Automatic Misconfiguration Troubleshooting with *PeerPressure*

Helen J. Wang, John C. Platt, Yu Chen, Ruyun Zhang, Yi-Min Wang
{helenw, jplatt, ychen, ymwang}@microsoft.com
Microsoft Research

Abstract

Technical support contributes 17% of the total cost of ownership of today's desktop PCs [25]. 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 system: *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 [30]. 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 prototyped 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 these cases. For the remaining cases, PeerPressure significantly narrows down the number of root-cause candidates by three orders of magnitude.

1 Introduction

Today's desktop PCs have not only brought their users an enormous and ever-increasing number of features and services, but also an increasing amount of troubleshooting costs and productivity losses. Studies have shown that technical support contributes 17% of the total cost of ownership of today's desktop PCs [25]. A large amount of technical support time is spent on troubleshooting.

Many troubleshooting cases are due to misconfigurations. Such misconfiguration is often caused by data that is in shared persistent stores such as the Windows registry and UNIX resource files. These 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 one another. 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). To simplify our presentation, we will focus our discussion on a particular type of important configuration data: the Windows Registry [24], which provides hierarchical persistent storage for named, typed entries. Our discussions, techniques, and principles are directly applicable to other types of configuration stores, such as files, and other platforms, such as UNIX.

Misconfigurations can be introduced in many ways. For example, an application may unilaterally make seemingly innocuous changes to shared system configurations and cause unexpected behaviors in another application. A software bug may corrupt a Registry entry (by leaving a data field empty, for example) and break other programs that cannot handle an incorrect data format. Applying security patches may introduce Registry settings that are incompatible with existing applications [11]. Failed uninstallation of applications may introduce configuration inconsistencies resulting in malfunctions [17]. Administrators may inadvertently corrupt Registry entries when they try manually to fix misconfiguration problems using a Registry editor. Ganapathi et al. analyzed and categorized Registry misconfiguration problems based on their manifestation and scope of impact [14]. Some examples include missing Registry entries causing all network connections to disappear from the Control Panel, an extraneous entry causing a CD-ROM player to become inaccessible, a corrupted entry causing the system to log out a user upon any successful login.

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 [19]. The considerable number of possible configurations and the difficulty in specifying the "golden state" [26] (the perfect configuration) have made the problem appear to be intractable.

In this paper, we address the problem of misconfigu-

ration 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 in a short amount of 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 [30]. 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.

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 by conforming to the majority of the samples. Hence, we name this statistical troubleshooting method *PeerPressure*.

We have prototyped a PeerPressure-based troubleshooting system which draws samples from a database of 87 real-usage machine configuration snapshots. 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.

We will first give an overview of the architecture and operations of the PeerPressure troubleshooting system in Section 2. In Section 3, we detail the formulation and the analysis of the PeerPressure algorithm. We discuss our prototype implementation in Section 4. Then, we present our empirical results in Section 5. We compare and contrast our work with the related work in Section 6, address future work in Section 7, and finally conclude in Section 8.

2 The PeerPressure Architecture

Figure 1 illustrates the architecture and the operations of a PeerPressure troubleshooting system. A troubleshooting user first records the symptom of the faulty application execution on the sick machine with "App Tracer". "App Tracer" captures the registry entries that are used as input to the failed execution. These entries are the misconfiguration *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 [28]. Next, PeerPressure fetches the respective values of the canonicalized suspects from the sample set machines that also run the application under troubleshooting. The statistical analyzer then uses Bayesian estimation to calculate 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: if not, configuration state needs to be properly rolled back. This last step is not shown in the figure and we will not address it for the rest of the paper.

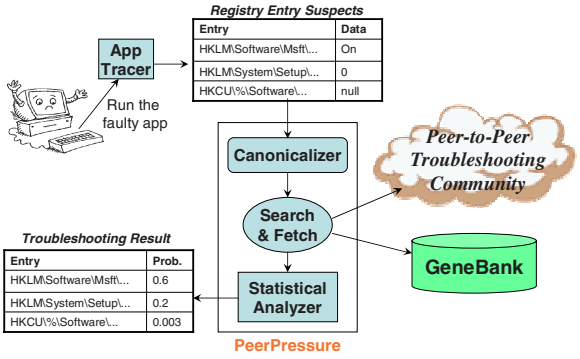

Figure 1: PeerPressure troubleshooting system architecture and its operations

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.

