Snitch: Interactive Decision Trees for Troubleshooting Misconfigurations

James Mickens
University of Michigan
Ann Arbor, MI
jmickens@eecs.umich.edu

Martin Szummer
Microsoft Research
Cambridge, UK
szummer@microsoft.com

Dushyanth Narayanan
Microsoft Research
Cambridge, UK
dnarayan@microsoft.com

Abstract

Troubleshooting misconfigurations of modern applications is difficult due to their large and complex state. Snitch is a prototype tool that assists human troubleshooters by finding relationships between application state and subsequent faults. It correlates configuration state and application errors across many machines and users, and across long periods of time. Snitch aids the human expert in extracting patterns from this rich but enormous data set by building decision trees pinpointing potential configuration problems.

We applied Snitch to 114 GB of configuration traces from 151 machines over 567 days. We illustrate how Snitch can suggest misconfigurations in case studies of two Windows applications: Messenger and Outlook.

1 Introduction

Misconfiguration is a leading cause of software faults. Configuration errors are the largest category of operator errors for Internet services [6], and many types of misbehavior in Windows programs are caused by registry entries [11]. As programs grow more complex with ever more configuration settings, it becomes increasingly difficult to understand all of their potential interactions. Furthermore, software components from different vendors often update and interpret shared state in mutually incompatible ways, leading to "DLL hell" in Windows and RPM drift in Linux [3].

The sheer volume of persistent state, and the complex interactions between this state and the programs that manage it, make it difficult to determine which settings are the root cause of a particular fault. For example, a typical Windows PC has 70,000 files, 200,000 registry entries, and dozens of programs [9]. If one of these programs starts to misbehave, how can we determine the particular piece of state which is misconfigured?

In this paper, we propose a three-step process to troubleshoot a misconfigured application. First, we use an *always-on tracing environment* [2, 10] to record all reads and writes of configuration state. Second, we explicitly identify faulty application runs using *outcome markers*. Some markers, such as error messages or exit codes, must be captured in real time by tracing. Others might be identified after the fact, e.g., via offline log analysis. Given traces of configuration operations and outcome markers, we then use machine learning to do "root-cause localization" [4], i.e., to correlate bad outcomes with the configurations that likely caused them.

Creating a fully automatic method for unearthing root causes is very difficult. Standard machine learning techniques cannot distinguish correlation from causation; they generally ignore temporal information (how long ago and in what order state was read or written); and most importantly they lack semantic understanding of the state being accessed. Thus, while we use machine learning techniques to reduce the burden on a human troubleshooter, we also let her guide the learning algorithms by applying expertise that is not easily automated.

Using *interactive decision trees*, we allow user input to guide the construction of a fault diagnosis tree. The interaction process is supported by *timeline views* and *attribute generalization*. We have implemented these techniques in a prototype of Snitch, a tool to aid in configuration troubleshooting. Our target scenario is that of a user calling a help desk to complain about an observed application fault. The help desk expert has access to configuration traces from many users, but the sheer volume of this data makes manual examination impossible. Instead, she runs Snitch on the data to interactively diagnose the probable cause of the fault.

The remainder of this paper is organized as follows. Section 2 describes interactive decision trees, timeline views, and attribute generalization. Section 3 describes the implementation of the Snitch prototype. Sections 4 and 5 present two application case studies. Section 6 discusses related work and Section 7 concludes with a summary of the lessons learned.

2 Interactive Decision Trees

In our model, programs access configuration state by reading and writing *keys* that have associated *values*. Specifically, on the Windows platform, keys are either file names or the names of registry entries. The goal of our troubleshooting algorithm is to identify the keys and values that induce faults.

We assume an always-on tracing environment such as Flight Data Recorder [10] which tracks every key/value read and write. The tracing environment provides a timestamp and key name for each registry and file access as well as information about the process generating the access, e.g., program name, start time, and exit code. It is desirable to also know the value that was read or written, but useful fault inferences can be made in the absence of such knowledge.

Each application run is associated with one or more *outcome markers* which indicate any anomalous behavior that was exhibited by the run. Using outcome markers, we can explicitly differentiate between "good" runs and "bad" runs, allowing us to avoid the assumption that healthy configurations are widespread. Possible outcome markers include application exit codes and error log messages, as well as users explicitly marking faulty application runs.

