one of them being picked as the exit is C/G, where G is the cluster size. A Sybil attack can be used to increase these chances; on the other hand, threshold participation (Section IV-D.3) mitigates the consequences of a collusion. It is also possible for a cluster entrance to cooperate with the entrance of the next cluster and isolate the contributions of the cluster. However, this type of collusion is less likely, since the next entrance is picked by the cluster exit and never revealed to the previous entrance.

The use of a historyless random walk makes polling attacks less productive, as an attacker cannot reduce the likelihood of a friend forwarding a troubleshooting request to other clusters. Any response to a request is likely to include the aggregate information from several clusters and not reveal privacy-compromising details. As another defense against polling attacks, we limit the rate at which a node becomes a helper in queries (see Section VI).

B. Friends Network Characteristics

We obtained a snapshot of MSN IM operational data from 2003. It had 150,682,876 users. The number of friends of a IM user represents the upper limit of our cluster size. Figure 3 depicts the distribution on the number of friends, showing a median of 9, and an average of 19. We excluded those users with just one friend since these nodes will not be included on the FTN forwarding path.

The number of common friends impacts the FTN routing because FTN needs to avoid loops or double-counting (Section IV). we found that two neighboring nodes have 14.15% of common friends in average; two nodes that are two hops away from each other have 2%; three hops, 0.3%; and 4 hops, less than 0.1%. Further, we randomly picked 100 IM users with more than 4 friends (since an FTN node will not form a cluster unless it has more than 4 friends (Section IV-D.1). We find that on average, 28.92% of a node's friends do not share any common friends with the node; 21.52% have one common friend with the node; 10.61% have 2; 9.85% have 3; the remaining 29.1% have 4 or more.

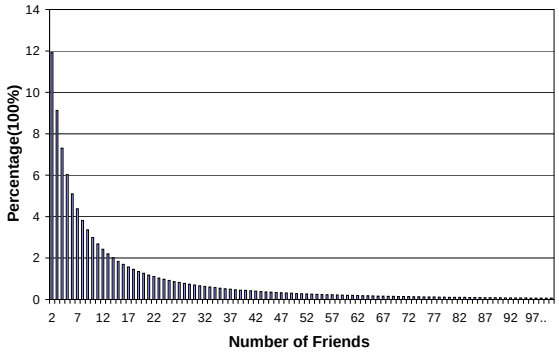

Fig. 3. Distribution of the Number of Friends

Another friends network characteristics that is of interest to the FTN is how likely is a troubleshooting request to be routed to a "dead end", a node on the forwarding path with no other friends to proxy the request on. In such cases, parameter propagation terminates without gathering enough samples. The probability of routing to such dead ends is $P_{ToDeadEnd} = \sum_i P(G = i) \cdot P_{DeadEnd}(i)$, where $P(G = i)$ denotes the percentage of clusters of size i, and $P_{DeadEnd}(i)$ represents the average percentage of dead end participants in a cluster of size i. According to our computation from our MSN IM data, $P_{ToDeadEnd} = 0.0013$. So, the average number of nodes that need to be traversed before reaching such a dead end is $1/0.0013 = 770$, which far exceeds the number of nodes that need to be traversed with the FTN protocol (Section V-C).

C. Tradeoff Analysis

1) Metric: We use the metric of *probable innocence* that was introduced by Reiter and Rubin [14] for measuring the uncertainty of a cluster member being a helper: A cluster member is considered to be probably innocent if, from the colluders' point of view, the member appears no more likely to be a helper than not to be one. This requires that no more than half of the cluster participants should help with troubleshooting, or $P_h < 0.5$. Therefore, we define the *innocence level*, $I = -log_{10}P_I$, where P_I is the probability that over half of the cluster participants help with troubleshooting. We exclude the cluster entrance and exit from the participants since we are assuming that they are colluding in order to find out the cluster aggregate. $P_I = \sum_{i=[(G-2)/2]+1}^{G-2} \binom{G-2}{i} P_h^i (1 - P_h)^{G-2-i}$ where G is the cluster size. Here, we assume the worst case scenario that every cluster member owns the application under troubleshooting — if not, P_I will be smaller. We also assume that every cluster member chooses the same P_h. Therefore,

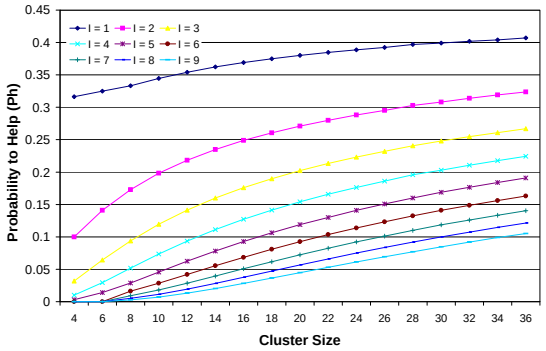

Fig. 4. The Selection of P_h for Different Cluster Sizes to Achieve Different Innocence Levels.

the higher the innocence level I is, the better the privacy is, and the smaller the P_I is. Figure 4 shows P_h's that a member should take with various cluster sizes for achieving different innocence levels. In general, P_h takes a smaller value for a higher innocence level or a smaller cluster size.

