PeerPressure: A Statistical Method for Automatic Misconfiguration Troubleshooting

Helen J. Wang
John Platt
Yu Chen
Ruyun Zhang
Yi-Min Wang

Microsoft Research

November 2003

Technical Report
MSR-TR-2003-80

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

PeerPressure: A Statistical Method for Automatic Misconfiguration Troubleshooting

Helen J. Wang, John Platt, Yu Chen, Ruyun Zhang, Yi-Min Wang
Microsoft Research

***Abstract*— Technical support contributes 17% of the total cost of ownership of today's desktop PCs [20]. An important element of technical support is troubleshooting misconfigured applications. Misconfiguration troubleshooting is particularly challenging, because configuration information is shared and altered by multiple applications.**

In this paper, we present a novel troubleshooting algorithm, *PeerPressure*, which uses statistics from a set of sample machines to diagnose the root-cause misconfigurations on a sick machine. This is in contrast with methods that require manual identification on a healthy machine for diagnosing misconfigurations [24]. The elimination of this manual operation makes a significant step towards automated misconfiguration troubleshooting.

In PeerPressure, we introduce a ranking metric for misconfiguration candidates. This metric is based on empirical Bayesian estimation . We have developed a PeerPressure troubleshooting system and used a database of 87 machine configuration snapshots to evaluate its performance. With 20 real-world troubleshooting cases, PeerPressure can effectively pinpoint the root-cause misconfigurations for 12 of them. For the remaining ones, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude.

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 [18][20] have shown that technical support contributes 17% of the total cost of ownership of today's desktop PCs [20]. A large amount of technical support time is spent on troubleshooting.

Many troubleshooting cases are due to misconfigurations. This misconfiguration is often caused by data that is in shared persistent stores such as Windows registry and Unix resource files. Such stores may serve many purposes. They include system-wide resources that are naturally shared by all applications (e.g., the file system). They allow applications installed at different times to discover and integrate with each other. They enable users to customize default handlers or appearances of existing applications. They allow individual applications to register with system services to reuse base functionalities. They permit individual components to register with host applications that provide an extensibility mechanism (e.g., toolbars in browsers).

Maintaining healthy configurations of a computer platform with a large installed base and numerous third-party software packages has been recognized as a daunting task [13]. The considerable number of possible configurations and the difficulty in specifying the "golden state" [21], the perfect configuration, have made the problem appear to be intractable.

In this paper, we address the problem of misconfiguration troubleshooting. There are two essential goals in designing such a troubleshooting system:

1) Troubleshooting effectiveness: the system should effectively identify a *small* set of sick configuration candidates with a short response time;
2) Automation: the system should minimize the number of manual steps and the number of users involved.

To diagnose misconfigurations of an application on a sick machine, it is natural to find a healthy machine to compare against [24]. Then, the configurations that differ between the healthy and the sick are misconfiguration suspects. However, it is difficult to identify a healthy machine *automatically*. Involving the user in confirming the correct application behavior seems unavoidable [1].

We can avoid extensive manual identification work by observing that *the golden state is in the mass*. In other words, an application functions correctly on *most* of machines, therefore we can use the statistics from a large enough sample set as the "statistical golden state". The statistical golden state can be combined with Bayesian statistics to identify anomalous misconfigurations on sick machines. Then, the misconfigurations can be corrected

[1]Different users may even have different views on what the correct application behaviors are.

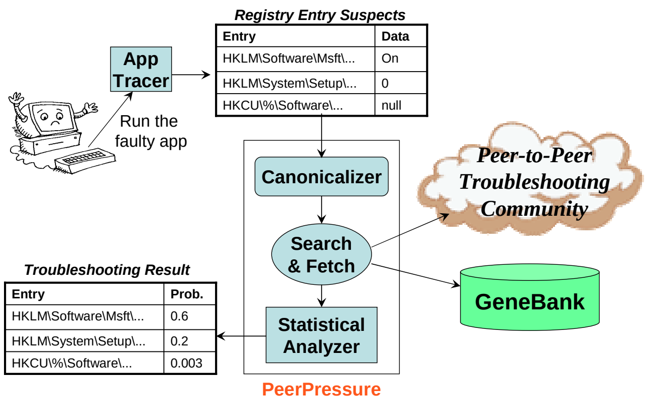
by comforming to the majority of the samples. We name this statistical troubleshooting method *PeerPressure*.

We have prototyped a PeerPressure troubleshooting system which carries out the PeerPressure algorithm using samples from a database of 87 real-usage machine configuration snapshots. And we have evaluated the system with 20 real-world troubleshooting cases. PeerPressure can effectively pinpoint the root-cause misconfigurations for 12 of the cases. For the remaining ones, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude. These results have demonstrated PeerPressure as a promising troubleshooting method.

To simplify our presentation, we will focus our discussion on a particular type of important configuration data, the Windows Registry [19], which provides hierarchical persistent storage for named, typed entries. The principles and techniques are directly applicable to other types of configuration stores such as files and other platforms such as Unix.

We will first give an overview on the architecture and operations of our PeerPressure troubleshooting system in Section II. In Section III, we detail the formulation and the analysis of the PeerPressure algorithm. We discuss our prototype implementation in Section IV. Then, we present our empirical results in Section V. We compare and constrast our work with the related work in Section VI, address the future work in Section VII, and finally conclude in Section VIII.

