Applications of Secure Electronic Voting to Automated Privacy-Preserving Troubleshooting

Qiang Huang
Princeton University
qhuang@princeton.edu

David Jao
Microsoft Research
davidjao@microsoft.com

Helen J. Wang
Microsoft Research
helenw@microsoft.com

ABSTRACT

Recent work [27, 15] introduced a novel peer-to-peer application that leverages content sharing and aggregation among the peers to diagnose misconfigurations on a desktop PC. This application poses interesting challenges in preserving privacy of user configuration data and in maintaining integrity of troubleshooting results. In this paper, we provide a much more rigorous cryptographic and yet practical solution for preserving privacy, and we investigate and analyze solutions for ensuring integrity.

Categories and Subject Descriptors

K.4.1 [**Computers and Society**]: [Public Policy Issues-*privacy*]

General Terms

Security, Design

Keywords

Privacy, Integrity, Automatic Troubleshooting, Homomorphic Encryption, Zero Knowledge Proof

1. INTRODUCTION

Recent work [27, 15] introduced a novel (and legal) peer-to-peer application that leverages content sharing and aggregation among the peers to diagnose misconfigurations on a desktop PC automatically. The diagnosis is based on the PeerPressure troubleshooting algorithm [28]. The key intuition of PeerPressure is that misconfigurations of a PC are likely anomalous when compared with the respective configurations from other PCs having the same setting. Hence, in a peer-to-peer setting, the troubled PC collects respective configuration data from the peers. Then the anomalous-looking configuration entries on the troubled machine are diagnosed as misconfigurations, and the most popular configuration values from the peers are used as the correction values.

CCS'05, November 7–11, 2005, Alexandria, Virginia, USA.

This application poses interesting challenges in preserving privacy of user configuration data and in maintaining integrity of troubleshooting results, since configuration data often contain privacy-sensitive information and a peer is not always trustworthy. To this end, the authors of [27, 15] proposed a Friends Troubleshooting Network (FTN), which is an unstructured peer-to-peer network where a link between two machines represents friendship of their owners, and the two machines trust each other's content being exchanged. A structured peer-to-peer network is unsuitable because building up the indices for routing necessarily compromises the privacy of application ownership; for example, one may want to maintain privacy with respect to owning the KaZaa application. Furthermore, recursive trust, rather than transitive trust, is assumed in FTN — Alice trusts Bob's content, and Bob trusts Carol's content, but Alice does *not* trust Carol's content. In FTN, privacy is achieved using source-less and destination-less random-walk of a troubleshooting request initiated from a troubled machine, but where the immediate last hop and next hop are recorded on each involved node; during the random walk, respective configuration data is gathered.

There are two key limitations of the current FTN solution. The first one is that the gathered configuration data is in plain text, though in aggregated form. Some forms of collusion can give away privacy-revealing statistics. The second limitation is that the data integrity problem is not explored. In this paper, we address the first limitation by tailoring a homomorphic encryption scheme to scale with the FTN scenario. Our design has the novel property that decryption shares can be dynamically assembled among the participants during the data collection phase, with no need for a dedicated key sharing phase. To explore the integrity problem, we investigate and analyze the effectiveness of zero knowledge proof together with a branching solution where multiple branches are taken to gather the configuration data, using a real-world friends network topology. We find that when the percentage of compromised nodes is moderate or small (e.g. 1% or less), our approach can effectively reduce the risk of malicious data injection attacks to nearly zero.

For the rest of the paper, we first provide background on PeerPressure in Section 2. Then we state our attacker model in Section 3. In Section 4, we review the previous FTN solution and its limitations. We then introduce a privacy-preserving data aggregation protocol for FTN based on homomorphic encryption and various enhancements and optimizations (Section 5, 6, 7, 9). Section 8 addresses the data integrity problem. We analyze our protocol overhead in Sec-

tion 10. Using real-world instant messenger (IM) data, we present a security evaluation of our design in Section 11. A brief review of related work is given in Section 12, followed by our conclusion.

Acknowledgments: We are grateful to Josh Benaloh for providing insightful discussion and comments on an earlier draft of this paper, and to the anonymous reviewers for their numerous helpful suggestions and corrections.

2. BACKGROUND: PEERPRESSURE

The PeerPressure troubleshooting algorithm uses respective configuration data from peers to diagnose the anomalous configuration entries on the troubled machines. The operation goes as follows: PeerPressure first uses application tracing to capture the configuration entries and values that are touched by the abnormal execution of the application under troubleshooting. These entries are misconfiguration *suspects*. Next, from a sample set of helper machines, for each suspect entry e, PeerPressure obtains the number M_e of samples that match the value of the suspect entry, 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 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 configuration 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. PeerPressure has been shown to be effective in troubleshooting [28].

3. ATTACKER MODEL AND SECURITY OBJECTIVES

3.1 Security Objective

The information being communicated in FTN is PC configuration data. We denote the complete set of configuration data on a machine as D. A certain subset of D contains identity-revealing information, such as usernames and cookies, and we denote this subset as D_i. A canonicalizer first filters any user-specific entries into a *canonicalized* form[2]. The remaining set of configuration data $D_r = D - D_i$ 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 (i.e., the troubleshooter) and the peer helpers. In some cases, the mere fact that one is running a particular application may be privacy-sensitive.

[1]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.

Aside from the privacy of troubleshooting users, we also aim to protect the integrity of their contributed configuration information, since a compromised friend may lie about the configuration state it has, leading to incorrect troubleshooting results.

3.2 Attacks

We assume a friendly operational environment in FTN where attackers are simply curious friends, together with an occasional compromised machine that has not yet been repaired by its user.

Curious friends may launch passive attacks to obtain private information, but will never intentionally lie about their configuration state or alter troubleshooting results, since they do have incentives to help their friends out. Passive attacks initiated by curious friends include the following:

1. Eavesdropping on machines on the same LAN.
2. Message inspection attack: Infer privacy-sensitive information by passively inspecting the messages that are passing by.
3. Gossip attack: Friends may participate normally in a legitimate request, but simply gossip and share what they know to compromise privacy.

Friends can exchange public keys out of band and use them to establish secure communication channels, which renders eavesdropping attacks ineffective. Therefore, we do not specifically address this attack in our paper.

A compromised machine, on the other hand, may launch active attacks against its peer friends to compromise their privacy, or to corrupt the integrity of the troubleshooting result. Active attacks include the following:

1. Troubleshooter attack: A compromised host may fabricate a troubleshooting request to infer his friend's private information. For example, it may collude with a node on the forwarding path to determine the aggregate data values of all the nodes in between. We call this form of attack a troubleshooter attack, because it relies on the initiator participating in the attack by fabricating a troubleshooting request.
2. Data injection attack: A compromised host may lie about the application it owns and the configuration state it has, or tamper with other peers' contributions, leading to incorrect troubleshooting results.

Another form of passive attack that a compromised node may launch is non-participation. A compromised host may refuse to propagate the troubleshooting request it receives, or drop the response messages, causing the troubleshooting communication path to fail. However, this attack has no effect on our security objective, since by merely not participating, the attacker does not inject false configuration information into the response, and gains no private information about other peers. The troubleshooter may still seek help from other honest friends after a suitable timeout on the non-participating node has elapsed. Therefore, we do not address this attack in our paper.

4. PREVIOUS DESIGN

In this section, we briefly review the previous FTN protocol design [27, 15] and its weaknesses.

4.1 Creating a Request on the Sick Machine

A sick machine first filters out the identity-revealing entries from the suspects. This filtering step prevents information compromise via entry names, and in practice does not hurt the performance of the PeerPressure algorithm since identity-revealing entry names are unlikely to be a root cause of the symptom. 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 — that is, a list of values that e can take, and the vector $m_e(i)$ counting the occurrences of each value i of e from the sample set. 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, the number of matches, and the most popular value to carry out the PeerPressure diagnosis (Section 2).

To preserve source anonymity, the troubleshooting message is designed to be ownerless, and the value distribution field is randomly initialized by the requester.

4.2 Parameter Aggregation Through a Source-less and Destination-less Random Walk

The FTN is an unstructured peer-to-peer network where overlay links are made only to trusted friends' machines. Search for samples and parameter aggregation is integrated in a source-less and destination-less random walk on the friends overlay network topology.

The sick machine first establishes a secure channel with an available friend chosen at random and sends this friend the troubleshooting request. To avoid routing loops or double-counting, the friend responds with an acknowledgment only if it has not already seen the $ReqID$ of the arriving request.