We use the average number of clusters involved in troubleshooting, $E(N_c)$, as the metric for evaluating protocol efficiency, since the troubleshooting response time $E(N_c)$ is dictated by the number of clusters a troubleshooting request traverses. (The expected number of nodes involved is approximately $19 \cdot E(N_c)$ since a node has 19 friends on average.) We have: $E(N_c) = \frac{N}{P_{own} \cdot \sum_i P(G=i) \cdot i \cdot P_{h,i}}$ where P_{own} is the percent of users that own the application under troubleshooting, i denotes the cluster size, $P(G = i)$ is the percent of clusters with size i, and $P_{h,i}$ is the probability to help according to cluster size i for a given innocence level. With the common friends' statistics from our MSN IM friends network topology (Section V-B), we estimate $P(G = i) \cdot i$ as $P(F = i) \cdot i \cdot (1 - P_{overlap})$, where $P(F = i)$ denotes the percentage of users with i friends, and $P_{overlap}$ is the percentage of the cluster entrance's friends that have already seen the request (and hence will not join its cluster). According to overlapping friends distribution, neighboring nodes have 14.15% of common friends in average, and since neighboring cluster entrances are two hops away, one can estimate the upper bound of $P_{overlap}$ as $\sum_{l=2}^{\infty}(0.1415)^l = 2.33\%$, where $(0.1415)^l$ approximates the percentage of common friends between the current cluster entrance and a previous cluster entrance that is l hops away.

2) Innocence Level Vs. Number of Clusters: Now, we use the MSN IM friendship topology to evaluate the trade-off between privacy (I) and average number of clusters ($E(N_c)$) involved in a troubleshooting event. We assume that $P_{own} = 1$ (e.g., the application under troubleshooting is very popular). We impose an upper bound on the cluster size to 36 for limiting the intra-cluster communication overhead. This reduces the average number of nodes in a cluster to 14 based on our MSN IM data. Table I shows the expected number of clusters and nodes needed to obtain 10 samples (with which PeerPressure is already effective [17]) using the static IM friendship topology (Figure 3) for achieving nine different innocence levels. We also simulated our FTN routing protocol on the static MSN IM topology, configured with various innocence levels. For each innocence level, we randomly picked 100 starting nodes as the requestor, and set $P_f = 1 - 1/N = 0.9$ for $N = 10$ samples. We list in Table I the number of clusters and nodes involved based on our simulation. One can see that in general, the number of clusters involved in our simulation is slightly smaller than our calculations. This is because we used an upper-bound estimate of $P_{overlap} = 2.33\%$ for our calculations, while in our simulation, $P_{overlap}$ is different for each cluster, and is in general less than 2.33%. In reality, the number of clusters required to collect 10 samples might be larger, since not all friends are available or own the corresponding application. It is clear from the table that the higher the privacy requirement is, the longer the routing path it takes.

3) Iterative helper selection.: Using the iterative helper selection method described in Section IV-D.2, we can guarantee probable innocence for all the cluster participants in the face of cluster entrance and exit collusion, while achieving a higher participation rate. For example, we can set the probability to participate P_p based on the cluster size to the value corresponding to $I = 1$ in Figure 4. The average number of helpers in a cluster will match the one when P_h is chosen similarly, and hence we expect the search to terminate after only 2.02 clusters. However, we can compare the privacy level to that with innocence level 9, as in both cases the probability that more than half of the cluster members will become helpers is negligible. (Of course, at innocence level 9, the expected number of helpers will be significantly less than half.)

The helper selection causes involves at least one extra round communication in each cluster, more if retries are necessary, which will happen with probability 10% in our example choice of P_p. Therefore, the latency of a request using the iterative method going through n clusters will be higher than if the adaptive P_h method is used by a factor of about 3.1/2 (as there are two rounds of communication in the adaptive P_h method). However, this is still lower than the latency of a request when innocence level 3 or higher is desired. Furthermore, since fewer machines are involved in the aggregate computation, there's less of a chance of encountering a malicious or compromised node. Therefore, the iterative helper selection method is useful when a high level of privacy is desired, and when the median cluster sizes are small.