3 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.

3.1 Intuition and Objectives

We use an example to illustrate the intuition and objectives of formulating the sick probability calculation for each suspect. Table 1 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.

3.2 Formulation

Table 2 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 [15], we have:

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

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 assume 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.

For $P(V|H)$, we leverage the observation of 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

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

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

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 1, maximum likelihood would claim that $e2$ and $e3$ are both sick with complete and equal confidence.

Bayesian estimation [15] 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. A multinomial is a probability distribution that governs a multi-sided die, where side j has label V_j attached to it. The probability of throwing the die and getting value V_j is p_j. Because one side is always up when a die is thrown, the p_j sum to one. The multi-sided die (and the multinomial) is completely characterized by the vector p_j.

We take the Bayesian approach of treating the p_j values as random variables. Therefore, the p_j themselves follow a distribution. Purely for mathematical simplicity, we assume that the p_j follow a Dirichlet distribution [15]. Dirichlet distributions are a natural prior for multinomials, because they are *conjugate* to multinomials. That is,

	Registry Key	Suspect Machine Registry Value	Machine 1 Value	Machine 2 Value	Machine 3 Value	Machine 4 Value	Machine 5 Value
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 1: Intuition behind PeerPressure Sick Probability Formulation

combining observations from a multinomial with a prior Dirichlet yields a posterior Dirichlet. The use of Dirichlet priors is very common in Bayesian inference.

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

To perform Bayesian estimation of $P(V|H)$, we first start by choosing a prior distribution. We assume that all values V_j are equally probable *a priori*. Therefore, we start by assuming a Dirichlet distribution with a count n for all possible values V_j. We then observe our N samples of values for this registry key, collecting counts m_j for the different values. The mean of the posterior Dirichlet yields the posterior estimate $P(V_j|H)$ [15]:

$$P(V_j|H) = \frac{m_j + n}{N + cn}. \quad (4)$$

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 $1/c$. In other words, the higher the n is, the less confidence we have for the knowledge obtained from the GeneBank. The parameter 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.

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. Combining Equations (4) and (1) yields

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

Notice that Equation (5) never predicts a sick probability of zero or one, even if m is 0 or N, which is the whole point of using Bayesian statistics.

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.

Bayesian smoothing with an uninformative prior is just one method for smoothing probabilities. Another common method is the Turing-Good smoother [8], which also estimates the probability of a value unseen in the sample set. Unfortunately, Turing-Good is not easily applicable for smoothing probability of registry values: Turing-Good requires a large number of distinct values which occur once or a few times in the sample set, in order to extrapolate the probability of a value that occurs zero times in the sample set. There are many registry values that have a small number of distinct values, hence the extrapolation performed by Turing-Good is not accurate. In other words, Turing-Good works well in linguistics, where the number of distinct values (words) is large, which is not always the case in the registry.

N	Number of sample machines
t	Number of suspect registry keys
i	The index for the suspect key (from 1 to t)
V_i	The value of a suspect key i
c	Cardinality: the number of possible sample values for a suspect key
m	The number of samples that match the suspect value
$P(S)$	The prior probability that a suspect key is sick
$P(H)$	$1 - P(S)$
$P(S\|V)$	The probability that a suspect key is sick given its value
$P(V\|S)$	The probability that a sick suspect key i has value V_i

Table 2: Notation

Because we do not use Turing-Good, we must handle the case of a value that is present in the suspect registry key, but is unseen in the sample values. We handle this by counting the number of distinct values in the sample set, c_0, then adding an additional new value, called "unseen". Any suspect value that is not present in the sample set will be assigned to the "unseen" class. Thus, we set the cardinality $c = c_0 + 1$ and compute the probability of a registry key being sick given a previously unseen value to be $cn/(cnt + cm(t-1))$.