A friend that receives a troubleshooting request and runs the application under troubleshooting only becomes a helper with probability P_h. A helper needs to update the troubleshooting request. For each suspect entry e, the helper increments $m_e(i)$ where i is its own value for e. Then, with a probability of forwarding $P_f = 1 - 1/N$, the helper proxies the request to one of its friends, where N is the number of samples needed; otherwise it becomes the last hop. This probabilistic proxying makes routing entirely history-less. Nodes that do not help always forward the request to their friends. This results in N helpers being involved on average. Each node on the forwarding path must record the $ReqID$, along with the previous and next hop friend.

The last-hop node waits for a random amount of time, then sends the reply back to the previous hop. 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.

4.3 Clustering

If a helper contributes its relevant configuration state directly, its previous and next hop may gossip to determine its configuration information. The previous design addressed the gossip attack via a cluster-based multiparty secure sum protocol.

When a node receives a troubleshooting request, instead of contributing to the request individually, it forms a *troubleshooting cluster* from its immediate friends. The initiating node serves as the *cluster entrance*. Each cluster participant represents its own contribution using the vector format $m_e(i)$. 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 cluster entrance then initiates a secure multi-party sum procedure that blends individual cluster member's contributions into an aggregate that encapsulates the contributions from both the cluster and the past hops. A separate cluster member (other than the entrance) is selected as the *cluster exit* for receiving the aggregate. With probability $P_f^{V_h}$ (where V_h is the number of helpers in the cluster), the exit further proxies the request to one of its friends chosen at random, which becomes the cluster entrance of the next hop.

4.4 Weaknesses of Previous Design and Our Motivations to Use Homomorphic Encryption

In the previous design, the gathered configuration data is in plain text. A curious entrance and exit can launch a passive attack merely by sharing what they know to compromise privacy of the cluster as a unit. In this paper, we use homomorphic encryption (Section 5) to encrypt each individual's contribution, which robustly guarantees privacy under the passive attacker model.

The previous design does not address compromised nodes. In the real world, friends' machines might be occasionally compromised, leading to active attacks against the FTN protocol, in the forms of the troubleshooter attack and the data injection attack (Section 3.2). Homomorphic encryption can be combined together with the clustering strategy (Section 6) to mitigate the troubleshooter attack. We also propose a further enhancement by forking the troubleshooting path, to make the troubleshooter attack less productive (Section 11.2).

The previous design does not provide any protection for the integrity of the troubleshooting result. A compromised host may hence launch a data injection attack by contributing false configuration information, leading to incorrect troubleshooting results. The use of homomorphic encryption enables verification of data validity via zero knowledge proof (ZKP), which together with our proposed multiple-branch troubleshooting strategy largely reduces the risk of successful data injection attacks (Section 11.3).

5. PRIVACY-PRESERVING DATA AGGREGATION USING HOMOMORPHIC ENCRYPTION

We now present an encryption scheme for parameter aggregation which provides robust guarantees of data privacy through the use of secure electronic voting protocols. For simplicity, we assume that the entry e takes on only two possible values (e.g. 0 or 1) and that clustering is not used; later we will explain how to remove these restrictions. Note that this scenario is very similar to that of a secure election: in both cases we want to generate an accurate tally of the number of participants who voted 1, without revealing who contributed a 1 and who contributed a 0. In order to accomplish encrypted data collection, we employ a type of homomorphic voting system with threshold decryption [2,

3]. The particular scheme we use is a simplified version of the ElGamal based election scheme of Cramer, Gennaro, and Schoenmakers [9]; this scheme was chosen over the others because of its optimality with respect to communication complexity. In this paper, we modify the CGS protocol so that shares of the decryption key can be aggregated in parallel with the participant recruitment and data collection steps. To our knowledge, this represents the first threshold decryption design which performs secret key sharing and data aggregation simultaneously.

Let $(G, \cdot)$ be a mathematical group which is generated by an element $g \in G$ of finite order. Given a public key g^s, where s is secret, an ElGamal encryption $E(m)$ of a message m consists of a pair $(g^r, g^{rs} \cdot g^m)$, where r is chosen randomly by the machine performing the encryption. A crucial property of ElGamal encryption is that it is *homomorphic*: given two encryptions

$$E(m_1) = (g^{r_1}, g^{r_1 s} \cdot g^{m_1})$$
$$E(m_2) = (g^{r_2}, g^{r_2 s} \cdot g^{m_2})$$

of m_1 and m_2 respectively, the encryption

$$E(m_1 + m_2) = (g^{r_1+r_2}, g^{(r_1+r_2)s} \cdot g^{m_1+m_2})$$

of $m_1 + m_2$ can be computed by multiplying the components of $E(m_1)$ and $E(m_2)$, without having to perform any decryption. If one imagines that m_1 and m_2 represent vote tallies, then the homomorphic property means that anyone can add up encrypted votes, but only those who know the secret key can decrypt the tally.

We now describe the procedure for performing parameter aggregation within an FTN request. In order to avoid concentrating the secret key in the hands of one party, we mutate the secret key at each step of the random walk, as described below. We assume that the group G and the generator g are global constants which are built into the FTN client program.

5.1 Initialization phase

- The troubleshooter picks a pair of integers r_0, s_0 at random and computes $E(m) = (g^{r_0}, g^{r_0 s_0} \cdot g^m)$, where m is the troubleshooter's value for the entry, either 0 or 1.
- The troubleshooter stores the secret key s_0, and forwards the encrypted value $E(m)$ and the public key g^{s_0} to an available friend.

5.2 Random walk phase

- Assume that the i-th node on the request path receives a public key of the form $g^{s_0+s_1+\cdots+s_{i-1}}$ and an encryption $E(m) = (g^r, g^{r(s_0+s_1+\cdots+s_{i-1})} \cdot g^m)$ of m, where m represents the number of votes for 1 which have been accumulated so far in the random walk. For convenience, write s for $s_0 + s_1 + \cdots + s_{i-1}$.
- The node picks a new secret key s_i at random, and computes the values

$$\begin{aligned} g^{s+s_i} &= g^s \cdot g^{s_i} \\ g^{r(s+s_i)} \cdot g^m &= (g^{rs} \cdot g^m) \cdot g^{rs_i} \end{aligned}$$

  The quantity $E'(m) = (g^r, g^{r(s+s_i)} \cdot g^m)$ is now a valid encryption of m under the new public key g^{s+s_i}.
- The friend replaces the old public key g^s with the new public key g^{s+s_i}, and replaces the old encrypted value $E(m)$ with the new encrypted value $E'(m)$. Note that $E(m)$ and $E'(m)$ both represent encryptions of the same value, but under different public keys.
- If the friend chooses to become a helper, then he forms an encrypted message $E'(m_i) = (g^{r_i}, g^{r_i(s+s_i)} \cdot g^{m_i})$ under the new public key where m_i is his own value for the entry and r_i is chosen randomly, and uses the homomorphic property to compute the encrypted tally $E'(m + m_i) = (g^{r+r_i}, g^{(r+r_i)(s+s_i)} \cdot g^{m+m_i})$.
- The new public key and the new encrypted tally are forwarded to the next node. The secret key s_i is stored for use during decryption.

5.3 Decryption phase

- The last hop has a public key of the form $g^{s_0+\cdots+s_t}$ and an encrypted value $E(m) = (g^r, g^{r(s_0+\cdots+s_t)} \cdot g^m)$ representing the final tally of the number of votes for 1. Since he owns the secret key s_t, he can compute

$$\begin{aligned} g^{s_0+\cdots+s_{t-1}} &= (g^{s_0+\cdots+s_t})/g^{s_t} \\ g^{r(s_0+\cdots+s_{t-1})} \cdot g^m &= \frac{(g^{r(s_0+\cdots+s_t)} \cdot g^m)}{g^{rs_t}} \end{aligned}$$

  Note that $(g^r, g^{r(s_0+\cdots+s_{t-1})} \cdot g^m)$ is a valid encryption of m under the public key $g^{s_0+\cdots+s_{t-1}}$.
- The last hop sends the encrypted value $(g^r, g^{r(s_0+\cdots+s_{t-1})} \cdot g^m)$ to the previous hop.
- Proceeding inductively, the i-th node possesses a public key of the form $g^{s_0+\cdots+s_i}$ and receives an encrypted value $E(m) = (g^r, g^{r(s_0+\cdots+s_i)} \cdot g^m)$, and sends the encrypted value $(g^r, g^{r(s_0+\cdots+s_{i-1})} \cdot g^m)$ to the previous hop.
- The troubleshooter receives $(g^r, g^{rs_0} \cdot g^m)$ from the first node, and recovers

$$g^m = (g^{rs_0} \cdot g^m)/g^{rs_0}$$

  using the stored value of s_0.