II. PeerPressure Troubleshooting System Architecture



Registry Entry Suspects

Entry	Data
HKLM\Software\Msft\...	On
HKLM\System\Setup\...	0
HKCU\%\Software\...	null

Troubleshooting Result

Entry	Prob.
HKLM\Software\Msft\...	0.6
HKLM\System\Setup\...	0.2
HKCU\%\Software\...	0.003

Fig. 1. PeerPressure Troubleshooting System Architecture and its Operations

Figure 1 illustrates the architecture and the operations of a PeerPressure troubleshooting system. A troubleshooting user first expresses the symptom of the sick machine through the use of "App Tracer". "App Tracer" records the registry entries that are used as input to the failed application execution. We term these misconfiguration candidates *suspects*. Then, the user feeds the suspects into the PeerPressure troubleshooter which has three modules: a canonicalizer, a searcher/fetcher, and a statistical analyzer. 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 "USERNAME" and "MACHINENAME", respectively. Then, PeerPressure searches for a sample set of machines that run the same application. The search can be performed over a "GeneBank" database that consists of a large number of machine configuration snapshots or through a peer-to-peer troubleshooting community. (In this paper, we base our discussions on the GeneBank database approach. For the peer-to-peer approach, we refer interested readers to [23].) Next, PeerPressure fetches the respective values of the canonicalized suspects from the sample set machines. The statistical analyzer then performs statistical analysis, calculates the probability for each suspect to be sick, and outputs a ranking report based on the sick probability. Finally, PeerPressure conducts trial-and-error fixing, by stepping down the ranking report and replacing the possibly sick value with the most popular value from the sample set. The fixing step interacts with the user to determine whether the sickness is cured. This last step is not shown in the figure; and we will not further address it for the rest of the paper.

As careful readers can see, there are still some manual steps involved. The first one is that the user must run the sick application to record the suspects. The second one is that the user is involved in determining whether the sickness is cured for the last step. We argue that these manual steps are difficult to eliminate because only the user can recognize the sickness, and therefore has to be in the loop for those steps. Nonetheless, these manual steps only involve the troubleshooting user, and not any second parties.

III. The PeerPressure Algorithm

In this section, we first illustrate the intuition and objectives for calculating the probability of a suspect being sick. Then, we derive the sick probability formula. At last, through our analysis, we show that our formulation achieves the objectives.

	Name	Suspect	Sample 1	Sample 2	Sample 3	Sample 4	Sample 5
e1	.jpg/contentType	*image/jpeg*	image/jpeg	image/jpeg	image/jpeg	image/jpeg	image/jpeg
e2	.htc/contentType	not exist	text/x-comp	text/x-comp	text/x-comp	text/x-comp	text/x-comp
e3	url-visited	yahoo	hotmail	nytimes	SFGate	google	friendster

TABLE II

INTUITION BEHIND PEERPRESSURE SICK PROBABILITY FORMULATION

N	Sample set size
t	Suspect set size
i	The index for the suspect (from 1 to t)
V	The value of a suspect
c	The number of possible sample values for a suspect
m	The number of samples that match the suspect value
$P(S)$	The prior probability that a suspect is sick
$P(H)$	$1-P(S)$
$P(S\|V)$	The probability that a suspect is sick given its value
$P(V\|S)$	The probability that a sick suspect has value V_i

TABLE I

NOTATION

A. Intuition and Objectives

We use an example to illustrate the intuition and objectives of formulating the sick probability calculation for each suspect. Table II shows three suspects ($e1$,$e2$,$e3$) and their respective values from a sample set of machine configuration snapshots from the GeneBank. A cursory examination of the sample set suggests that $e1$ is probably healthy and $e2$ is more likely to be sick than $e3$. The suspect $e2$ is more likely to be sick because all samples have the same value, while the suspect value is different.

In fact, we have seen two types of state in canonicalized configuration entries: (I) application configuration states such as $e1$ and $e2$, (II) operational states such as timestamps, usage counts, caches, seeds for random number generators, window positions, and MRU (Most Recently Used)-related information. For troubleshooting configuration failures, we are mostly concerned with type I entries. Type II entries constitute the "natural biological diversity" among machines and are less likely to be root causes of configuration failures. In our example, $e3$ belongs to category II.

Therefore, the objective for the sick probability formulation is not only to capture the anomaly from the golden mass, but also to weed out the operational state false positives.

B. Formulation

Table I summarizes our notation.

To estimate whether a suspect is sick, we need to estimate $P(S|V)$, the probability that a suspect is sick given its value V. We estimate this probability for all suspects independently. In the derivation below, let us consider only one suspect i: all parameters are implicitly indexed by i.

According to Bayes rule [10], we have:

$$P(S|V) = \frac{P(V|S)P(S)}{P(V|S)P(S)+P(V|H)P(H)} \qquad (1)$$

We need to estimate each of the terms on the right-hand-side of Equation (1). We first assume that there is only one sick entry amongst the suspects (leaving the multiple sick entry case for future work). Before we observe any values, the prior probabilities of a suspect being sick and healthy are

$$P(S) = \frac{1}{t} \qquad P(H) = 1-\frac{1}{t}$$

where t is the number of possible suspects.

We do not have an extensive training set of sick suspects. Therefore, we make an assumption that a sick entry has all possible values with equal probability:

$$P(V|S) = \frac{1}{c}$$

where c is the cardinality of the suspect entry, the total number of values that entry can take. Note that we compute c by counting the number of unique values for that entry in the sample set (including "no entry", if that occurs), then adding one to account for all entries that do not occur in the sample set.

For $P(V|H)$, we leverage the observation from a sample set of machine configurations from the GeneBank.

Let m denote the number of samples matching V, and N, the size of the sample set. If we assume that $P(V|H)$ is estimated via maximum likelihood, we get the estimate

$$P(V|H) = \frac{m}{N} \tag{2}$$

$$P(S|V) = \frac{N}{N + cm(t-1)} \tag{3}$$

However, maximum likelihood has undesirable properties when the amount of sample data is limited. For example, when there are no matching values to V in the sample set, then $m = 0$ and $P(S|V) = 1$, which expresses complete certainty that is unjustified. For example, in Table II, maximum likelihood would claim that $e2$ and $e3$ are both sick with complete and equal confidence.