3.3 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: 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 how each of these parameters affects the sick probability and whether the trend agrees with our objectives (see Section 3.1).

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 zero and one. 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: when $N = 0$, we are unable to make any observations. In this case, 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 $t > 1$. When $t = 1$, $P(S|V) = 1$ as desired. Therefore, for our example in Table 1, 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 5, we further demonstrate through real-world troubleshooting cases that our PeerPressure algorithm is indeed effective.

4 The 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 [10], which contains real-usage registry snapshots from 87 Windows XP desktop PCs. Approximately half of these snapshots are from three diverse organizations within Microsoft: Operations and Technology Group (OTG) Helpdesk in Colorado, MSR-Asia, and MSR-Redmond. The other half are from machines across Microsoft that were reported to have potential Registry problems.

We have implemented the PeerPressure troubleshooter in C# [22], which issues queries to the GeneBank to fetch the sample values and carries out the sick probability calculation (Section 3). We use a set of heuristics for canonicalizing user-specific and 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", and "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 type. (Registry entries contain a "type" field). For a registry entry that has both numeric-typed and string-typed values among different registry snapshots, all string values are converted into numbers.

Our PeerPressure troubleshooter, although unoptimized in its present form, is already fast. We use a dual Intel Xeon 2.4GHz CPU workstation with 1 GB RAM to host the SQL server and to run the troubleshooter. On 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 sus-

Maximum registry size	333,193
Minimum registry size	77,517
Average registry size	198,376
Median registry size	198,608
Distinct canonicalized entries in GeneBank	1,476,665
Common canonicalized entries	43,913
Distinct entries data-sanitized	918,898

Table 3: Registry Characteristics

pects. Further analysis shows that database queries dominate the troubleshooting response time 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.

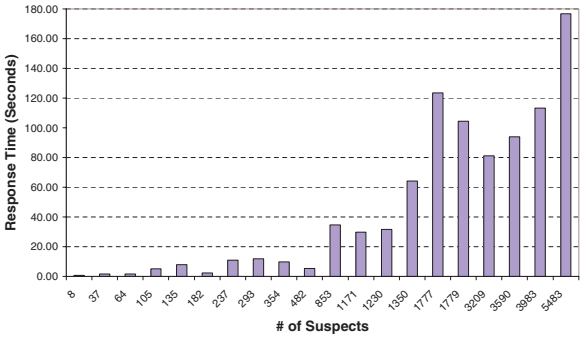

Figure 2: Response Time vs. Number of Suspects for 20 real-world troubleshooting cases.

5 Troubleshooting Effectiveness

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

5.1 Registry Characteristics

The Windows Registry contains most of the configuration data for a desktop PC. Table 3 summarizes some registry statistics 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 sizes range from 77,517 to 333,193 entries. The median is 198,608 entries. The total number of distinct canonicalized entries in the GeneBank is 1,476,665. 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 heuristics (see Section 4), we have sanitized 918,898 entries in the GeneBank.

Cardinality is an essential parameter of our PeerPressure algorithm (Section 3). Because the GeneBank may not contain all possible "genes" (entry values), we treat all values that are unknown to the GeneBank as a single value *unseen*. This unseen value effectively increments the observed cardinality from the GeneBank by one. Therefore, any entry from the GeneBank has a cardinality of at least two; and entries that do not exist in the GeneBank have a cardinality of one. In addition, 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 two, 94% no more than three, and 97% no more than four. The low cardinality of registry entries contributes to the excellent troubleshooting results of our PeerPressure algorithm (as shown in the next section) because when the cardinality is low, the conformity among the samples is strong.

5.2 PeerPressure Performance with Real-World Troubleshooting Cases

In this section, we present our empirical troubleshooting results for PeerPressure.