5.4 Tallying phase

- Since the tally m is guaranteed to be less than the number of participants t in the FTN request (typically, under 256), the troubleshooter can find m by computing $g^0, g^1, \ldots, g^t$ and stopping at the first one of these which matches g^m.
- For very large values of t, the value of m can be found in $O(\sqrt{t})$ time using the time-space tradeoff known as baby-step-giant-step: form a table of values $T = \{g^{j\lceil\sqrt{t}\rceil} \mid j = 0, 1, \ldots, \lceil\sqrt{t}\rceil\}$, and compute g^m/g^i for $i = 0, 1, \ldots, \lceil\sqrt{t}\rceil$, stopping at the first value of i for which $g^m/g^i = g^{j\lceil\sqrt{t}\rceil}$ appears in T. The value of m will be $i + j\lceil\sqrt{t}\rceil$.

Readers who are familiar with the framework of [9] will recognize the above scheme as an additive (t, t) threshold decryption scheme where each FTN node is an election authority. The main difference here is that we accumulate shares of

the secret key dynamically during the random walk phase instead of instantiating secret key shares statically during the initialization phase. In principle, more general (t, n) threshold decryption schemes such as [21] and [24] could provide greater robustness against non-cooperative nodes, but this does not help here since random walk path is historyless and the return packet already needs the cooperation of every FTN node in order to make its way back to the troubleshooter. This reliance on honest behavior means that the encryption scheme given here does not protect a user's privacy against active attacks such as the troubleshooter attack. Instead, the purpose of the encryption is to protect users from *passive* attacks where we assume the attackers are curious friends who log and share whatever data they acquire in the course of responding to a legitimate request, but do not alter results or fabricate false data in any way. In the passive attack model, [9, Theorem 2] combined with the Diffie-Hellman assumption shows that no coalition can gain any information about individual votes except for that which is implied by the votes which are cast by the coalition and the final tally.

Another advantage of using homomorphic encryption is that we can require the participants to give zero knowledge proofs of data validity, in order to show that all protocol operations are legitimate and that they are not manipulating the tallies in any way. Since zero knowledge proofs interact with the protocol elements described in the next few sections, we defer discussion of the topic to Section 8.

6. CLUSTERING

In addition to protecting users against gossip attacks, the cluster-based secure multi-party sum protocol of [15] provides some protection against troubleshooter attacks by limiting the extent to which any single machine's information can be isolated. As we will see in this section, it is possible to combine the clustering enhancement together with encrypted vote tallying without making any major changes to either scheme.

The clustering procedure requires four steps:

1. Random share generation and distribution: Each cluster member generates $G-1$ random shares for its contribution vector where G is the cluster size, and distributes each share to a distinct cluster member. The shares are signed with the sender's public key. Note that these shares do not need to be encrypted, since no proper subset of the shares reveals any information about the value of the vector.

2. Cluster exit election: This step proceeds unchanged, except that in addition to selecting a cluster exit, a number of *keyholders* are also selected (typically, one keyholder per cluster will suffice). The cluster entrance broadcasts the first component g^r of the encrypted message to each keyholder. Each keyholder in turn generates a secret key s_i as in the basic aggregation protocol, and unicasts the quantities g^{s_i} and g^{rs_i} to the cluster entrance. The cluster entrance then calculates the new public key using the values g^{s_i}, and broadcasts the new public key g^{s+s_i} to the cluster, where g^s is the old public key it received from the previous hop.

3. Unicast subtotal to the cluster exit: Each cluster member sums up all the shares it has received, encrypts this sum with the new public key, and unicasts its encrypted subtotal to the cluster exit. In addition, the cluster entrance modifies the old encrypted total to use the new public key, and adds the old total into his subtotal using the homomorphic property before sending the combined total to the cluster exit.

4. Exiting the cluster: The cluster exit uses the homomorphic property of the encryption scheme to sum up the received subtotals from all participants, and then sends this encrypted total along with the new public key to the next recipient. All other aspects of the protocol remain unchanged.

On the return trip, the encrypted result is relayed to each keyholder and decrypted in the same manner as in Section 5.

7. DEALING WITH UNKNOWN CARDINALITIES

We now consider the case where the set of possible values for the entry is unknown. As in [15], this problem can be solved by using a hash function h to map the values of the entry into a small numerical range $1, \ldots, C$. The FTN nodes will then maintain the number of entry values $(m_1, m_2, \ldots, m_C)$ that hash to each of the C values in the troubleshooting request. In other words, the voting scheme must maintain a vector of tallies instead of a single tally. There are two ways that this can be achieved:

1. Pick a predetermined list of generators $g_1, g_2, \ldots, g_C$, and encode a vector $\mathbf{m} = (m_1, \ldots, m_C)$ as
$$E(\mathbf{m}) = (g^r, g^{rs} \cdot \prod_{i=1}^{C} g_i^{m_i}).$$

2. Encrypt each value m_i separately, and transmit a list of encryptions $(E(m_1), \ldots, E(m_C))$ instead of a single encryption.

Option 1 increases the complexity of the tallying phase of the protocol — instead of recovering a single value m given g^m, the troubleshooter now has to recover an entire vector $\mathbf{m} = (m_1, \ldots, m_C)$ given only the single value $\prod_{i=1}^{C} g_i^{m_i}$. Assuming each coordinate m_i has maximum size t, the recovery of $\mathbf{m}$ will take $O(t^C)$ time using the naive brute-force search method, or $O(t^{C/2})$ if the baby-step-giant-step method of Section 5 is used. For this reason, Option 1 can only accommodate values of C up to about $C = 4$ before the computational overheads become prohibitive.

Option 2 is less computationally taxing because the computational overhead increases only linearly in C instead of exponentially. However, the size of each packet also increases by a factor of C under Option 2. Since we want to keep bandwidth utilization to a minimum, we use a hybrid scheme wherein we transmit a list of encryptions of length C_2 as in Option 2, but each element of that list in turn encodes a short vector of length C_1 as in Option 1. Even a modest value of C_1 such as 3 will reduce the bandwidth overhead by a factor of 3.

In [15] the authors propose using a range of 96 values per entry partitioned into six independent hashes of 16 values

Prover		Verifier
$(x, y) = (g^{s_i}, g^{rs_i})$		
$w \in \mathbb{Z}$ chosen randomly		
$(a, b) = (g^w, g^{rw})$	$\xrightarrow{a,b}$	
	$\xleftarrow{c}$	$c \in \mathbb{Z}$ chosen randomly
$z = w + cs_i$	$\xrightarrow{z}$	$g^z \stackrel{?}{=} ax^c$
		$g^{rz} \stackrel{?}{=} by^c$

Figure 1: Zero knowledge proof that $y = x^r$, given g and g^r

each. Using these parameters, the probability of any root cause candidate triggering a simultaneous collision in all six hash functions is under 5% for typical troubleshooting requests. If we encode this list of 96 values using $C_1 = 3$ and $C_2 = 32$, then a median troubleshooting request consisting of 1171 suspect entries requires maintaining a list of ≈ 37000 atomic encryptions each representing three tallies.

8. ZERO KNOWLEDGE PROOFS

8.1 Proofs of Decryption

Certain portions of the protocol can take advantage of zero knowledge proofs of validity to ensure that the encrypted data values are not tampered with during aggregation. For example, during the decryption phase, each keyholder must divide the public key by some quantity $x = g^{s_i}$ and the encrypted portion of the data by the quantity $y = g^{rs_i}$. If the previous and next nodes wish to verify that the keyholder is performing a legitimate decryption operation (and not, for example, altering the data values), they may combine their shared knowledge of the pre-decryption and post-decryption packets to infer what values of x and y were used in the decryption phase. The keyholder can then execute the interactive Chaum-Pedersen proof of knowledge protocol [6] given in Figure 1 to prove that the values of x and y satisfy the relation $x = g^{s_i}$ and $y = g^{rs_i}$, without revealing his private key s_i. For practical applications, the Fiat-Shamir heuristic [11] is used to implement the protocol non-interactively by using pseudorandom hash functions to determine the values of the random inputs used in the interactive protocol.