Bayesian estimation [10] of probabilities is more appropriate for the situation of small sample size N, such as our GeneBank scenario. Bayesian estimation uses a prior over $P(V|H)$, before the sample set is examined. The estimation then uses the posterior estimate of $P(V|H)$ after the sample set is examined. Therefore, $P(V|H)$ is never 0 or 1.

We first assume that $P(V|H)$ is multinomial over all possible values V. The multinomial has parameters p_j. Each p_j is the probability that the value V_j is used. The p_j sum to 1.

Now, the p_j have prior and posterior values which we draw from a Dirichlet distribution [10]. Dirichlet distributions are a natural prior for multinomials, because they are *conjugate* to multinomials. That is, combining observations from a multinomial with a prior Dirichlet yields a posterior Dirichlet. Thus, the Dirichlet distribution is mathematically convenient.

Dirichlet distributions are completely characterized by a count vector n_j, which corresponds to the number of possibe counts for each value V_j. These counts do not need to reflect real observations: as we'll see below, we can count phantom data, also.

To perform Bayesian estimation of $P(V|H)$, we start with a prior set of counts n'_j that reflect our prior belief about the likelihood of various values V_j. We then observe our N samples of values for this suspect, collecting counts m_j for the different values. The mean of the posterior Dirichlet yields the posterior estimate $P(V_j|H)$ [10]

$$P(V_j|H) = \frac{m_j + n'_j}{N + \sum_j n'_j} \tag{4}$$

We only need to estimate the $P(V_j|H)$ for the value that actually occurs in the suspect entry. Therefore, we can replace m_j with m, the number of samples that matches the suspect entry. Furthermore, we can assume that all values V_j have the same *a priori* probability (before looking at the sample set). Thus, n'_j can be replaced with some value n and the sum $\sum_j n'_j$ can be replaced with cn. Combining these assumptions with Equations (4) and (1) yields

$$P(S|V) = \frac{N + cn}{N + cnt + cm(t-1)} \tag{5}$$

The parameter n is proportional to the number of observations that are required to overwhelm the prior and to move the estimated $P(V|H)$ probabilities away from $E(p_j) = 1/c$. In other words, the higher the n is, the less confidence we have for the knowledge obtained from the GeneBank. n indicates the strength of the prior. A higher n leads to a stronger prior, which requires more evidence (observations N) to change the posterior. Notice that Equation (5) never predicts a sick probability of 0 or 1, even if m is 0 or N.

We choose $n = 1$ for our prior, which is equivalent to a flat prior: all multinomial values p_j are equally likely *a priori*. This is known as an "uninformative" prior.

C. Asymptotic analysis

To show that our Bayesian probability estimates in Equation (5) produce sensible results, we illustrate the asymptotic behavior of the estimates in various cases.

Given a suspect set of size t, there are four variables that affect the sick probability ranking for the suspects, namely, the number of matches m, the Dirichlet prior strength n, the sample set size N, and the cardinality c. Please note that N can vary among the suspects because of the canonicalized entries. For example, for a user-specific canonicalized entry, the number of samples is the number of users rather than the number of machines in the GeneBank; and a machine can have multiple users. Now, we analyze on how each of these parameters affects the sick probability and whether the trend agrees with our objectives (see Section III-A).

Fixing N, c, and n, as the number of matches m increases, the sick probability decreases, as desired:

$$\lim_{m\to\infty} P(S|V) = 0$$

Fixing N, c, and m, as the prior strength n increases, we have

$$\lim_{n\to\infty} P(S|V) = \frac{1}{t} = P(S)$$

This means that conducting a statistical analysis over such a sample set is useless in this case. This makes sense, because when n reaches infinity, the prior has infinite strength, and therefore observations offer no additional knowledge.

For understanding the influence of N, we assume that as N grows, m also grows as fN, for some fraction f between 0 and 1. Therefore,

$$\lim_{N \to \infty} P(S|V) = \frac{1}{1 + cf(t-1)}$$

Notice in the infinite data limit, the prior is completely “washed out”, and the higher c, f, or t is, the less likely an entry is to be sick. We also have, for $N = m = 0$,

$$\lim_{N \to 0} P(S|V) = \frac{1}{t} = P(S)$$

This is also accurate. Since when $N = 0$, we are unable to make any observations. So, the suspect set is the only factor that determines the sick probability.

To illustrate the impact of the cardinality c, we first note that $c \to \infty$ implies $N \to \infty$. So, applying the analysis for N above, we have

$$\lim_{c \to \infty, N \to \infty} P(S|V) = \lim_{c \to \infty} \frac{1}{1 + cf(t-1)} = 0$$

This is desirable because when c is large, it represents a higher level of “biological diversity”, and therefore, being different is less likely due to some sickness.

Now, we examine the case of operational state where $m = 0$ most likely, we have

$$P(S|V) = \frac{N + cn}{N + tcn}$$

Fixing N, the sick probability decreases with increased cardinality when there are no matches because the derivative of $P(S|V)$ with respective to c is negative when $t > 1$; and when $t = 1$, $P(S|V) = 1$ as desired. Therefore, for our example in Table II, Formula 5 will rank $e2$ sicker than $e3$, as desired.

In summary, our analysis demonstrates that Formula 5 achieves our objective of capturing anomalies and weeding out operational state false positives. Later, in Section V, we further demonstrate through real-world troubleshooting cases that our PeerPressure algorithm is indeed effective.