We use the 20 cases listed in Table 4 for our experiments. They were all real-world failures that troubled some users. We have the knowledge of root-cause misconfiguration *a priori*. We picked 20 out of 129 accessible cases from Microsoft (MS) Product Support Service (PSS) e-mail logs, MS Helpdesk, web support forums, and our co-workers' reported problems. The only criterion we used in selecting these cases is the ease of reproducing application failures since some cases require special hardware setup or specific software versions. Cases 1, 11, 13, and 14 are among the most commonly encountered configuration problems from MS PSS e-mail logs [14]; Cases 2, 9, and 10 are from MS Helpdesk; Cases 15-18 are from a web support forum; and the rest are from our co-workers.

Since we know the misconfigured, root-cause entry for each case, we use the ranking of the 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, the entries that are

ID	Name	Description of Problem
1	System 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 does not 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 does not offer to automatically save passwords; the option to re-enable the feature is difficult to find.
5	Media Player	Windows Media Player "Open Url" function fails if the EnableAutodial Registry entry is changed from 0 to 1 on a corporate desktop.
6	IM	MSN Instant Messenger (IM) significantly slows down if the firewall client is disabled on a corporate desktop.
7	IE Proxy	IE on a machine with a corporate proxy setting fails 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 is then be presented with a cached offline page instead of the default start page when launching IE.
9	Taskbar	IE windows are unexpectedly grouped under the Windows Explorer taskbar group, due to the addition of a Registry entry.
10	Network Connections	Control Panel → Network Connections shows nothing, due to a missing Registry key.
11	Folder Double-Clicking	Double clicking any folder in the right pane of Windows Explorer incorrectly brings up the "Search Results" window.
12	Outlook Express	Microsoft Outlook could not be started because the Outlook Express installation appears to be missing, due to a missing Registry key.
13	Cannot Start Executables	Double-clicking any EXE file does not launch the application.
14	Shortcut	Double-clicking any shortcut does not launch the application.
15	IE Menu Bar	IE menu bar disappears due to a corrupted Registry key name.
16	IE Favorites	IE uses the "unknown file type icon" for some of the links in the Favorites.
17	Sound Problem	Warning sound is 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" shows nothing.
19	Yahoo Toolbar	Yahoo Companion per-user installation affects all users.
20	Media Player in IE	Internet Explorer always launches Media Player on the left pane.

Table 4: 20 Real-World Troubleshooting Cases Used for PeerPressure Evaluation

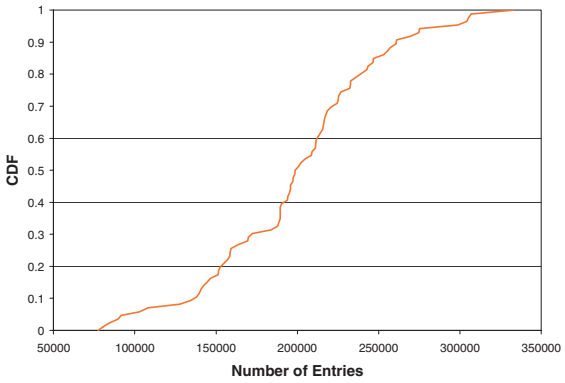

Figure 3: Registry Size Distribution

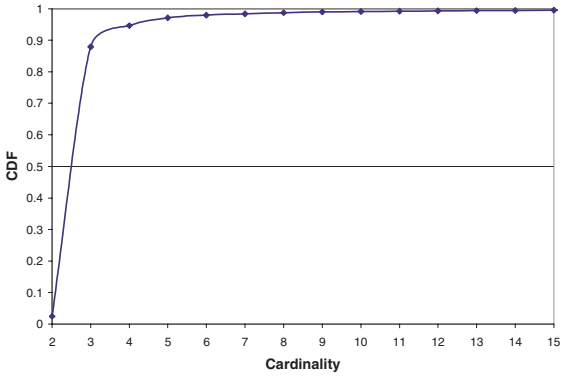

Figure 4: Cardinality Distribution

touched during the faulty execution of the application (see Section 2). Finally, we ran PeerPressure to produce the ranking reports.