8.2 Proofs of Validity

We now address the problem of proving validity of votes. By "validity" we mean that each vote is syntactically valid in the sense that it increments at most one component's total by one. We do not attempt to guarantee that a valid vote accurately reflects the actual internal configuration of the machine, since such a guarantee is impossible to achieve. Therefore, the goal here is to allow participants to prove that their contribution $E(\mathbf{m})$ represents the encryption of a message $\mathbf{m} = (m_1, \ldots, m_C)$ where at most one of the values m_i is 1 and the rest are 0, without revealing the values themselves. The generic construction of [10] provides a method to transform the Chaum-Pedersen zero knowledge equality protocol of Figure 1 into a zero knowledge proof of validity for encrypted ballots of this form, at the cost of increasing communications complexity by $O(C)$. For example, if we assume for simplicity that $C = 1$, then the interactive protocol of Figure 2 (or its non-interactive Fiat-Shamir analogue)

Prover			Verifier
If $M = 1$	If $M = g_1$		
$w, r_1, d_1 \in \mathbb{Z}$	$w, r_2, d_2 \in \mathbb{Z}$		
$x = g^r$	$x = g^r$		
$y = g^{rs} \cdot 1$	$y = g^{rs} g_1$	$\xrightarrow{x,y}$	
$a_1 = g^{r_1} x^{d_1}$	$a_1 = g^w$		
$b_1 = g^{r_1 s} \left(\frac{y}{g_1}\right)^{d_1}$	$b_1 = g^{ws}$	$\xrightarrow{a_1,b_1}$	
$a_2 = g^w$	$a_2 = g^{r_2} x^{d_2}$		$c \stackrel{?}{=} d_1 + d_2$
$b_2 = g^{ws}$	$b_2 = g^{r_2 s} \left(\frac{y}{1}\right)^{d_2}$	$\xrightarrow{a_2,b_2}$	$a_1 \stackrel{?}{=} g^{r_1} x^{d_1}$
		$\xleftarrow{c}$	$a_2 \stackrel{?}{=} g^{r_2} x^{d_2}$
$d_2 = c - d_1$	$d_1 = c - d_2$	$\xrightarrow{d_1,d_2}$	$b_1 \stackrel{?}{=} g^{r_1 s} \left(\frac{y}{g_1}\right)^{d_1}$
$r_2 = w - rd_2$	$r_1 = w - rd_1$	$\xrightarrow{r_1,r_2}$	$b_2 \stackrel{?}{=} g^{r_2 s} \left(\frac{y}{1}\right)^{d_2}$

Figure 2: Zero knowledge proof that $(x, y) = (g^r,\ g^{rs} \cdot M)$ where $M = 1$ or $M = g_1$.

suffices to prove that an encrypted message $(g^r, g^{rs} \cdot g_1^{m_1})$ satisfies either $m_1 = 0$ or $m_1 = 1$, without revealing which of the two is the case. In general, for a given value of C, the proof requires transmitting $C + 1$ quadruplets (a_i, b_i, d_i, r_i), and expands the bandwidth requirement of the protocol by a factor of $2C + 2$.

If we transmit a vector of C_2 encrypted messages as described in Section 7, then we can transmit zero knowledge proofs of validity for each component of a contributed vector, and apply the Shamir secret sharing scheme as described in [10] to pick the challenges c in such a way that the validity of the vector as a whole can be verified (i.e., no more than one component contains a vote). The total cost in bandwidth remains $2C_1 + 2$ times that of the original unvalidated vector, since the $O(C_2)$ cost factor is already reflected in the size expansion of the original unvalidated vector.

Even if this bandwidth cost is too high to allow all of the first round data to be validated, it still makes sense to perform retroactive validation of the top ranking entries which appear in the second round (Section 9), since the integrity of these entries is especially crucial to the success of the troubleshooting request.

8.3 Zero Knowledge Proofs and Clustering

Zero knowledge validity proofs can be combined with clustering by having each cluster member encrypt and sign the $N - 1$ shares that it distributes to each of the $N - 1$ other members of the cluster. Instead of relying on the shared keyholders' encryption key, we require each member to generate a new encryption key from scratch in order to prevent malicious keyholders from colluding with the cluster exit and decrypting the encrypted values later. Since this encryption key is different from the encryption key used in the homomorphic tally, the cluster member should also include a zero knowledge proof that the encrypted share has the same value as the unencrypted share, using the zero knowledge equality proof of Figure 1. The recipients of each share should then verify that the equality proof is correct—if it is incorrect, then they should publish both the signed original share and the signed encrypted share and proof, so that the other cluster members can see that the proof is incorrect.

After verifying the equality proofs, the recipients forward

the encrypted shares to the cluster exit and the cluster exit combines the encrypted shares using the homomorphic properties of the encryption scheme to recover an encrypted version of the original raw vote. The owner of this vote can then provide a zero knowledge proof as in Section 8.2 to demonstrate that this encrypted value represents a valid vote.

9. SECOND ROUND QUERY

The use of a hash function to digitize all entry values means that the troubleshooting machine only knows the hash of the most popular value of a top ranking, root-cause candidate entry returned by the PeerPressure diagnosis (cf. Section 2), and not the entry value itself. In order to communicate the actual most popular values of the top ranking entries to the troubleshooter for the purpose of correcting misconfigurations, we perform another round of queries using a Chaumian-style mixnet [5] to protect the identities of the machines having those entries.

The second round uses the same clusters, keyholders, entrance and exit nodes as in the first round. For each top ranking, root-cause candidate entry e, the troubleshooter queries the network asking for participants whose value for entry e has a hash value equal to the known most popular hash value of entry e. Those participants with the matching hash value and who helped in the first round convert their actual entry values to integers V_e (using ASCII strings, say), and V_e will be their contribution in the second round. All other participants set their contribution to 0. Similarly to the first round, each cluster member first generates and distributes random shares of its second round contribution to every distinct cluster member. Each cluster member then sums up all the shares that it has received, and encrypts this subtotal m using the formula

$$E(m) = (g^r, g^{rs} \cdot m),$$

where r is chosen randomly and g^s is the public key it used in the first round (for the i-th cluster, $s = s_0 + \cdots + s_i$). If the length of m exceeds the length of g^{rs}, then we use generic transforms such as CFB or CBC to convert a cipher on a short block to one on a long block. If m is shorter than g^{rs}, then suitable padding such as OAEP is applied to make it match g^{rs} in length. Finally, each cluster member unicasts its encrypted subtotal to the cluster exit.

In addition, the cluster entrance modifies each old encrypted message $E(m_{old}) = (g^{r_{old}}, g^{r_{old}(s_0+\cdots+s_{i-1})} \cdot m_{old})$ that it has received from the previous hop to use the new public key, and computes $E'(m_{old}) = (g^{r_{old}}, g^{r_{old}(s_0+\cdots+s_{i-1})} \cdot g^{s_i} \cdot m_{old}) = (g^{r_{old}}, g^{r_{old} \cdot s} \cdot m_{old})$. As in the first round, the value $E'(m_{old})$ constitutes a valid encryption for m_{old} under the new public key g^s. The entrance then appends its own subtotal, encrypted under the public key g^s, and passes the encrypted messages to the cluster exit. The set of all encrypted messages from a cluster is collected into one large packet and then passed to the next cluster, using the same entrance and exit nodes as in the first round.

In order to protect source anonymity, the troubleshooter should initialize the second round query with a number of randomly selected encrypted messages which will be discarded upon conclusion of the query.

On the return path, each keyholder decrypts his share of the secret key from each of the incoming messages as in the first round, and then passes the messages to the next keyholder *in randomly permuted order*. We also require that each decrypter *re-encrypt* the messages by transforming an encrypted message $(g^r, g^{rs} \cdot m)$ into an equivalent encryption $(g^{r+r'}, g^{(r+r')s} \cdot m)$ for a randomly chosen value of r', in order to prevent other participants from correlating the g^r component of an encrypted message back to its originator. If the decrypters do not re-encrypt, then the keyholder of the last cluster in the chain (who is also the first decrypter) can store the previous cluster's value of g^r and collude with the troubleshooter to determine which decrypted subtotal corresponds to the previous cluster's contribution.

In the final step, the troubleshooter decrypts all the messages and recovers all the subtotals using its stored value of s_0. It then discards the random messages used to initialize the second round request, and sums up all the decrypted subtotals to obtain sum_e, which is an aggregate of the most popular entry value V_e. The troubleshooter can compute the most popular value V_e by dividing sum_e by the number of helpers who contributed V_e in the second round, i.e., the number of helpers whose entry e has a hash value matching the most popular value. This number can be read from the value distribution histogram that the troubleshooter received in the first round. The division result, interpreted as an ASCII string, will be the most popular value for the top-ranking suspect entry e, which can then be used to repair the sick machine.

The effect of this protocol is to implement a simple threshold re-encryption mixnet guaranteeing data privacy, without providing any integrity checks on the decryption process. The techniques of Section 8 already protect the integrity of the first round hash values which determine the PeerPressure diagnosis, and any node attacking the troubleshooter with a data injection attack will not know the number of helpers and hence at worse can only induce random faults in the second round output, which can be easily recognized and remedied by repeating the query (cf. Section 11.3).