IV. Implementation of PeerPressure Prototype

We have prototyped the PeerPressure troubleshooting system as shown in Figure 1. We have created a GeneBank database using Microsoft SQL Server 2000 [7], which now contains real-usage registry snapshots from 87 Windows XP desktop PCs. We have implemented the PeerPressure troubleshooter in C# [16], which issues queries to the GeneBank to fetch the sample values and carries out the sick probability calculation (Section III). We use a set of heuristics for canonicalizating user-specific, machine-specific configuration entries in the suspect set. One obstacle we encountered during our prototyping is that values for a specific registry entry across different machines are the same but with different representations. For example, 1, “#1”, “1” all represent the same value. Nonetheless, the first one is an integer and the latter two are different string representations. Such inconsistent representations of the same data affect all parameter values needed by the sick probability calculation. We use heuristics to unify the different representations of the same data value. We call this procedure “data sanitization” for future reference. For example, one such heuristic is to find all entries that have more than one types. (Registry entries contain a “type” field). For a registry entry that have both numeric-typed and string-typed values among different registry snapshots, all string values are converted into numbers.

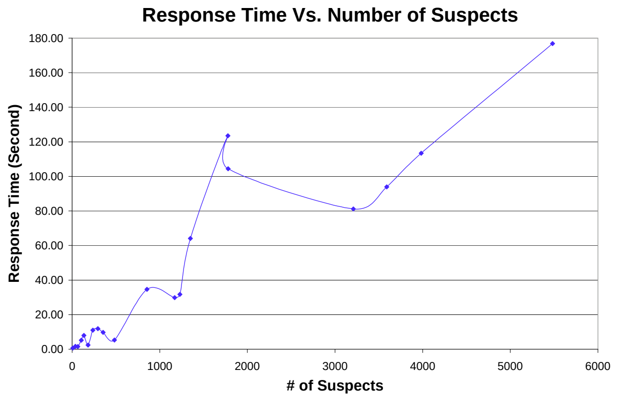


Fig. 2. Response Time Vs. Number of Suspects for 20 real-world troubleshooting cases.

Our PeerPressure troubleshooter, although unoptimized in its present form, is already fast. In average, it takes less than 45 seconds to return a root-cause ranking report for suspect sets of thousands of entries. The response time generally grows with the number of suspects because we issue one query per suspect entry. Figure 2 shows the relationship between the response time and the number of suspects for the 20 troubleshooting cases under study . With aggressive database query batching, we anticipate that the response time can be greatly improved.

Maximum registry size	333,193
Minimum registry size	77,517
Average registry size	198,376
Median registry size	198,608
Distinct canonicalized registry entries in GeneBank	1,476,665
Common canonicalized registry entries	43,913
Distinct entries data-sanitized	1,820,706

TABLE III
REGISTRY CHARACTERISTICS

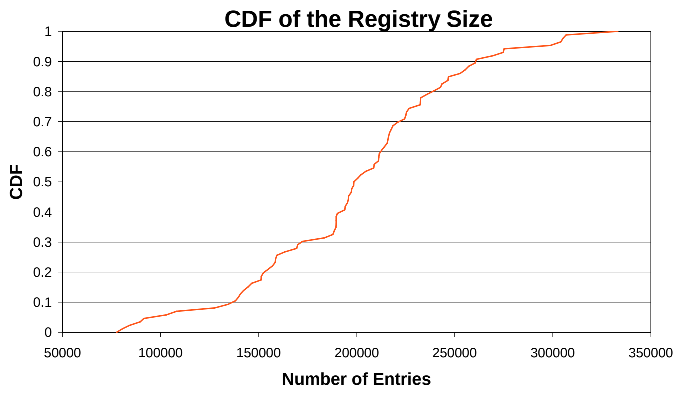


Fig. 3. Registry Size Distribution

V. Troubleshooting Effectiveness of the PeerPressure Troubleshooter

In this section, we evaluate the troubleshooting effectiveness of the PeerPressure prototype on the 20 real-world troubleshooting cases. We first take a peek of the registry characteristics based on the registry snapshots from our GeneBank repository, then we present and analyze our troubleshooting results.

A. A Peek of Registry Characteristics

Windows registry contains most of the configuration data for a desktop PC. Table III summarizes some registry characteristics manifested from the GeneBank. The sheer volume of configuration data is daunting. Figure 3 shows the registry size distribution among the registry snapshots in the GeneBank. Registry size ranges from 77,517 to 333,193 entries. The median is 198,608 entires. The total number of distinct canonicalized entries in the GeneBank is as large as 1,476,665, which represents the total number of "genes" contained in the small world of 87 machines. Across all the machines, there are 43,913 common canonicalized entries. With our canonicalization heuristics, an average of 68,126 entries from each registry snapshot are canonicalized. With our data sanitization (see Section IV) heuristics, we have sanitized as many as 1,820,706 entries in the GeneBank.

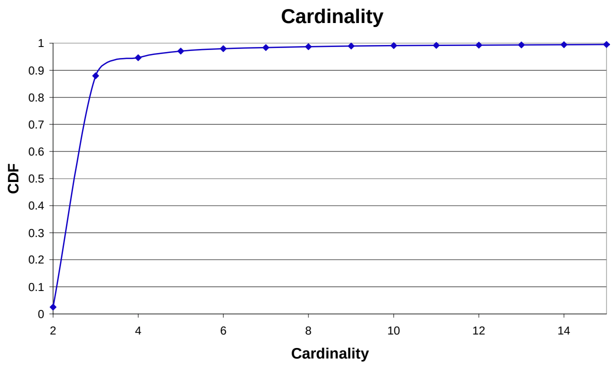


Fig. 4. Cardinality Distribution

Cardinality is an essential parameter of our PeerPressure formulation (Section III). Because the GeneBank may not contain all possible "genes" (entry values), we count all values that are unknown to the GeneBank as a single value *unknown*. This effectively increments the observed cardinality from the GeneBank by one. Therefore, any entry from the GeneBank has a cardinality of at least 2; and entries that do not exist in the GeneBank have a cardinality of 1. Also, some entries may not exist on some sample machines. For such cases, these entries have the value *no entry*. Figure 4 shows the distribution of the cardinality for all canonicalized entries in the GeneBank. 87% of the registry entries have a cardinality of 2, 94% no more than 3, and 97% no more than 4.