2.1 Basic Design

A decision tree [7] predicts the *class* of an *instance* based on its *attributes*. Each interior node splits the set of instances by testing a specific attribute: each child of that node will correspond to a distinct value of that attribute. Each leaf node predicts the class of all instances whose attributes trace that path down the tree. In Snitch, classes correspond to application outcomes. Instances are application runs, attributes are the registry entries and files accessed by the application, and attribute values indicate whether the attribute was accessed or not (and possibly the value). We reduce all attributes to binary ones, each indicating whether a particular key/value pair, or a particular key irrespective of value, was accessed.

We chose decision trees because they can be efficiently learned and applied to large data sets. They also offer compact, human-readable representations of relationships between faults and potential causes. This is a key requirement since our aim is *explanation* rather than *prediction*: Snitch's output is intended to assist help desk experts in finding and fixing misconfigurations. Approaches such as neural nets offer "black-box" prediction rather than human-readable explanation and are unsuitable for our purposes.

We build our decision trees as binary classifiers that distinguish a single outcome from other outcomes, since these are smaller and easier for users to interpret. Our method is similar to the C4.5 algorithm [7]: leaf nodes are split greedily until they have sufficiently low classification error (we do not prune the tree after building it).

Snitch greedily splits each node on the attribute with the highest *information gain*. Given a set of instances S with outcome probability distribution $P(c)$, we define the entropy, a measure of class "pureness," as:

$$\text{Entropy}(S) = -\sum_{\text{all classes } c} P(c)\log_2 P(c).$$

The information gain from splitting on an attribute A with value v resulting in an instance subset S_v is then

$$\text{Gain}(S, A) = \text{Entropy}(S) - \sum_{\text{val } v \in A} \frac{|S_v|}{|S|}\text{Entropy}(S_v).$$

If different attributes have the same information gain, standard decision tree algorithms will choose one of them arbitrarily. Such arbitrary tie-breaking can hide important troubleshooting information. For example, suppose that a program acts faultily when it reads a particular configuration value, and that an error-reporting DLL is loaded after the fault occurs. The fault will be equally correlated with both the bad configuration setting and the DLL file. The decision tree might split on either one, although a human user would probably regard the configuration value as a more useful causal explanation of the fault.

In general, actions that are highly correlated with a fault might be causes or effects of the fault. Snitch allows the user to distinguish these through *interactive splitting*, permitting the user to select the most appropriate split attribute at each step. Whenever a node must be split, Snitch displays the *attribute group* with the greatest information gain and asks the user to choose one group member as the human-readable label for the split. Each attribute group contains a set of attributes that are perfectly correlated with each other, i.e., splitting on any member of the group is equivalent to splitting on any other. However, some attributes in the group may be more useful to a human as explanations of a fault.

If none of the attribute group members seem meaningful, the user can examine the next best attribute group. This group will provide less information gain but it may have more meaningful attributes. Thus, interactive splitting facilitates two types of interaction. Allowing the user to choose attribute group representatives gives us meaningful node labels without changing the tree structure. Allowing the user to choose a different attribute group lets them change the tree structure.

If desired, the entire decision tree can be generated in non-interactive mode with some specified target classification error. The user can then examine each split in the generated tree to choose the labels.

Deep trees may overfit the data. We rely on the user to interactively stop tree construction early, before overly complex explanations have been built. The user can decide to stop growing a tree branch based on statistics calculated by Snitch, such as the distribution of exit codes

Table	Columns
Event	TimeStamp, IsRead, Attribute, Value, InstanceId
Instance	InstanceId, StartTime, ExitTime, ExitCode, ProgramName, MachineName
String	StringId, String

Table 1: Schema for post-processed traces.

at the branch and their absolute numbers. The user may also stop branch growth due to the desired amount of tree detail.

A fault may arise from the misconfiguration of multiple keys. If the keys have perfectly correlated values, then only one of them will be explicitly represented as a split attribute in the tree. However, during interactive splitting, Snitch will display all of these keys in a single attribute group. This allows a human troubleshooter to notice any unexpected correlations between configuration state.

2.2 Timeline Views

When deciding between multiple attributes that are equally correlated with some outcome, it is often useful to consider the *timestamps* of the corresponding read/write events. For example, an earlier event might be a better candidate for a root cause than a later event.