A cluster exit participating in a troubleshooter attack (cf. Section 3.2) could in principle fabricate all of the public key data originating in the request path, and collude with the keyholder to decrypt each member's subtotal. Together with the troubleshooter, they will be able to find out the aggregate value of V_e within their cluster. However, they still cannot determine which individual member contributed V_e if there is any. The forking mechanism of Section 11.2 can further reduce this threat of troubleshooter attacks.

10. RESOURCE USAGE

10.1 Bandwidth Overhead

If we assume the parameter values given at the end of Section 7, then a median first round troubleshooting request of 1171 entries requires handling about 37000 ElGamal encryptions, each one consisting of two elements of G. The bit length of a request is therefore directly proportional to the number of bits needed to represent an element of G. Traditional ElGamal encryption uses a group G equal to the multiplicative group of a finite field, for which 512 bits per group element is considered necessary for minimal security, and 1024 bits for good security. At these group sizes, an FTN request consisting of 37000 encryptions would be about 4.8 MB in size for a 512 bit group, or 9.6 MB for a 1024 bit group. These estimates do not take into account any extra overheads which would be incurred by zero knowledge proofs.

Using elliptic curve based ElGamal groups [19], we can achieve the same level of security using fewer bits. It is estimated [20] that a 110 bit elliptic curve achieves cryptographic security equivalent to that of a 512 bit finite field, and a 139 bit elliptic curve is comparable to a 1024 bit finite field. These group sizes result in FTN request sizes of 1.0 MB with 110 bit elliptic curves, or 1.25 MB with 139 bit elliptic curves.

For the second round, if we use 1024 bits per suspect entry and include the 20 top ranking suspect entries in the mixnet then the total size of each user's collection of encrypted entries is $40 * 1024$ bits per user, for a final accumulated packet size of 1 MB assuming that 200 users participate in the mixnet. Therefore, the packet size is comparable to the first round.

The above first round and second round figures only apply to communications between cluster entrances, cluster exits, and keyholders, and not to intra-cluster communications within a single cluster. The latter transmissions do not need encryption because the cluster-based secure multi-party sum protocol already provides sufficient protection against privacy attacks. Accordingly, the bandwidth requirements for communication within a single cluster remain the same as in [15].

10.2 CPU Overhead

Recall from Section 7 that the tallying phase requires the troubleshooter to recover, for each suspect value, a list of $C_2 = 32$ vectors each of the form $\mathbf{m} = (m_1, m_2, m_3)$ where m_i ranges from 0 to the maximum number of participants in an FTN request. If we assume no more than 255 participants in a request, then each of the values m_i ranges from 0 to 255 and we must recover the vector $\mathbf{m} = (m_1, m_2, m_3)$ given the quantity $g_1^{m_1} g_2^{m_2} g_3^{m_3}$; this recovery must be done 32 times per suspect value, or a total of ≈ 37000 times assuming a median request size of 1171 candidate suspects.

We are mainly concerned with the amount of CPU time needed to recover these 37000 vectors, since every other computation required by the protocol is at least an order of magnitude less time consuming than the recovery step. Since there are 2^{24} possible values for each $\mathbf{m}$, and 37000 different vectors $\mathbf{m}$ to recover, it would be too slow to recover the vectors $\mathbf{m}$ by brute force trial and error. Instead we use the baby-step-giant-step search algorithm from Section 5. We compute a lookup table of the values $g_1^{m_1} g_2^{m_2} g_3^{m_3}$ for approximately 2^{20} different values of (m_1, m_2, m_3). Since our table contains one out of every sixteen possible values of $\mathbf{m}$, we can recover any given $\mathbf{m}$ using at most 16 successive exponentiations and table lookups, or 37000 values using at most $37000 \cdot 16 \approx 2^{20}$ such operations.

In Figure 3, we present some experimental measurements for the amount of CPU time used in an actual recovery operation of this scale. Our test program makes use of the optimized finite field arithmetic routines in the MS bignum library. We use elliptic curve groups over prime fields, with field sizes indicated in Figure 3. All tests represent the averages of ten trial runs and were performed on a 3 GHz Pentium machine.

11. PROTOCOL EVALUATION

We evaluated our protocol based on a real-world friends network topology. It is a snapshot of MSN instant messenger (IM) operational data from 2003, with 150,682,876 users.

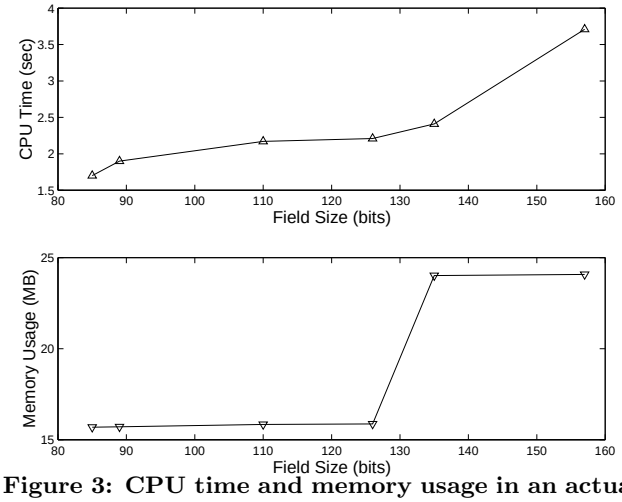

Figure 3: CPU time and memory usage in an actual recovery operation.

The number of friends of an IM user has a median of 9, and an average of 19, which represents the upper limit of our cluster size.

11.1 Robustness Against Passive Attacks

Homomorphic encryption protects users' privacy from passive attacks where the troubleshooting requests and the contributed data are all legitimate, but friends are "curious" and might collude and snoop to try to infer other peoples' data (however without going to the lengths of falsifying information).

Occasionally, compromised hosts may launch active attacks against their peer friends, in the forms of the troubleshooter attack and the data injection attack (Section 3.2). The homomorphic encrypted vote tallying can be combined together with the cluster-based secure multi-party sum protocol to increase robustness against active attacks, which we will evaluate in the following subsections.

11.2 Mitigating the Troubleshooter Attack

A compromised host may fabricate a troubleshooting request and form a troubleshooting cluster to infer its friends' information. Without colluding, message inspection at the troubleshooter does not reveal any privacy-sensitive information, due to the use of the homomorphically aggregated network-wide tally in the first round, and the re-encryption mixnet in the second round. By colluding with the cluster's exit, the malicious troubleshooter can determine the aggregate contributions from other honest participants within its cluster, but will still be unable to determine what an individual member contributed.

Based on clustering, we propose an enhancement to fork the request path, to further reduce the risk of a successful troubleshooter attack. If two exits are selected for a cluster, and each cluster member randomly chooses one of them to unicast the subtotal of the random shares it receives from other participants, then the troubleshooter would have to collude with both exits in order to reconstruct the cluster's aggregate information. Due to the use of a random walk based on a probability of forwarding P_f (cf. Section 4.2), the troubleshooter cannot reduce the likelihood of an honest exit forwarding its request to other clusters. Hence, the response from an honest exit is likely to include the aggregate information from multiple clusters and not reveal privacy-

compromising details. In the first round, the key holder will wait until it receives the replies from both exits, and aggregate the encrypted tally contained in both replies using the homomorphic property. In the second round, the key holder will wait until it receives both replies, collect all the encrypted messages contained in both replies into one packet and mix them by random shuffling.

Assume a malicious troubleshooter forms a cluster of size G, among which C members are colluding with it. The probability of a successful troubleshooter attack with only one exit is $P_1 = \frac{C}{G}$. If the forking strategy is used, the probability is reduced to $P_2 = \frac{C(C-1)}{G(G-1)}$. We have $\frac{P_2}{P_1} = \frac{C-1}{G-1}$. If only a small portion of the cluster's participants collude with the troubleshooter, then P_2 will be much smaller than P_1. Obviously, the more forks we use, the less likely that a troubleshooter is able to successfully launch the attack, since it has to collude with more exits.