B. PeerPressure Performance with Real-World Troubleshooting Cases

Now, we present our empirical troubleshooting results for PeerPressure.

We use the 20 cases listed in Table IV for our experiments. They were all real-world failures that troubled some users. And we have the knowledge of their root-cause misconfiguration a priori. Therefore, we use the ranking of the root-cause entry as our evaluation metric. To allow parameterized experiments, we reproduced these failures on a real-usage desktop using configuration user interface (e.g., Control Panel applets) to inject the failures whenever possible, and using direct editing of the Registry for the remaining cases. Then, we used "App Tracer" to get the suspects (see Section II). Finally, we ran PeerPressure to produce the ranking reports.

1) **Root Cause Ranking***:* For each troubleshooting case, Table V shows the ranking of the root-cause entry, the number of ties, the number of suspects, the cardinality of the root-cause entry, the number of samples matching the suspect's root-cause entry value, and the

ID	Name	Description
1	Systems Restore	No available checkpoints are displayed because the calendar control object cannot be started due to a missing Registry entry.
2	JPG	Right-clicking on a JPG image and choosing the Send To → Mail Recipient option no longer offer the resize option dialog box due to a missing Registry entry.
3	Outlook	User is always asked upon exiting Outlook whether she wants to permanently delete all emails in the Deleted Items folder, due to a hard-to-find setting.
4	IE Passwords	Internet Explorer (IE) browser no longer offers to automatically save passwords; the option to re-enable the feature is difficult to find.
5	Media Player	Windows Media Player "Open Url" function would fail if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.
6	IM	MSN Instant Messenger (IM) would significantly slow down if the firewall client is disabled on a corporate desktop.
7	IE Proxy	IE on a machine with a corporate proxy setting would fail when the machine is connected to a home network.
8	IE Offline	IE "Work Offline" option may be automatically turned on without user knowledge; user would then be presented with a cached offline page instead of the default start page when launching IE.
9	Taskbar	IE windows would be unexpectedly grouped under the Windows Explorer taskbar group, due to the addition of a Registry entry.
10	Network Connections	Control Panel → Network Connections showed nothing, due to a missing Registry key.
11	Folder Double-Clicking	Double-clicking any folder in the right pane of Windows Explorer would incorrectly bring up the "Search Results" window.
12	Outlook Express	Microsoft Outlook could not be started because the Outlook Express installation appeared to be missing, due to a missing Registry key.
13	Cannot Start Executables	Double-clicking any EXE file would not launch the application.
14	Shortcut	Double-clicking any shortcut would not launch the application.
15	IE Menu Bar	IE menu bar disappeared due to a corrupted Registry key name.
16	IE Favorites	IE used the "unknown file type icon" for some of the links in the Favorites.
17	Sound Problem	Warning sound was missing when an invalid command was typed into Start-¿Run.
18	IE New Window	Right-clicking a link inside IE and choosing "Open in New Window" would show nothing.
19	Yahoo Toolbar	Yahoo Companion per-user installation affects all users.
20	Media Player	Windows Media Player "Open URL" function would fail if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.

TABLE IV
20 REAL-WORLD TROUBLESHOOTING CASES USED FOR PEERPRESSURE EVALUATION

number of samples. The non-zero values for the "# of Matches" column indicate that the GeneBank contains registry snapshots with the same sickness. Nonetheless, our assumption that the golden state is in the mass is still correct, since there are indeed only very small percentage of the sick machines in the GeneBank.

As we can see from the table, the number of suspects is large: ranging from 8 to 26,308, with a median of 1,171, and an average of 2,506. Therefore, PeerPressure is an indispensible step of troubleshooting since sieving through these large suspect sets for root-cause entries is like finding a needle in a haystack.

For 12 out of the 20 cases, PeerPressure ranks the root-cause entry as number one without any ties. For the remaining cases, PeerPressure narrows down the root-cause candidates in the suspect set by three orders of magnitude for most cases. There is only one case, case 19, which our GeneBank cannot help because only two machines in the GeneBank have the sick application and they happen to have the same sick values as well.

2) **The Causes of False Positives***:* Now, we give an analysis on the causes of false positives. The sick probability metric essentially ranks on the conformance level of a suspect entry to the samples from the GeneBank. The more conforming a suspect is in comparing with other suspects, the larger its rank number is (i.e., the more healthy the suspect is).

One source of false positives is due to the nature of the root-cause entry. If the root-cause entry has a large cardinality, it likely receives a larger rank number based on our sick probability formula in Section III. Case 20 falls into this category. The root-cause entry for Case 20 has a high cardinality of 65 while the rest of the cases have low cardinalities (Table V).

The nature of the root-cause entry is only one factor. The ranking also depends on how the root-cause entry relates to other entries in the suspect set. A highly customized machine likely produces more noise, since the unique customizations can be even less conforming than a sick entry value. Case 11, 12, and 16 fall in this category.

Lastly, GeneBank is not pristine. The non-zero values in Column "# of Matches" in Table IV indicates the number of machines in the GeneBank that have the same sickness. This affected the ranking of Case 2, 6, and 10.

3) **The Impact of the Sample Set Size***:* It is intuitive that the larger the sample set is, and better the root-cause ranking will be. However, our evaluation results indicate that this is not *entirely* true.