5.2.1 Root Cause Ranking

For each troubleshooting case, Table 5 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 number of samples. Ranking ties can happen when the estimated probabilities of some entries have the same values. 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 indispensable 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 application and they happen to be sick and have the same sick values as well.

5.2.2 The Causes of False Positives

In this section, we give an analysis on the causes of false positives. The sick probability metric ranks the novelty of a suspect entry in the samples from the GeneBank. The more novel a suspect is compared with other suspects, the lower its rank number is (i.e., the more sick the suspect is). xs 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 3. Case 20 falls into this category of false positives. The root-cause entry for Case 20 has a high cardinality of 65 while the rest of the cases have low cardinalities (Table 5).

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 4 indicate the number of machines in the GeneBank that have the same sickness. This affected the ranking of Case 2, 6, and 10.

5.2.3 The Impact of the Sample Set Size

It is intuitive that the larger the sample set is, and the 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 6 shows root-cause ranking trend for various sample set sizes. The average number of ties for each sample set size is indicated in the parentheses. For the first three cases in the table, the root-cause ranking is perfect regardless of the sample set size. This perfect behavior is caused by all samples of the root-cause entry taking the same value. Any subset of the GeneBank samples still has the same value. No other suspects have a high sick probability when the sample set is small.

The cases belonging to the middle portion of Table 6

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 Favorites	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 5: 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 6: Impact of the Sample Set Size

Case	Machine 1	Machine 2	Machine 3
2	16 (0) / 1779	32 (2) / 1272	14 (0) / 1272
5	1 (0) / 182	1(0) / 566	1 (0) / 1657
6	1(0) / 2789	12(0) / 1777	12 (0) / 2017
14	1(0) / 105	1(0) / 84	1 (0) / 64
16	1 (0) / 302	2(0) / 3209	1 (3) / 1908

Table 7: Sick Machine Sensitivity Evaluation. Each entry has the format of RootCauseRanking (numberOfTies) / numberOfSuspects.

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. Therefore, 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 conformance 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 6, 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 5. In this situation, enlarging the sample set reduces the impact of the polluted entries and therefore contributes to the decreasing trend of the rankings.

5.2.4 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 7 shows that the troubleshooting results on these sick machines are mostly consistent. In some cases, such as Case 6, a larger suspect set leads to better ranking rather than introducing more noise, as one would have expected. 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.

6 Related Work

There are two general approaches in system management: the white-box [6][3][9][21][16][27] and the black-box approach [30]. 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 [30], the precursor of this work, uses 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 2). 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 that is unlikely to be a root cause.

PeerPressure also takes the general black-box approach. PeerPressure differs from Strider in the following 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 replaces the state-differencing and noise-filtering steps of Strider with a more general step of statistical analysis.

3. Strider uses order ranking which means that the final ordering of suspects is 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 that 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. [23] 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 [12][18][13] 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. [12] 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 [18]. Forrest et al.'s seminal work on host-based intrusion detection system [13] 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 components or source code for debugging. Liblit et al. [20] 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.

Brown et al [5] takes a black-box approach to infer hardware or software component dependencies by actively probing a system. Statistical model is then used to estimate dependency strengths. Similarly, Brodie et al [4] uses active probing with network events like *pings* or *traceroutes* for diagnosing network problems in a distributed system: a small set of high quality probes are first selected, then a Bayesian network is constructed for inferring faults.

The PinPoint root-cause analysis framework [7] is a debugger for component-based systems. PinPoint identifies individual faulty components that cause service failures in a distributed system. PinPoint uses data clustering 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.

7 Future Work and Discussion

We have interesting 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 in addition to formulating the multi-gene sick probability are non-trivial tasks because the sick probability of entries are no longer independent.

Sometimes, a failed application execution may be caused by another application's misconfigurations. For example, a web browser may fail because of an incorrect VPN setting. The trace obtained from our AppTracer would not contain root-cause misconfigurations of another application. Cross-application misconfiguration troubleshooting remains an open challenge.

In environments where most or all machines have automatically maintained configurations, sample set selection criteria would need to be adjusted not to include the machines under the same automatic management.