To quantitatively measure the uncertainty of the malicious troubleshooter about whether the decrypted tally is from its own cluster, or has been mixed with contributions from future hops, we use an information theoretic metric based on Shannon's definition of entropy [25]. In the ideal situation, where there are no colluding participants, the troubleshooter only has the information that under probability $1 - \bar{P}_f$, the reply only contains the aggregate data values from its own cluster, where $\bar{P}_f$ is the average probability of forwarding the request from one cluster to another. The troubleshooter's uncertainty can be quantified as $H_{ideal} = -\bar{P}_f \log(\bar{P}_f) - (1 - \bar{P}_f) \log(1 - \bar{P}_f)$. When there are C colluders in the participants, the troubleshooter's uncertainty is reduced to $H_{nofork} = \frac{C}{G} \cdot 0 + (1 - \frac{C}{G}) \cdot H_{ideal} = (\frac{G-C}{G}) \cdot H_{ideal}$. However, if two exits are selected to fork the troubleshooting path, the troubleshooter cannot infer whether future hops are involved unless it colludes with both exits. If only one of the exits colludes with the troubleshooter, then under probability $\bar{P}_f$, the honest exit will forward the request to future hops. We note that the troubleshooter and the colluding exit cannot determine if future hops are involved from the reply returned by the honest exit in the first round, since they are unaware of the actual number of helpers inside the cluster. However, since every participant contributes an encrypted subtotal in the second round, and the messages are concatenated, they may infer that no future hops are involved by counting the number of encrypted messages contained in the reply returned by the honest exit in the second round. To prevent the troubleshooter from gaining extra information in such cases, we require the honest exit to randomly select a number of subtotals which sum up to 0, encrypt them and add the encrypted messages into the second round reply, if it does not forward the request due to P_f or it encounters a dead end situation. Therefore, with two exits to fork the request path, the troubleshooter's uncertainty becomes $H_{twofork} = \frac{C(C-1)}{G(G-1)} \cdot 0 + \frac{(G-C)(G-C-1)}{G(G-1)} \cdot H_{ideal} + 2 \cdot \frac{C(G-C)}{G(G-1)} \cdot H_{ideal} = \frac{(G-C)(G+C-1)}{G(G-1)} \cdot H_{ideal}$. The forking strategy increases the troubleshooter's uncertainty by a factor of $\frac{H_{twofork}}{H_{nofork}} = \frac{G+C-1}{G-1} = 1 + \frac{C}{G-1}$. Figure 4 shows the ratio of $\frac{H_{twofork}}{H_{ideal}}$ and $\frac{H_{nofork}}{H_{ideal}}$, for different values of C and G. The closer the ratio is to 1, the more robust the system is against the troubleshooter attack.

If a compromised host launches a successful troubleshooter attack by colluding with both exits, together they will be

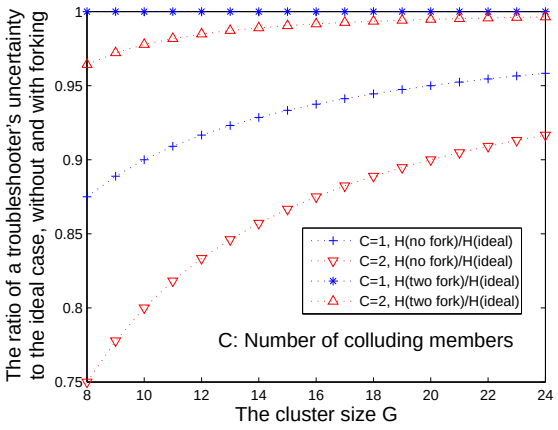

Figure 4: The ratio of a troubleshooter's uncertainty to the ideal case, with and without forking strategy.

able to determine the collective contributions of the other honest cluster members. Nevertheless, they will still be unable to determine each individual member's data: the secure multi-party sum protocol in both rounds ensures that all other cluster members must collude to reveal the contribution of an individual member. A cluster member can adjust its probability to help P_h (cf. Section 4.2) to improve its privacy in case of a successful troubleshooter attack. In general, P_h should take a smaller value for a smaller cluster size and for better privacy. Although a malicious troubleshooter may invite fewer friends when forming its troubleshooting cluster, the honest cluster members can always make the number of helpers a small fraction of the cluster size, by reducing P_h according to the cluster size, and therefore their privacy will not be compromised. We note that some configuration data or application ownership information (e.g., owning Microsoft Word) is not privacy-sensitive. For those cases, an FTN node simply sets $P_h = 1$. In general, an FTN user can configure policies on how to adjust P_h for data of different privacy levels.

Since forking at every hop doubles the number of clusters involved in the troubleshooting process, we need to use an appropriate probability of forwarding to achieve a relatively short path length per branch. Let L denote the average path length per branch; then the total number of clusters involved is $\sum_{i=0}^{L-1} 2^i = 2^L - 1$ on average. The probability of forwarding P_f can be determined by equations *(Average number of helpers per cluster)* $\cdot (2^L - 1) = 10$ and *(Average number of helpers per cluster)* $\cdot 0.5 \cdot L = \frac{1}{1-P_f}$. The first equation is due to the fact that the total number of samples to be collected should be around 10, in order for the PeerPressure diagnosis to be effective [28]. The second equation corresponds to a branch in the forking scenario, where the average number of times that an exit flips a biased coin (to determine if it should forward the request or not) is equal to half of the average number of a cluster's helpers, and in total the coin will be flipped $\frac{1}{1-P_f}$ times on average.

In general, we want L to be a small value. A short path length not only saves communication overhead, but also reduces the chance of encountering a compromised peer which may contribute false configuration data and affect the in-

tegrity of the troubleshooting result. If we set $P_f = 0.7$, corresponding to the average path length $L = 2$, then the total number of clusters involved would be 3 on average, the same as an Innocence level[3] of $I = 2$ in [15]. Therefore, a cluster member can set the probability to help based on its cluster size to the value corresponding to $I = 2$, in order to gather 10 samples.

We simulated our FTN routing protocol with the forking strategy on the static IM topology, configured with probability of forwarding $P_f = 0.7$. We randomly picked 100 starting nodes as the requester, and set the corresponding P_h according to the Innocence level of $I = 2$, in order to obtain approximately 10 samples. In our simulation, we imposed an upper bound of 36 on the cluster size to limit the intra-cluster communication overhead. When we set $P_f = 0.7$, the median number of clusters involved is 3, and the median number of nodes involved is 70. In occasional cases, the number of clusters involved is more than 16, which incurs a high communication overhead. The initiator may send a message via the request path to cancel the request propagation, after an appropriate timeout period has elapsed. The timeout should be chosen randomly and kept private to the initiator itself.

Another friends network characteristic that is of interest to the forking mechanism is the percentage of clusters that can only find one exit. In such cases, the cluster members will not contribute any sample in order to prevent the troubleshooter attack. The request can still be propagated through the unique exit. Encountering such nodes on the forward path will increase the communication overhead without helping to gather any samples. According to our computation from the 81790827 IM users with at least 3 friends (we excluded those nodes with one or two friends, since the former will not be included on the FTN forwarding path, while the latter will not form a cluster), the probability of routing to such nodes is 0.0025. Therefore, the average number of hops that need to be traversed before reaching such a node is $1/0.0025 = 400$, which far exceeds the number of hops that need to be traversed with the FTN protocol (typically under 10).

11.3 Mitigating the Data Injection Attack

Ensuring the integrity of the troubleshooting result is challenging considering the occasional possibility that a friend's machine may be compromised, and hence may lie about the configuration state it has.

An advantage of using homomorphic encryption is that the key holder can be asked to prove in zero knowledge that its decryption operation is legitimate, and each cluster's exit can require the participants to give zero knowledge proof of the validity of their vote, to ensure that the encrypted tally is not tampered during aggregation within the cluster. However, a cluster's entrance and exit can still manipulate the tally on the propagation path, since unlike each cluster member's binary valued vote that can be verified in zero knowledge, the randomly initialized tally can take any integer value, and hence its validity is hardly possible to verify. Nevertheless, use of zero-knowledge proof does reduce the number of nodes that can corrupt the tally from G (the cluster's size) to 2 per hop.

To further limit the impact of data injection attacks, a troubleshooter can send several requests to disjoint subsets of friends, enabling each subset to carry out a branch of troubleshooting, and examine the results returned by all branches to check if any one of the results is anomalous. The troubleshooter can filter out maliciously tampered results if the majority of the branches do not encounter any compromised nodes. We call this a multiple-branch troubleshooting strategy.

Guaranteeing the integrity of the return values requires that all samples must be gathered from the troubleshooter's own cluster, and every participant must verify the validity of their contribution to the troubleshooter itself. However, there are several reasons that this requirement cannot be met. First of all, it would compromise the troubleshooter's anonymity. Furthermore, the PeerPressure algorithm needs 10 samples to be effective. According to our IM topology, 51% of users have fewer than 10 friends, and hence must collect samples from their friends' clusters. Finally, a troubleshooter would easily determine the aggregate information of its immediate friends if a single cluster were used. Involving several clusters in the troubleshooting process renders the troubleshooter attack harder to launch.