We found it helpful to look at timeline views while interacting with Snitch. Given some set of attributes that are highly correlated with the outcome, a timeline view shows the order of the corresponding configuration events in a specific application run. As we show in Section 4, these timelines can be useful in validating a root cause hypothesis that is suggested by looking at a decision tree.

2.3 Attribute Generalization

By default, we assume that each key name/value pair is a unique attribute. However, when processing data across machines and users, we often find key names that are semantically equivalent but differ because they contain a user name or machine ID. In these cases it might be valuable to generalize these attributes, for example by wildcarding the user or machine name. In the case study in Section 5 we show how simple generalization based on wildcarding can substantially improve the decision trees.

3 Data and Implementation

To evaluate Snitch, we used trace logs from the Flight Data Recorder (FDR) project [10]. The logs tracked the registry, file, and process activity of 151 Windows hosts over 567 days for a total of 7,401 machine-days. Registry and file activity consisted of operations such as opens, reads, deletes, etc. Process activity included process creation and exit events as well as exit codes.

Before building a tree for a particular application, we determined all of the keys (registry entries and files) that were read during any run, using them to generate the attributes for tree-building. Each key generated at least one binary attribute which specified whether it was read or not read. Additionally, each observed value for a key generated one binary attribute which specified whether that key was read with that value.

We used a program's exit code as its outcome marker. Compared to more detailed markers such as error logs, exit codes provide limited information about a subset of program faults. Unfortunately, our input traces did not include error logs.

The FDR traces presented us with a few other challenges. For reasons of efficiency, the traces did not contain the binary data that was read from or written to files. Data values were logged for registry writes but not reads. We inferred values for registry read events using previous writes (if any) to the same registry entry on the same machine. However, the traces did not start at the beginning of each machine's lifetime, and few writes were observed during the traced period. Thus, only 2% of read/write events were of a known value, and in our case studies these were never selected by the decision tree algorithms. In other words, due to an artifact of the data, the trees generated by Snitch used only "read/not read" attributes. However, as we will see, these attributes were still useful as indicators of misconfigured state.

We stored the FDR data in a Microsoft SQL Server database using the schema shown in Table 1. The **Event** table stored all reads and writes to persistent state. The **Instance** table uniquely identified each application run. Strings such as key names and machine names were stored in a separate **String** table and referenced in the other two tables through unique integer IDs. The tree construction code was implemented in C# and used LINQ [5] to query the database.

A few simple optimizations made tree construction fast and efficient. First, by filtering out all key events except reads and writes, and by representing the data as described above, we reduced 114 GB of raw trace data to 5.3 GB of SQL data. Second, by careful use of LINQ queries, we were able to push much of the data processing into the SQL Server back end, minimizing the amount of data read from the server and the processing load on the client. Third, before constructing the tree, we

collected all perfectly correlated attributes into attribute groups. This substantially reduced the number of split candidates, e.g., by a factor of 25 in our Outlook case study. Since each split candidate must be examined at each node to compute its information gain, this led to a substantial speedup. In our two case studies, tree size varied from 1 to 19 nodes, and the worst-case tree construction time in non-interactive mode was 0.45 seconds.

4 Case study: MSN Messenger

MSN Messenger is Microsoft's instant messaging client. In Figure 1(a), we depict a classifier for the exit code -1073741819. Using a single split, the tree achieves perfect classification accuracy — Messenger exits with this code if and only if it reads a `wpad[1].dat` file in Internet Explorer's temporary data folder.

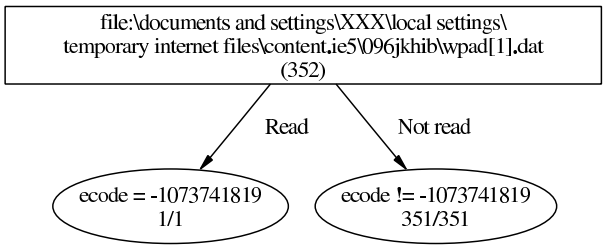
The decision tree is simple and accurate, but it is not interpretable. *Why* does this split offer perfect classification accuracy? Since we are performing black-box analysis, we cannot look at Messenger's source code and find the code paths that are activated by consumption of specific files. However, there is additional context that is ignored by standard tree algorithms but can provide clues to the root cause of the fault.