4) Threshold-Driven Helping Vs. Number of Clusters: Now, we evaluate the path length overhead due to the use of threshold-driven helping strategy (Section IV-D.3). Based on our common friends data from the MSN IM network (Section V-B), a threshold of $T = 1$ reduces the number of helpers from the cluster by 28.92%, $T = 2$ by 50.44%, and $T = 3$ by 61.05%. Figure 5 shows the trend of average number of clusters needed with these threshold values to obtain 10 samples for nine different innocence levels.

Innocence Level	1	2	3	4	5	6	7	8	9
Expected # Clusters	2.02	2.82	3.67	4.62	5.68	6.91	8.3	9.84	11.65
Expected # Nodes Involved	28	39.4	51.4	64.69	79.55	96.8	116.2	137.82	163.11
Avg # Clusters in Simulation	2	3.1	3.59	4.55	5.78	6.75	8.18	9.49	11.27
Avg # Nodes Involved in Simulation	27.6	44.5	47.47	69.2	85.9	92.5	120.35	140.4	162.3

TABLE I

AVERAGE NUMBER OF CLUSTERS AND NODES INVOLVED TO OBTAIN 10 SAMPLES

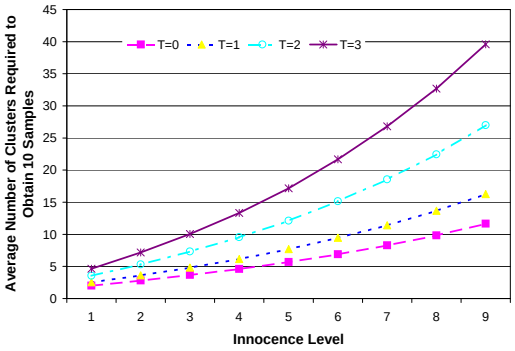

Fig. 5. The average number of clusters required to obtain 10 samples for different threshold helping strategies.

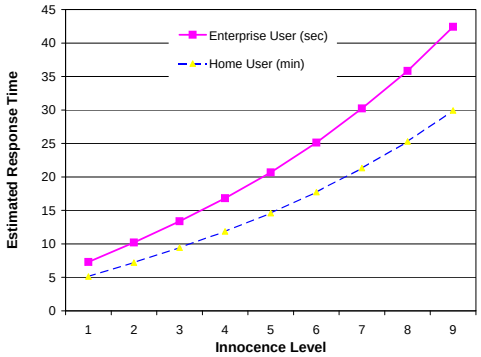

Fig. 7. The estimated average response time for enterprise users (5 Mbps available bandwidth) and home users (100 Kbps available bandwidth)

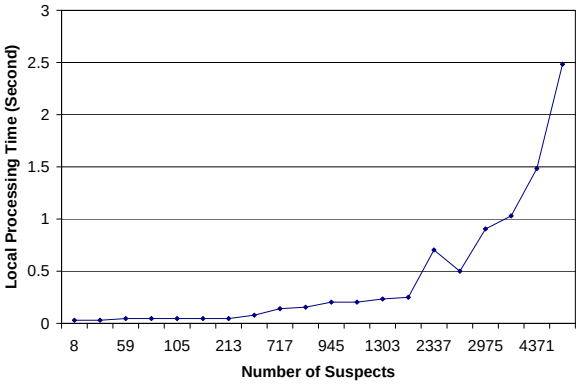

Fig. 6. Local Processing Time Vs. Number of Suspects for 20 Real-world Troubleshooting Cases.

VI. PROTOTYPE IMPLEMENTATION AND PERFORMANCE

We have prototyped an FTN system in C#. In our implementation, aside from P_h, we also set a *help budget*, in the unit of "requests per friend per day", for FTN nodes to control the rate of configuration state exposure. In addition, a disk budget is configured by an FTN user to set aside for maintaining FTN protocol state such as previous and next hops for respective $ReqID$'s that have been traversing the node. The disk budget is fair-shared among the node's active troubleshooting friends. Figure 6 shows the local processing times for the 20 troubleshooting cases under study [17]. The processing time grows with the number of suspect entries.

In terms of bandwidth overhead, for the troubleshooting cases [17] we evaluated with, there is a median of 1171 suspect entries. When cardinality is unknown, the size of the value distribution field depends on the range of the small-valued hash function. If we use six 16-valued hash function, and we reserve 1 byte for the count, the troubleshooting request message is about 100 KB. If we choose the 20 top-ranking entries as the root-cause candidates, and reserve 1024 bytes to aggregate the sum of the most popular value, the second round query message is approximately 20 KB. (Of course, the requests can be compressed to save network bandwidth.)