We assume that there is only an occasional possibility for a trusted friend's machine to get compromised, and hence the percentage of compromised nodes is moderate or small (e.g., 1% or less). To quantitatively study the effect of using zero knowledge proof and multiple-branch troubleshooting on mitigating the data injection attack, we performed simulations on our IM topology under two different attacker scenarios, in which we assumed the percentage of compromised nodes was 1% and 0.1% respectively. For each attacker scenario, we randomly marked the nodes being compromised. We then randomly picked 1000 honest nodes to send a troubleshooting request to 1 to 10 distinct branches, depending on the number of friends they may have. We simulated the troubleshooting cases when different numbers of clusters were involved on each request path. With zero knowledge proof, we marked the result returned by one branch as being corrupted only if the request hit a compromised entrance or exit on that branch; otherwise, if the request encountered any compromised node during its propagation, we marked the result as being tampered. A requester's troubleshooting was considered unsuccessful if half or more of its branches returned a tampered result. Then under each attacker scenario, we computed the probability of a successful data injection attack as the ratio of failed troubleshooting.

Figure 5 compares the probability of successful data injection attacks with and without zero knowledge proof protection, for different percentages of compromised nodes, average path lengths, and the number of branches the requester selects to send a troubleshooting request. Based on Figure 5, we have the following observations:

- The risk of data injection is smaller when a single path is used than two branches. This is because the multiple-branch troubleshooting is only effective when the majority of the branches return an untampered result. If only two branches are used, as long as one of them encounters a compromised node, the troubleshooting fails since the troubleshooter cannot determine which branch's result is valid, and since more nodes are involved compared to using a single branch, the risk of encountering a compromised node is increased. Therefore, if the troubleshooter has only a

[3]Innocence level is defined as a metric in [15] to evaluate different privacy levels.

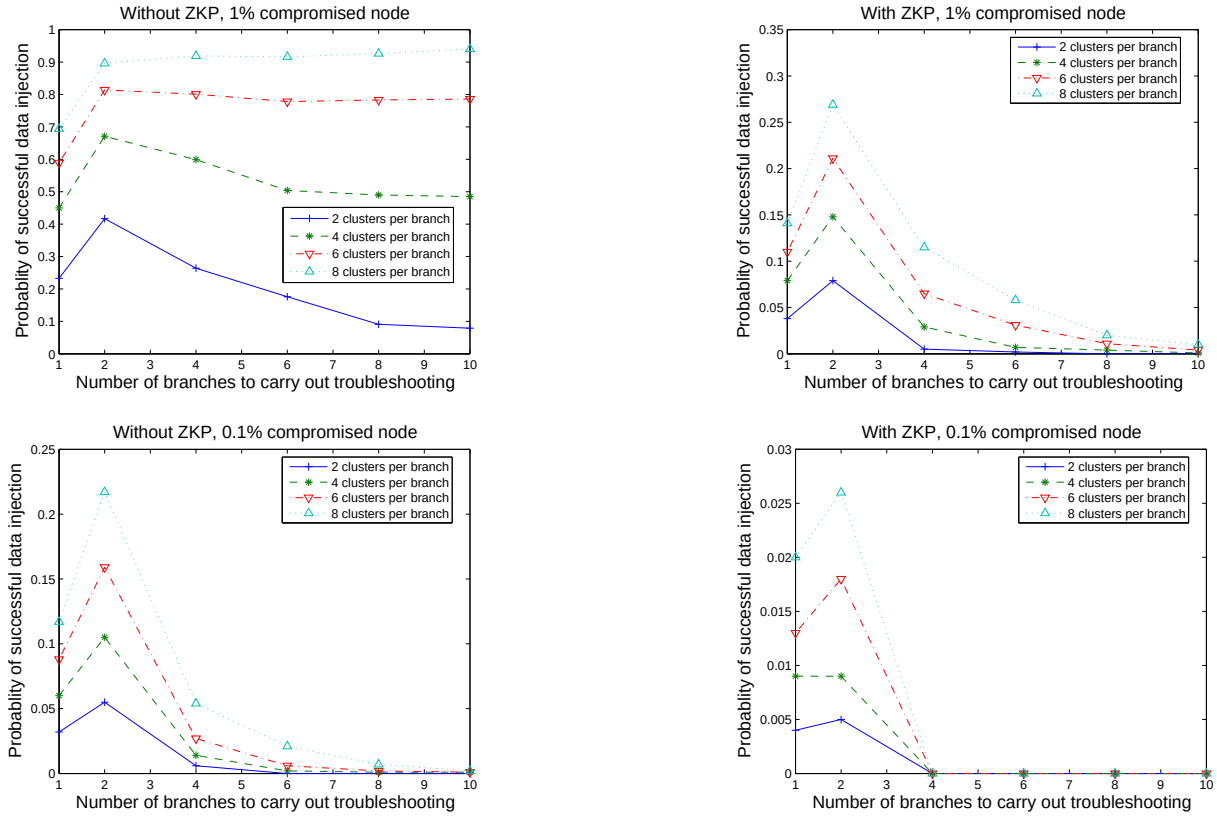

Figure 5: The probability of a successful data injection attack when the troubleshooter sends out a request on multiple branches.

limited number of friends (e.g., less than 4), it should use a single path for troubleshooting.

- When the percentage of compromised nodes is moderate or small (e.g., 1% or less), the multiple-branch troubleshooting strategy works particularly well together with zero knowledge proof, since it is less likely that over half of the branches will be compromised. The combination is especially efficient at reducing the risk of data injection when a large number of clusters are involved on a branch. For example, if there are 1% compromised nodes and the path length is 8, the probability of successful data injection attacks on a single path troubleshooting request without zero knowledge proof protection is 0.694; the probability is reduced to 0.14 if zero knowledge proof is enabled, and issuing the request over 6 branches further reduces the risk to 0.058, 8 branches to 0.02 and 10 branches to 0.01.

- In general, involving more clusters increases the risk of successful data injection attacks. Although a smaller path length is desirable by the troubleshooter, the peers in FTN tend to choose smaller values of P_h to achieve a higher privacy level in case of a troubleshooter attack, which forces the request to traverse a longer path to gather enough samples. The iterative helper selection method [15] guarantees probable innocence[4] for all the cluster participants in the face of a successful troubleshooter attack, while achieving a higher helping rate, and the request propagation terminates after only 2 clusters on average. Then with zero knowledge proof, the risk of a successful data injection attack is below 0.04 when the percentage of compromised nodes is 1%, which is further reduced to 0.005, 0.002 and nearly 0 if the request is sent to 4, 6 and 8 branches respectively. When 0.1% nodes are compromised, the risk is below 0.004 and further reduced to nearly 0 with 4 or more branches.

- Finally, issuing the request over multiple branches incurs a higher communication overhead than using zero knowledge proof, since we only validate the few top ranking entries that make it through to the second round. Therefore, a single path is preferable if the size of the troubleshooting message is large (e.g., the number of suspect entries is large) and the bandwidth assigned for troubleshooting is limited.

When the set of possible values for suspect entries is unknown, we use a second round query to find out the most popular values of the top ranking root cause candidates (Section 9). However, the contribution of participants in the second round is the unknown entry value that we need to query for, and hence its validity cannot be verified. A compromised host may thus contribute a false entry value to corrupt the aggregate sum_e. The consequence is that during the process of recovering the actual entry value, the division will result in a non-integral value, or the resulting string will not be intelligible (e.g., contains non-ASCII characters). In this way, the troubleshooter is able to recognize the corruption, and hence discard the invalid result. The troubleshooter may still send the second round request to a different subset of friends to seek the entry value.

Our multiple-branch troubleshooting strategy increases the robustness of the second round query against tamper-

[4]A participant is probably innocent if, from the attackers' point of view, it appears no more likely to be a helper than not to be one.

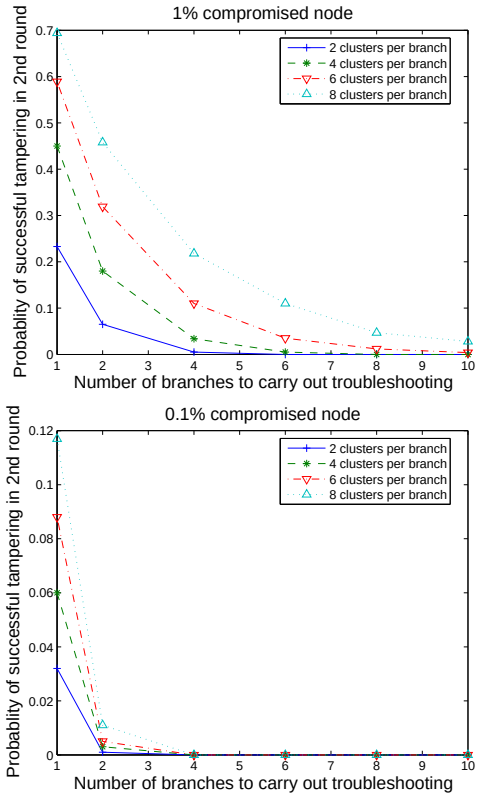

Figure 6: The probability that attackers may successfully tamper with the entry value in the second round.

ing. By querying multiple branches in the second round, the troubleshooter is able to obtain the actual entry value as long as one of the branches does not encounter any compromised node. We use our simulation data to calculate the probability that attackers may successfully tamper with the entry values returned by all branches in the second round, as shown in Figure 6.