During interactive splitting, Snitch displays attribute groups ranked by decreasing information gain. We searched the web for information about the most promising-looking attributes in the first two groups. These attributes fell into four categories:

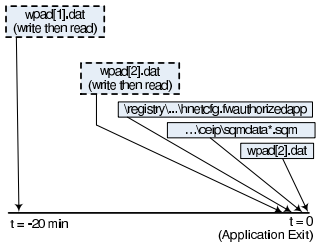
- `wpad*.dat` files in IE's temporary folder
- `windows messenger\ceip\sqmdata*.sqm` files in the user's application data folder
- A `HNetCfg.FwAuthorizedApplication` registry entry
- A registry entry containing the GUID `ec9846b3-2762-4a6b-a214-6acb603462d2`

The `wpad*.dat` files are Web Proxy Automatic Detection Files automatically generated by Internet Explorer and used by applications to discover nearby web proxies. The `.sqm` files are Service Quality Monitoring files generated by Microsoft's Customer Experience Improvement Program (CEIP). They log errors encountered during program execution. `HNetCfg.FwAuthorizedApplication` refers to a Windows firewall COM object which grants network access to applications; the GUID is simply an alias for this object.

Given that three of these categories pertain to firewall settings and the fourth to error reporting, we hypothesized that the exit code -1073741819 was caused by misconfigured firewall settings which prevented Messenger from using the network. This caused CEIP activity before the program exited. This hypothesis was supported by the timeline view (Figure 1(b)) for the left-hand tree branch (the single instance corresponding to the anomalous run of Messenger). Shortly before the



(a) Numbers in parentheses show the number of instances classified by an interior node. For leaf nodes, the numerator shows the number of correctly classified instances, and the denominator the total. `XXX` represents a single (anonymized) user.



(b) Timeline view

Figure 1: Decision tree and timeline view for Messenger.

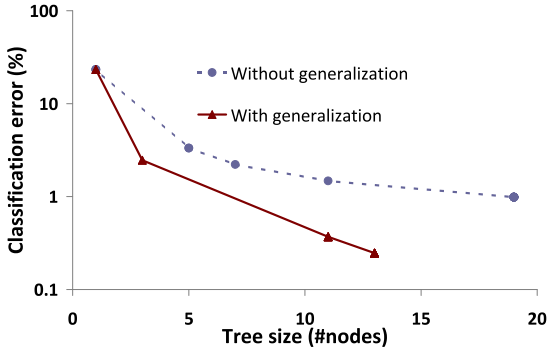


Figure 2: Classification error vs. tree size for Outlook.

program exited, it wrote and then read a `wpad[2].dat` file. Since there were no intervening writes by other programs, any bad settings in that file must have been written by Messenger itself. Soon after this Messenger accessed `HNetCfg.FwAuthorizedApplication`, learned (presumably) that it could no longer access the network, and generated CEIP activity before exiting.

5 Case study: Microsoft Outlook

In our next case study, we investigated the Outlook email client. Outlook is a complex application and a single run can generate over 30,000 read/write events. Given the volume of data, we confined user interaction in this case study to choosing node labels after the tree was generated.

We investigated exit code -1. Our database contained 812 runs of Outlook, 189 of which exited with a -1.

Across these runs we observed 97,056 distinct attributes in 3,687 attribute groups. We first explored the trade-off between tree size and classification accuracy. We built trees of varying sizes by varying the error threshold c between 50% and 0.07%. This produced trees with between 1 and 19 nodes (Figure 2). Note that the threshold affects the overall classification error but is neither an upper nor a lower bound on it.

Figure 3(a) shows the decision tree computed with $c = 12.5\%$. The tree has 5 nodes and an overall error of 3.3%. If we expand the tree further, the error shrinks but the tree size increases substantially; with $c = 6.25\%$, the error is 1.5% but the tree has 11 nodes (Figure 3(b)). Furthermore, each additional split only classifies a small number of instances, i.e., each split has to choose from a large number of attributes providing a small information gain.