During the process of cluster aggregation, each participant has to transmit $M * G$ KB information, where M KB is the troubleshooting message size. Each node on the return path only needs to transmit M KB information. The number of clusters involved on the forwarding path is $E(N_c)$ on average. The return path has $2E(N_c)$ nodes, since the entrance and exit nodes of each cluster on the forwarding path are both involved to propagate the reply back along the return path. Figure 7 depicts the estimated average response time for an enterprise user with 5 Mbps available bandwidth for troubleshooting, and a broadband home user with bandwidth of 100 Kbps, when cardinality is unknown, to achieve nine different innocence levels.

Also, we can estimate the timeout that a node on the forwarding path should set. The average hop length to obtain 10 samples under innocence level 6 is $AvgHopLen = 1/(1-\bar{P_f}) = 6.9$, where $\bar{P_f} = 0.855$ is the average probability of forwarding the request from one cluster to another. The variance of the hop length is $var = (\bar{P_f})/((1-\bar{P_f})^2)$. Hence, we have $AvgHopLen + 3 \cdot \sqrt{var} = 26$. The cumulative probability of all hop lengths ≥ 26 is $\sum_{L=45}^{\infty} \bar{P_f}^{(L-1)}(1-\bar{P_f}) = \bar{P_f}^{44}(1-\bar{P_f})\sum_{L=0}^{\infty}\bar{P_f}^{L} = \bar{P_f}^{44} = 0.001$. Therefore, we choose 26 to estimate the upper limit of the hop length, and set the timeout to be 1.6 minutes for an enterprise user with 5 Mbps bandwidth, and 67 minutes for a home user with 100 Kbps bandwidth.

VII. Related Work

There is much related work in the area of anonymization. The random walk approach is also those used in FreeNet [5] and Crowds [14]. FreeNet is a distributed anonymous information storage and retrieval system. Crowds provides anonymous web transactions. Other anonymization system are based on Chaum's *mixes* [4], which serve as proxies to provide sender-receiver unlinkability through traffic mixing. Onion routing [10] extends the mixes with layers of onion-style pre-encryptions. Tarzan [7] implements the mix idea using a peer-to-peer overlay and provides sender anonymity and robustness to the mix entry point.

All of the above anonymization techniques address point-to-point communications. However, our protocol in FTN involves one-to-many communication, in the form of broadcasting a troubleshooting request to peers. This broadcast should be limited according to the friend relationships, which is more naturally implemented using a peer-to-peer overlay. Further, as discussed in Section III, our recursive trust model requires that the configuration data be transmitted between friends. Fully anonymous configuration data arriving over a mix network could not be trusted to be authentic, as only friends can be trusted not to contribue false and potentially harmful information about their configurations.

Canny [3] proposed a collaborative filtering algorithm to allow a community of users to compute a public aggregate of their data without exposing individual users' data. In his scheme, homomorphic encryption[2] is used to anonymously aggregate encrypted user data and the decryption key is not held by any single person but instead secret-shared among all the clients. The FTN targets a highly dynamic friends community where users join and leave all the time. The key share generation process would incur a high cost since new shares would have to be generated every time a user joins. Furthermore, the collaborative filtering algorithm is designed for a known, fixed set of items, while the set of values for configuration entries relevant to troubleshooting requests is not known ahead of time.

Similarly, the well known secure multiparty sum protocol enables aggregation without revealing individual private contributions; however, this protocol only supports aggregations of fixed-length vectors. We use the secure sum protocol as a building block, but we extend it to support counting the number of distinct values in a set, as well revealing the most popular value, while keeping the individual contributions private. We also make sure to send the results of the aggregate to a single node, different than the cluster entrance, such that collusion between at least two nodes is required to find out the cluster-wide sum.

Another technique for privacy-preserving data aggregation is to introduce random perturbations [1] at each input. The idea is that these perturbations would not significantly affect the aggregate, while hiding individual contributions. However, this is only true when a large number of samples are collected; with only 10 samples needed for PeerPressure, the random noise would significantly impact ranking accuracy. Increasing the number of samples for effective noise filtering would unacceptably increase the overhead of troubleshooting requests.

Our problem of privacy-preserving parameter aggregation shares much similarity to the problem of secure and privacy-preserving voting [8], [2] with three distinctions. First, voting requires voters to be authenticated by a centralized authority, such as the government. Second, our protocol has an additional requirement of participation privacy; otherwise, the privacy of the application ownership is compromised. Lastly, most voting scenarios involve a fixed, limited number of voting chances, while our troubleshooting problem does not.