We have experimented with sample sets of size 5, 10, 20, 30, 50, and 87. For each sample set size N, we pick N samples from the GeneBank randomly for 5 times, then we average the root-cause ranking of the random sample sets. Table VI shows root-cause ranking trend for various sample set sizes. The average number of ties for each sample set size is indicated in the paretheses. For the first three cases in the table, the root-cause ranking is perfect regardless of the sample set size. One reason is that there is a strong conformance of values in the GeneBank for the root-cause entry (e.g., all samples take the same value). And such strong conformance manifests in any subset of the GeneBank samples. In addition, no other suspects become noise when the sample set is small.

The cases belonging to the middle portion of Table VI do not show a clear trend as a function of the sample set size. For Case 20, the root-cause entry has a scattered value distribution and a high cardinality of 65. So, drawing any subset of the samples reflects the same value diversity, and therefore the ranking does not improve with larger sample set. For the other cases, although there is strong comformance in their value distributions, their rankings are affected by other entries in the suspect sets.

For the third category of the cases in the bottom part of Table VI, the root-cause ranking improves with larger sample set. For the first 4 cases, they have near-perfect root-cause ranking. Nonetheless, the number of ties decreases quickly as the sample set size increases. For most of the cases belonging to this category, we can see that the GeneBank has polluted entries according to the "# of Matches" column in Table V. In this situation, enlarging the sample set reduces the impact of the polluted entries and therefore contributes to the decreasing trend of the rankings.

4) **Sick Machine Sensitivity Evaluation***:* So far, we have only presented results from one sick machine's vantage point. In fact, the troubleshooting results do depend on how uniquely the sick machine is customized and configured. To understand how our results vary with different sick machines, we have picked three real-usage machines that belong to different users, and evaluated the sick machine sensitivity with 5 cases. Table VII shows that the troubleshooting results on these sick machines are mostly consistent. In some cases, such as Case 6, there is this phenomenon that a larger suspect set leads to better ranking rather than introducing more noise as one may expect. This is simply because the larger suspect set on one machine is not necessarily a superset of the smaller suspect set on the other machine.

Cases	Machine 1	Machine 2	Machine 3
2. JPG	16 (0) / 1779	32 (2) / 1272	14 (0) / 1272
5. Media Player	1 (0) / 182	1(0) / 566	1 (0) / 1657
6. IM	1(0) / 2789	12(0) / 1777	12 (0) / 2017
14. ShortCut	1(0) / 105	1(0) / 84	1 (0) / 64
16. IE Favoriates	1 (0) / 302	2(0) / 3209	1 (3) / 1908

TABLE VII

SICK MACHINE SENSITIVITY EVALUATION. EACH ENTRY HAS THE FORMAT OF ROOTCAUSERANKING (NUMBEROFTIES) / NUMBEROFSUSPECTS.

VI. RELATED WORK

There are two general approachs in system management: the white-box [4][3][6][15][11][22] and the black-box approach [24]. In the former, languages and tools are designed to allow developers or system administrators to specify "rules" of proper system behavior and configurations for monitoring, and "actions" to correct any detected deviation. The biggest challenge for the white-box approach is in the accuracy and the completeness of the rule specification.

Strider [24] exemplifies the black-box approach for misconfiguration troubleshooting: problems are diagnosed and corrected in the absence of specification of correct behavior. In Strider, the troubleshooting user first identifies a healthy machine on which the application functions correctly. This can be done by finding a healthy configuration snapshot in the past on the same machine or by finding a different healthy machine. Next, Strider performs configuration state differencing between the sick and the healthy, the difference is then further narrowed down by intersecting with suspects obtained from "App Tracer" (Section II). Finally, Strider uses noise-filtering techniques to further narrow down the root-cause candidate set. Noise-filtering uses a metric called *Inverse Change Frequency* which looks at the change frequency of a registry entry. The more frequent an entry changes, the more likely it is a piece of operational state which is unlikely a root cause.

PeerPressure also takes the general black-box approach. PeerPressure differs from Strider in the following

Case	**Rank**	Ties	# of Suspects	Cardinality	# of Matches	# of Samples
1. System Restore	1	0	1350	3	1	87
2. JPG	16	0	1779	3	5	87
3. Outlook	1	0	37	4	7	566
4. IE Passwords	1	0	135	4	1	566
5. Media Player	1	0	182	6	1	566
6. IM	12	0	1777	4	8	87
7. IE Proxy	1	0	1171	16	0	566
8. IE Offline	1	0	1230	4	1	566
9. Taskbar	1	0	64	4	2	566
10. Network Connections	2	0	354	2	1	87
11. Folder Double-Click	2	1	26308	2	0	87
12. Outlook Express	3	0	482	2	0	87
13. Cannot Start Executables	1	0	237	2	0	87
14. ShortCut	1	0	105	2	0	87
15. IE Menu bar	1	2	3590	2	0	87
16. IE Favoriates	2	0	3209	3	0	87
17. Sound Problem	1	0	8	1	0	566
18. IE New Window	1	0	853	2	0	87
19. Yahoo Tool bar	n/a					
20. MediaPlayer in IE	9	0	5483	65	0	566