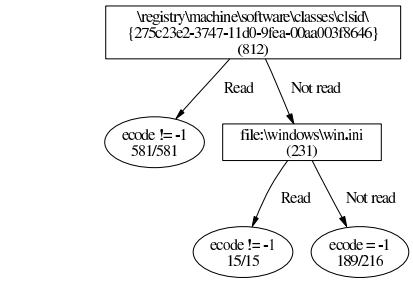
We noticed that several attributes had user names embedded in them. We speculated that *generalizing* such attributes by wildcarding the user name might improve the trees. We applied three simple generalizations based on regular-expression matching: wildcarding user names in keys beginning with `\registry\user\*\` and `\documents and settings\*\`, and wildcarding filenames while retaining the pathname and file extension. The generalized attributes were added to the original attribute set. This increased the number of attributes by 73% and the number of attribute groups by 30%. With generalized attributes, the same c yields a tree of the same size but with an error of only 0.7% (Figure 3(c)). More importantly, the tree achieves most of its accuracy by using two generalized attributes.

The generalized tree suggests that the exit code is related to reading user-specific `startupitems` data and `.pf` files residing in a global prefetch directory. We believe that this is a better explanation than the one generated without attribute generalization; however, we cannot be sure of this as we do not know the ground truth for this particular data set. We are currently investigating the -1 exit code in order to validate the tree.

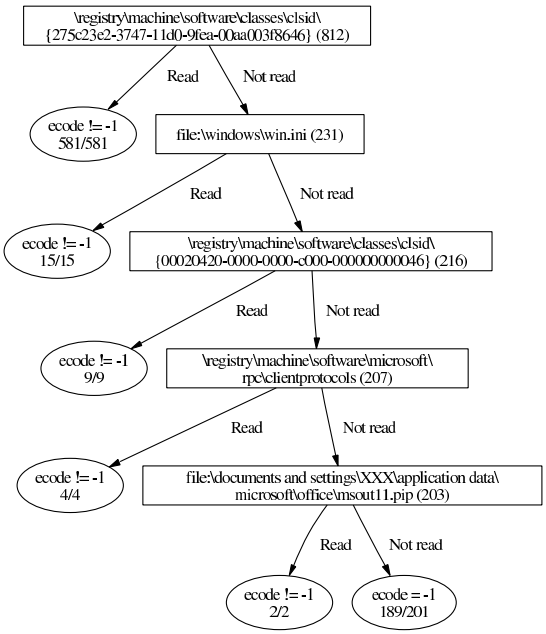
6 Related Work

The closest related work to Snitch is PeerPressure [11], the current state of the art in troubleshooting registry settings. PeerPressure identifies faults through statistical analysis of registry snapshots from a large number of machines. Faulty programs are re-executed in a special tracing environment which captures the registry data that is read. These values are then compared to the corresponding values that are stored in other machines' registries. Rare values are nominated as faulty.

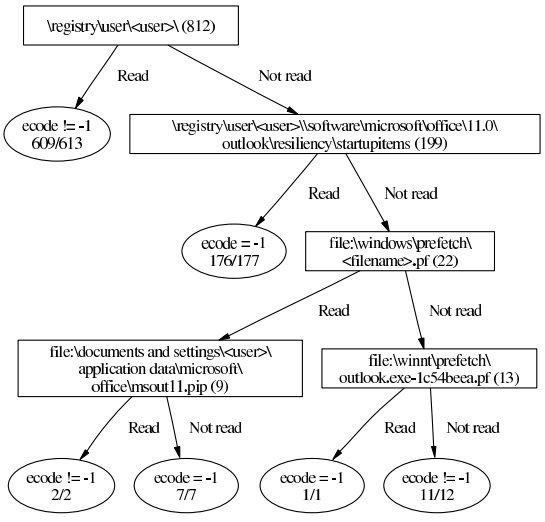
Snitch addresses three limitations of PeerPressure. First, PeerPressure's assumption that "common is correct" is not always true. For example, a patch for one application that breaks a configuration setting for another



(a) $c = 12.5\%$, 3.3% error



(b) $c = 6.25\%$, 1.5% error



(c) $c = 6.25\%$ with generalization, 0.7% error

Figure 3: Decision trees for Microsoft Outlook.

might be installed system-wide before any problems are noted. Second, PeerPressure requires the fault to be reproducible by re-running the application. Always-on tracing allows Snitch to diagnose past faults without requiring their reproduction. Finally, lack of always-on tracking means that PeerPressure is limited to "one-step" causal analysis. It can find suspect registry entries, but not the programs or the chain of events that caused them.