PeerPressure takes the advantage of the strong conformance in most of the configuration entries for diagnosing the anomalies. However, some technology savvy users may customize their PCs so heavily that their configurations appear unique or "anomalous" to PeerPressure. These are inevitable false positives produced by PeerPressure.

Another open question is GeneBank maintenance. The GeneBank currently has one-time machine configuration snapshots from 87 volunteers. Without further maintenance, 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 2). 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-preserving peer-to-peer troubleshooting system. For details, please see [29]

8 Conclusions

We have presented PeerPressure, a novel troubleshooting system that 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.

9 Acknowledgements

We inherited the "App Tracer" that was implemented by Chad Verbowski for the Strider toolkit [30]. Chad has also given us valuable feedback and discussions on the GeneBank database schema design as well as on canonicalization heuristics for Registry entries. Emre Kiciman has also contributed to the canonicalization heuristics design. Chris Meek has given us valuable suggestions on Bayesian estimation algorithms. We received much advice from our database experts (Venki Ganti, Zhiyuan Chen, and Nico Bruno) for the GeneBank design and optimizations. 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. Mike Jones and Atul Adya also gave us helpful comments on the paper and encouraged us to submit it to OSDI. The anonymous OSDI reviewers and our Shepherd Matt Welsh provided us detailed and insightful critiques. We thank everyone for their 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] BRODIE, M., RISH, I., AND MA, S. Intelligent probing: A Cost-Efficient Approach to Fault Diagnosis in Computer Networks. *IBM Systems Journal 41*, 3 (2002).

[5] BROWN, A., KAR, G., AND KELLER, A. An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment. In *Seventh IFIP/IEEE International Symposium on Integrated Network Management* (may 2001).

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

[7] 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).

[8] CHURCH, K., AND GALE, W. A comparison of the enhanced Good-Turing and deleted estimation methods for estimating probabilities in English bigrams. *Computer Speech and Language 5* (1991), 19–54.

[9] COUCH, A., AND GILFIX, M. It's Elementary, Dear Watson: Applying Logic Programming to Convergent System Management Processes. In *Proceedings of LISA* (1999).

[10] DELANEY, K. *Inside Microsoft SQL Server 2000*. Microsoft Press, 2001.

[11] DUNAGAN, J., ROUSSEV, R., DANIELS, B., JOHNSON, A., VERBOWSKI, C., AND WANG, Y.-M. Towards a Self-Managing Software Patching Process Using Black-Box Persistent-State Manifests. In *Proceedings of Int. Conf. on Autonomic Computing (ICAC)* (May 2004).

[12] 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).

[13] 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).

[14] GANAPATHI, A., WANG, Y.-M., LAO, N., AND WEN, J.-R. Why PCs Are Fragile and What We Can Do About It: A Study of Windows Registry Problems. In *Proceedings of IEEE Dependable Systems and Network (DSN)* (June 2004).

[15] GELMAN, A., CARLIN, J., STERN, H., AND RUBIN, D. *Bayesian Data Analysis*. Chapman, 1995.

[16] 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).

[17] KICIMAN, E., AND WANG, Y.-M. Discovering Correctness Constraints for Self-Management of System Configuration". In *Proceedings of Int. Conf. on Autonomic Computing (ICAC)* (May 2004).

[18] 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).

[19] LARSSON, M., AND CRNKOVIC, I. Configuration Management for Component-based Systems. In *Proceedings of International Conference on Software Engineering (ICSE)* (May 2001).

[20] 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).

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

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

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

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

[25] Web-to-Host: Reducing the Total Cost of Ownership, The Tolly Group. http://www.wrq.com/assets/products_tolly_tco.pdf, May 2000.

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

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

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

[29] WANG, H. J., HU, Y.-C., YUAN, C., ZHANG, Z., AND WANG, Y.-M. Friends Troubleshooting Network: Towards Privacy-Preserving, Automatic Troubleshooting. In *IPTPS* (San Diego, CA, Feb 2004).

[30] 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).