TABLE V
ROOT-CAUSE RANKING RESULTS

Case	5 Samples (Ties)	10 (Ties)	20 (Ties)	30 (Ties)	50 (Ties)	87 (Ties)	# of matches
Perfect ranking regardless of the sample set size							
5. Media Player	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1
14. Invalid ShortCut	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
17. Sound Problem	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
Ranking Trend not solely dependent on the sample set size							
10. Network Connections	1.6 (1)	1.4 (0.6)	2 (0.2)	1.4 (0.2)	1.4 (0)	2 (0)	1
20. Media Player in IE	6.2 (0.2)	6.2 (0)	8 (0)	11 (0)	11.2 (0)	9 (0)	0
2. JPG	8.4 (0.2)	13.4 (0.4)	14.6 (0.2)	13 (0.2)	14.2 (0)	16 (0)	5
6. IM	15.6 (1.6)	104 (0.2)	20 (0)	15.4 (0)	14.6 (0)	8 (0)	8
Larger Sample Set improves ranking							
8. IE Offline	1 (0.2)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1
13. Cannot Start Executables	1 (0.4)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0
1. System Restore	1 (0)	1 (0.2)	1 (0.2)	1 (0.2)	1 (0)	1 (0)	1
9. Taskbar	1.6 (5)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	2
3. Outlook	2.2 (0.4)	1.4 (0.8)	1.6 (0.8)	1.4 (0.8)	1 (0)	1 (0)	7
4. IE Passwords	5.8 (8.2)	3.2 (2.4)	3.2 (2.4)	1 (0)	1 (0)	1 (0)	1
7. IE Proxy	3.4 (1.8)	2.2 (0.2)	2 (0.8)	3(3.2)	1(0)	1 (0)	0
15. IE No Menu Bar	6.4 (10.8)	3.2 (3.6)	2.2 (2.6)	1.6 (2.4)	1 (2)	1 (2)	0
16. IE Favorites	18.2 (1)	3.8 (1.8)	3.2 (0.8)	3.8 (0)	2.8 (0)	2 (0)	0
18. IE New Window	7 (0.8)	3.8 (0.8)	2.2 (0)	1.6 (0)	1 (0)	1 (0)	0

TABLE VI
IMPACT OF THE SAMPLE SET SIZE

ways:

1) With statistical analysis, PeerPressure eliminates the manual step of the troubleshooting user identifying a healthy machine. This also eliminates the involvement of any second parties in cross-machine troubleshooting scenarios.
2) PeerPressure generalizes the state-differencing and noise-filtering steps with one step of statistical analysis.
3) Strider uses order ranking which means that the final ordering of suspects are based on the sequence of their usage during application execution. The later the root-cause entry appears during the execution, the more false positives there are. In contrast, PeerPressure is not sensitive to the sequence of suspect entry usage. Nonetheless, the larger the suspect set is, the more likely there are entries which are more unique than the root-cause entry.
4) On the measure of root-cause ranking, PeerPressure's yields better ranking for most of the cases.

Another interesting work that also takes the black-box approach is that of Aguilera et al. [1]. They address the problem of black-box performance debugging for distributed systems. They developed and compared two algorithms for inferring the dominant causal paths. One uses the timing information from RPC messages. The other uses signal processing techniques. The significant finding of this work is that traces gathered with little or no knowledge of application design or message semantics are sufficient to make useful attributions of the sources of system latency. Therefore, their techniques are applicable to almost any distributed systems.

In a recent position paper, Redstone et al. [17] described a vision of an automated problem diagnosis system that automatically captures aspects of a computer's state, behavior, and symptoms necessary to characterize the problem, and matches such information against problem reports stored in a structured database. Redstone's work addresses the troubles with *known* root causes. PeerPressure complements this work with the techniques that identify the root causes of unsolved troubleshooting cases.

The concept of using statistical techniques for problem identification has emerged in several areas in recent years. One way of using statistics is to build a statistical model of healthy machines, and compare a sick machine against the statistical model. PeerPressure falls into this category and is the first to apply Bayesian techniques to the problem of misconfiguration troubleshooting. Other related work in this category [8][12][9] first use statistics to build a correct behavior model which is used to detect anomalies. Then, the number of false positives is minimized as much as possible. Engler et al. [8] use static analysis on the source code to derive likely invariants based on the statistics on some pre-defined rule templates (such as a call to $function\ a()$ must be paired with a call to $function\ b()$). Then, potential bugs are recognized as deviant behaviors from these invariants. Engler et al. have discovered hundreds of bugs in Linux and FreeBSD to date. Later, they further improved the false positive rate in [12]. Forrest et al.'s seminal work on host-based intrusion detection system [9] builds a normal-behaving system call sequence database by observing system calls for various processes. Then, the intrusions with abnormal system call sequence can be caught. Apap et al. [2] designed a host-based intrusion detection system that builds a model of normal Registry behavior through training and showed that anomaly detection against the model can identify malicious activities with relatively high accuracy and low false positive rate.

Another way of using statistics is to correlate the observed service failure with root-cause software component or source code for the purpose of debugging. Liblit et al. [14] uses statistical sampling combined with a number of elimination heuristics to analyze program behaviors. Program failures, such as crashes, are correlated with specific features or even specific variables in a program.

The PinPoint root-cause analysis framework [5] is a a debugger for component based systems. PinPoint identifies individual faulty components that causes service failures in a distributed system. PinPoint uses data clustering analysis on a large number of multi-tier request-response traces that are tagged with perceived success/failure status. The clustering determines the root-cause subset component(s) for the service failures.

VII. Future Work

We have much future work ahead of us. In this paper, we assume that there is only one sick entry among the suspects. However, it is possible that multiple entries contribute to the sickness collectively. We call the process of identifying multiple root-cause entries, *multi-gene* troubleshooting. Determining the number of genes involved in a troubleshooting case as well as formulating the multi-gene sick probability are non-trivial tasks because the sick probability of each entry is no longer independent of one another.

Another open question is GeneBank maintenance. The GeneBank currently has a one-time machine configuration snapshots from 87 volunteers. Without further maintenace, these configuration snapshots will be essentially out-of-date because of numerous software and OS upgrades. Effectively managing the evolving GeneBank is a challenge. Further, we have not yet addressed the privacy issue. The privacy for both the users who contribute their configuration snapshots to the GeneBank and the users who troubleshoot their computers with the GeneBank need to be protected for real deployment. An alternative to the GeneBank approach is to "search and fetch" in a peer-to-peer troubleshooting community (see Section II). Drawing the sample set in a peer-to-peer fashion is essentially treating all computer configuration snapshots from all the peer-to-peer participants as a distributed database that is always up-to-date and requires no maintenance. Nonetheless, the peer-to-peer approach does result in longer search and response time. Further, ensuring the integrity of the troubleshooting result is a challenge in the face of unreliable or malicious peers. We have a proposal for a privacy and integrity-perserving peer-to-peer troubleshooting system. For details, please see [23].