Chen *et al* use decision trees to diagnose failures in the eBay system [1]. Each user request is tagged with attributes representing the software systems that handled it. These attributes are culled from an always-on logging infrastructure. There are three major differences between Chen's work and Snitch. First, Snitch provides interactive tree construction, allowing a human troubleshooter to guide the diagnostic process using extrinsic domain knowledge. Second, using timeline views, Snitch lets troubleshooters examine the temporal interactions between configuration operations. Third, Snitch must deal with noise in the attribute namespace. In our problem domain, the semantic equivalence between registries and files on different machines is often obscured by attribute names containing local identifiers. Snitch uses attribute generalization to remove this noise and produce tractable data sets.

7 Conclusions

Our case studies demonstrate that Snitch provides useful insights into the relationship between configuration state and exit codes. High-level explanations of faults still require a human expert to search the web and interpret the results. This is challenging when the troubleshooter is not the original application developer and lacks an understanding of the program's internal operation. Troubleshooting could be made easier by raising the level of abstraction, e.g., through a database that mapped GUIDs to the corresponding objects and services. Other useful databases would associate keywords such as "error reporting" or "firewall" with registry keys, file names, and application exit codes. A global repository of structured fault reports [8] would also make it easier to identify common misconfigurations across many machines.

We found that timeline views were very useful in diagnosing faults. Currently, the user must manually combine this temporal data with the information gain metrics provided by Snitch. Ideally, Snitch would combine these two metrics automatically, perhaps by favoring split attributes that have high information gain and also occur earlier in time, under the assumption that an earlier action is more likely to be a root cause than a later one. Since standard decision tree algorithms do not consider temporal information, future work involves extending our algorithms to do this.

Another insight from the Outlook study is the importance of attribute generalization through the removal of local uniquifiers from attribute names. Currently we generalize using a small, fixed set of regular expression templates; we plan to extend this by using string matching and clustering techniques to automatically extract generalized attributes.

Our immediate plan for the future is to validate Snitch using a broader set of applications and richer traces that include registry values and error logs. In the medium term, we plan to investigate other machine learning techniques and their applicability to this problem. We would also like to develop distributed versions of these techniques for use in large networks where centralization of all trace data might be impractical.

References

[1] M. Chen, A. Zheng, J. Lloyd, M. Jordan, and E. Brewer. Failure Diagnosis Using Decision Trees. In *Proceedings of the International Conference on Autonomic Computing*, pages 36–43, New York, NY, May 2004.

[2] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling intrusion analysis through virtual-machine logging and replay. In *Proc. Operating Systems Development and Implementation (OSDI)*, Dec. 2002.

[3] J. Hart and J. D'Amelia. An analysis of RPM validation drift. In *Proc. Large Installation System Administration Conference (LISA)*, Nov. 2002.

[4] E. Kıcıman and L. Subramanian. Root cause localization in large scale systems. In *Proc. 1st Workshop on Hot Topics in Systems Dependability (HotDep)*, June 2005.

[5] Microsoft. The LINQ project. `http://msdn.microsoft.com/data/ref/linq/`, Nov. 2006.

[6] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *Proc. USENIX Symposium on Internet Technologies and Systems (USITS)*, Mar. 2003.

[7] J. R. Quinlan. *Programs for Machine Learning*. Morgan Kaufmann, 1993.

[8] J. A. Redstone, M. M. Swift, and B. N. Bershad. Using computers to diagnose computer problems. In *Proc. 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[9] C. Verbowski, E. Kıcıman, B. Daniels, S. Lu, R. Roussev, Y.-M. Wang, and J. Lee. Analyzing persistent state interactions to improve state management. Technical Report MSR-TR-2006-39, Microsoft Research, Apr. 2006.

[10] C. Verbowski, E. Kıcıman, A. Kumar, B. Daniels, S. Lu, J. Lee, Y.-M. Wang, and R. Roussev. Flight Data Recorder: Monitoring persistent-state interactions to improve systems management. In *Proc. Operating Systems Development and Implementation (OSDI)*, Nov. 2006.

[11] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with PeerPressure. In *Proc. Operating Systems Development and Implementation (OSDI)*, Dec. 2004.