Our approach efficiently reduces the threat of second-round tampering for both cases. When there are 0.1% compromised nodes, querying 4 branches in the second round reduces this threat to nearly zero.

12. RELATED WORK

There is much related work in the area of anonymization. The random walk approach is also used in FreeNet [7] and Crowds [23]. FreeNet is a distributed anonymous information storage and retrieval system. Crowds provides anonymous web transactions. Other anonymization systems are based on Chaum's *mixes* [5], which serve as proxies to provide sender-receiver unlinkability through traffic mixing. Onion routing [14] extends the mixes with layers of onion-style pre-encryptions. Tarzan [12] 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, 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 contribute false and potentially harmful information about their configurations.

Homomorphic encryption is also used in [4] to allow a community of users to compute a public aggregate of their data without exposing individual users' data. Similarly, the well known secure multiparty sum protocol enables aggregation without revealing individual private contributions. However, these protocols rely on a public bulletin board and a beacon for random bits, and 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. We combine those two techniques to address both the passive and the active attacks efficiently, and we extend them to support counting the number of distinct values in a set, as well as revealing the most popular value, while keeping the individual contributions private.

Our problem of privacy-preserving parameter aggregation shares much similarity to the problem of secure and privacy-preserving voting [13, 2, 9, 8] with a few differences. 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, privacy of application ownership is compromised. Third, most voting scenarios involve a fixed, limited number of voting chances, while our troubleshooting problem does not. Finally, the scaling requirements of FTN are different from those of national elections: while national elections have at most a few dozen ballot items, and turnaround times of up to an hour are perfectly acceptable, FTN needs to accommodate several thousand ballot items and turnaround times need to be minimized.

As an alternative to full-blown homomorphic encryption, the authors of [17] present a voting system based on cryptographic counters that only support a restricted set of encrypted increment and decrement operations. Although the concept of cryptographic counters is potentially useful in FTN, the scheme given in [17] is not a good fit for FTN because of the differing scaling requirements. In particular, the use of an encryption function based on quadratic residuosity and the need for L rounds of communication with a public bulletin board (where L is the number of participants) mean that the bandwidth usage of this scheme exceeds that of our elliptic curve based scheme. Likewise, recent work of Kissner and Song [18] enables private computation of a very general class of set operations, but does so at the cost of sacrificing the bandwidth savings afforded by elliptic curves.

Wagner in [26] addresses the problem of compromised nodes in the context of sensor networks, and describes how resilient aggregation techniques can be used to limit the amount of damage a compromised sensor can inflict upon the aggregate results of the network. Using resilient aggregation in FTN may help when the amount of redundancy within peers' data contributions is large. The main difficulty in applying these techniques directly is that they assume a trusted base station is available to compute the resilient data aggregation function; this is infeasible in the peer-to-peer FTN context where data contributions must be kept confidential from the other participants.

The authors of SIA [22] also presented a set of techniques for secure information aggregation in sensor networks. The

integrity of information aggregation is achieved essentially through authentication which is identity-revealing. In FTN, we cannot do the same because of privacy concerns.

13. CONCLUSIONS

In this paper, we tackle the key security challenges in Friends Troubleshooting Network: preserving the privacy of aggregating peer configuration data and ensuring the integrity of troubleshooting results.

To guarantee data privacy, we apply the asymptotically optimal homomorphic encryption scheme from [9] and tailor it to scale with the FTN scenario. Our design has the novel property that shares of the secret key are assembled in parallel with the encrypted data. Although the use of homomorphic encryption adds some computational complexity, the only additional user-visible cost of the protocol is bandwidth, which only has to be paid by the minority of participants who are routing nodes or keyholders. The computational resources required by our scheme are practical and represent realistic commitments for a troubleshooting network in which one responds to direct queries from friends.

To ensure integrity, we combine the selective use of zero knowledge proofs together with a branching solution where multiple branches are taken to gather the configuration data using real-world friends network topology. We find that when the percentage of compromised nodes is moderate or small (e.g. 1% or less), our approach can effectively reduce the risk of malicious data injection attacks to nearly zero.

14. REFERENCES

[1] R. Agrawal and R. Srikant. Privacy Preserving Data Mining. In *Proceedings of SIGMOD*, 2000.

[2] J. Benaloh, *Verifiable Secret Ballot Elections*. Ph.D thesis, Yale University, 1987.

[3] J. Benaloh and M. Yung. Distributing the power of a government to enhance the privacy of voters. In *Proc. 5th ACM Symposium on Principles of Distributed Computing (PODC '86)*, pp. 52–62, New York, 1986.

[4] J. Canny. Collaborative Filtering with Privacy. In *IEEE Security and Privacy*, 2002.

[5] D. Chaum. Untraceable electronic mail, return addesses, and digital pseudonyms. *Communications of the ACM*, 24(**2**), pp. 84–88, 1981.

[6] D. Chaum and T. Pedersen. Wallet databases with observers. In *Advances in Cryptology — Crypto '92*, LNCS **740**, pp. 89–105, 1993.

[7] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system. In *Proc. International Workshop on Design Issues in Anonymity and Unobservability*, 2001.

[8] L. Coney, J. L. Hall, P. L. Vora, D. Wagner. Towards a Privacy Measurement Criterion for Voting Systems. Poster Paper, National Conference on Digital Government Research, May 2005.

[9] R. Cramer, R. Gennaro, and B. Schoenmakers. A secure and optimally efficient multi-authority election scheme. In *Advances in Cryptology — Eurocrypt '97*, LNCS **1233**, pp. 103–118, 1997.

[10] R. Cramer, I. Damgård, and B. Schoenmakers. Proofs of partial knowledge and simplified design of witness hiding protocols. In *Advances in Cryptology — Crypto '94*, LNCS **839**, pp. 174–187, 1994.

[11] A. Fiat and A. Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Advances in Cryptology — Crypto '86*, LNCS **263**, pp. 186–194, 1987.

[12] M. J. Freedman, E. Sit, J. Gates, and R. Morris. Introducing Tarzan, a Peer-to-Peer Anonymizing Network Layer. In *IPTPS*, 2002.

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

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

[15] Q. Huang, H. J. Wang, and N. Borisov. Privacy-Preserving Friends Troubleshooting Network. ISOC NDSS 2005, San Diego, CA.

[16] M. Jakobsson, A. Juels, and R. Rivest. Making Mix Nets Robust for Electronic Voting by Randomized Partial Checking. In *Proceedings of the 11th USENIX Security Symposium*, pp. 339-353, 2004.

[17] J. Katz, S. Myers, and R. Ostrovsky. Cryptographic Counters and Applications to Electronic Voting. In *Advances in Cryptology — Eurocrypt 2001*, LNCS **2045**, pp. 78–92, 2001.

[18] L. Kissner and D. Song. Privacy-Preserving Set Operations. In *Advances in Cryptology — Crypto 2005*, LNCS **3621**, 2005.

[19] N. Koblitz. Elliptic curve cryptosystems. *Math. Comp.* **48** (1987), pp. 203–209.

[20] A. Lenstra and E. Verheul. Selecting Cryptographic Key Sizes. *J. Cryptology* **14** (2001), no. 4, pp. 255-293.

[21] T. Pedersen. A threshold cryptosystem without a trusted party. In *Advances in Cryptology — Eurocrypt '91*, LNCS **547**, pp. 522-526, 1991.

[22] B. Przydatek, D. Song, and A. Perrig. SIA: Secure Information Aggregation in Sensor Networks. In *Proceedings of ACM SenSys*, Nov 2003.

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

[24] A. Shamir. How to share a secret. *Comm. ACM* **22** (1979), no. 11, pp. 612–613.

[25] C. E. Shannon. A Mathematical Theory Of Communication. *Bell System Tech. J.* **27** (1948), pp. 379-423, 623-656.

[26] D. Wagner. Resilient Aggregation in Sensor Networks. In *ACM Workshop on Security of Ad Hoc and Sensor Networks* (SASN '04), Oct. 2004.

[27] H. J. Wang, Y. C. Hu, C. Yuan, Z. Zhang, and Y. M. Wang. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *Proceedings of the 3rd International Workshop on Peer-to-Peer Systems (IPTPS)*, 2004.

[28] H. J. Wang, J. Platt, Y. Chen, R. Zhang, and Y. M. Wang. Automatic Misconfiguration Troubleshooting with PeerPressure. In *Proceedings of OSDI* 2004.