VIII. Conclusions

We have presented PeerPressure, a novel troubleshooting algorithm which uses statistics from a set of sample machines as the golden state to diagnose the root cause misconfigurations on a sick machine. In PeerPressure, we introduce a ranking metric based on Bayesian estimation of the probability of a suspect candidate being sick, given the value of that suspect candidate.

We have developed a PeerPressure troubleshooter and used a database of 87 real-usage machine configuration snapshots to evaluate its performance. With 20 real-world troubleshooting cases, PeerPressure can effectively pinpoint the root-cause misconfigurations for 12 of the cases. For the remaining cases, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude.

In addition to achieving the goal of effective troubleshooting, PeerPressure also makes a significant step towards automation in misconfiguration troubleshooting by using a statistical golden state, rather than manually identifying a single healthy state.

Future work includes multi-gene troubleshooting where there are multiple root-cause entries instead of one, as well as privacy-preservation mechanisms for real deployment.

Acknowledgement

We inherited the "App Tracer" that was implemented by Chad Verbowski for the Strider toolkit [24]. Chad has also given us valuable feedback and discussions on the GeneBank DB schema design as well as on canonicalization heuristics for Registry entries. Emre Kiciman has also contributed to the canonicalization heuristics design. We received much advice from our Database experts on the same floor for the GeneBank design and optimizations. They are Venki Ganti, Zhiyuan Chen, and Nico Bruno. This work also benefited from numerous discussions with Chun Yuan and Zheng Zhang. Zheng Zhang has given us great support and encouragement on this work. We thank everyone for their generous help.

References

[1] Aguilera, M. K., Mogul, J. C., Wiener, J. L., Reynolds, P., and Muthitacharoen, A. Performance Debugging for Distributed Systems of Black Boxes. In *Proceedings of SOSP* (2003).

[2] Apap, F., Honig, A., Hershkop, S., Eskin, E., and Stolfo, S. J. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In *Proceedings of LISA* (1999).

[3] Bailey, E. Maximum RPM, 1997.

[4] Burgess, M. A Site Configuration Engine. In *Computer Systems* (1995).

[5] Chen, M., Kiciman, E., Fratkin, E., Fox, A., and Brewer, E. Pinpoint: Problem Determination in Large, Dynamic, Internet Services. In *Proceedings of International Conference on Dependable Systems and Networks (IPDS Track)* (2002).

[6] Couch, A., and Gilfix, M. It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes. In *Proceedings of LISA* (1999).

[7] Delaney, K. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2001.

[8] Engler, D., Chen, D. Y., Hallem, S., Chou, A., and Chelf, B. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Proceedings of ACM Symposium on Operating Systems Principles (SOSP)* (October 2001).

[9] Forrest, S., Hofmeyr, S. A., Somayaji, A., and Longstaff, T. A. A Sense of Self for UNIX Processes. In *Proceedings of the IEEE Symposium on Research in Security and Privacy* (1996).

[10] Gelman, A., Carlin, J., Stern, H., and Rubin, D. *Bayesian Data Analysis*. Chapman, 1995.

[11] Keller, A., and Ensel, C. An Approach for Managing Service Dependencies with XML and the Resource Description Framework. In *Journal of Network and Systems Management* (June 2002).

[12] Kremenek, T., and Engler, D. Z-Ranking: Using Statistical Analysis to Counter the Impact of Static Analysis Approximations. In *Proceedings of 10th Annual International Static Analysis Symposium* (June 2003).

[13] Larsson, M., and Crnkovic, I. Configuration Management for Component-based Systems. In *Proceedings of International Conference on Software Engineering (ICSE)* (May 2001).

[14] LIBLIT, B., AIKEN, A., ZHENG, A. X., AND JORDAN, M. I. Bug Isolation via Remote Program Sampling. In *Proceedings of Programming Language Design and Implementation (PLDI)* (2003).

[15] OSTERLUND, R. PIKT: Problem Informant/Killer Tool. In *Proceedings of LISA* (2000).

[16] PETZOLD, C. *Programming Windows with C# (Core Reference)* . Microsoft Press, 2002.

[17] REDSTONE, J. A., SWIFT, M. M., AND BERSHAD, B. N. Using Computers to Diagnose Computer Problems. In *Proceedings of HotOS* (2003).

[18] SILVER, M., AND FIERING, L. Desktop and Notebook TCO Updated for the 21st Century.

[19] SOLOMON, D. A., AND RUSSINOVICH, M. *Inside Microsoft Windows 2000*, 3rd ed. Microsoft Press, September 2000.

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

[21] TRAUGOTT, S., AND HUDDLESTON, J. Bootstrapping an Infrastructure. In *Proceedings of LISA* (1998).

[22] Tripwire. http://www.tripwire.com/.

[23] WANG, H. J., HU, Y.-C., YUAN, C., ZHANG, Z., AND MIN WANG, Y. Friends Troubleshooting Network, Towards Privacy-Preserving Automatic Troubleshooting. Tech. Rep. MSR-TR-2003-81, Microsoft Research, Redmond, WA, Nov 2003.

[24] WANG, Y. M., VERBOWSKI, C., DUNAGAN, J., CHEN, Y., WANG, H. J., YUAN, C., AND ZHANG, Z. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. In *Proceedings of LISA* (2003).