The authors of SIA[13] presented a set of techniques for secure information aggregation in sensor networks with the presence of malicious sensors and aggregators. The integrity of information aggregation is achieved essentially through authentication which is identity-revealing. In FTN, we cannot do the same because of the privacy concerns.

VIII. Conclusions

In this paper, we have presented the design, implementation, and the evaluation of the *Friends Troubleshooting Network*, a peer-to-peer overlay network that aggregates privacy-sensitive configuration data from peers to carry out PeerPressure-based misconfiguration root-cause diagnosis. The links between FTN nodes reflect the friendship of their owners. The FTN manifests *recursive trust* rather than transitive trust. In FTN, we use a *historyless* and *futureless* random walk for integrated search and cluster-based parameter aggregation to achieve privacy. We further introduce a cluster-based secure aggregation protocol to find the cardinality and mode of a collection of values while preserving the privacy of individual contributions. Many of our design decisions are guided by a real-world friends network topology obtained from the MSN IM network. FTN poses interesting tradeoffs between privacy and protocol efficiency which we have analyzed in detail with the real-world friends network data. The performance of our current prototype allows enterprise users to diagnose misconfigurations in a minute with a high privacy guarantee. We believe our techniques can be applied to other application scenarios that require privacy-preserving information aggregation.

IX. Acknowledgments

Luis von Ahn, Josh Benaloh, David Brumley, John Dunagan, Yih-Chun Hu, David Jao, and Dan Simon have given us invaluable discussions and critiques on the technical content, as well as the presentation of this paper. We are grateful for their help. We also thank the anonymous reviewers for their insightful comments and suggestions.

References

[1] Rakesh Agrawal and Ramakrishnan Srikant. Privacy Perserving Data Mining. In *Proceedings of SIGMOD*, 2000.

[2] Benaloh. *Verifiable Secret-Ballot Elections.* PhD thesis, Yale University, Sept. 1987.

[3] John Canny. Collaborative Filtering with Privacy. In *IEEE Security and Privacy*, 2002.

[4] D. L. Chaum. Untraceable Electronic Mail, Return Addresses and Digital Pseudonyms. In *CACM*, 1981.

[5] Ian Clarke, Oskar Sandberg, Brandon Wiley, and Theodore W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In Proc. International Workshop on Design Issues in Anonymity and Unobservability. In *Proceedings of International Workshop on Design Issues in Anonymity and Unobservability*, 2001. Lecture Notes Computer Science Volume 2009.

[6] John R. Douceur. The Sybil Attack. In *Proceedings of the 1st International Workshop on Peer-to-Peer Systems (IPTPS)*, 2002.

[7] Michael J. Freedman, Emil Sit, Josh Gates, and Robert Morris. Introducing Tarzan, a Peer-to-Peer Anonymizing Network Layer. In *IPTPS*, 2002.

[8] T. Fujioka, T. Okamoto, and K. Ohta. A Practical Secret Voting Scheme for Large Scale Elections. In *Proceedings of Auscrypt*, Dec. 1992.

[9] The Gnutella v0.6 Protocol, Gnutella Development Forum, 2001.

[10] D. M. Goldschlag, M. G. Reed, and P. F. Syverson. Onion Routing for Anonymous and Private Internet Connections. In *CACM*, Feb 1999.

[11] KaZaa. http://www.kazaa.com.

[12] Moni Naor. Bit Commitment Using Pseudo-Randomness. In *Advanced in Cryptology — CRYPTO '89*, pages 128–136, 1989.

[13] Bartosz Przydatek, Dawn Song, and Adrian Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proceedings of ACM SenSys*, Nov 2003.

[14] Michael K. Reiter and Aviel D. Rubin. Crowds: Anonymity for Web Transactions. In *ACM Transactions on Information and System Security*, Nov 1998.

[15] M Silver and L Fiering. Desktop and Notebook TCO Updated for the 21st Century, September 2003.

[16] Web-to-Host: Reducing the Total Cost of Ownership, The Tolly Group, May 2000.

[17] Helen J. Wang, Yu Chen, John Platt, Ruyun Zhang, and Y. M. Wang. PeerPressure, A Statistical Method towards Automatic Troubleshooting. Technical Report MSR-TR-2003-80, Microsoft Research, Redmond, WA, Nov 2003.

[18] Helen J. Wang, Yih-Chun Hu, Chun Yuan, Zheng Zhang, and Yi-Min Wang. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[19] Yi-Min Wang, Chad Verbowski, John Dunagan, Yu Chen, Helen J. Wang, Chun Yuan, and Zheng Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of LISA*, 2003.