Automated Known Problem Diagnosis with Event Traces

Chun Yuan[1], Ni Lao[3], Ji-Rong Wen[1], Jiwei Li[4], Zheng Zhang[1], Yi-Min Wang[2], Wei-Ying Ma[1]

[1]Microsoft Research Asia
5/F, Sigma Center, No. 49 Zhichun Road,
Beijing, China 100080
+86-10-62617711
{cyuan,jrwen,zzhang,wyma}@microsoft.com

[2]Microsoft Research
One Microsoft Way
Redmond, WA 98052
+1 (425) 7063467
ymwang@microsoft.com

[3]Tsinghua University
9-101, Tsinghua University,
Beijing, China 100084
+86-10-62777057
noon99@mails.tsinghua.edu.cn

[4]University of Science and Technology of China
No. 96 Jinzhai Road,
Hefei, Anhui, China 230026
+86-551-3601800
li_jiwei@ustc.edu

ABSTRACT
Computer problem diagnosis remains a serious challenge to users and support professionals. Traditional troubleshooting methods relying heavily on human intervention make the process inefficient and the results inaccurate even for solved problems, which contribute significantly to user's dissatisfaction. We propose to use system behavior information such as system event traces to build correlations with solved problems, instead of using only vague text descriptions as in existing practices. The goal is to enable automatic identification of the root cause of a problem if it is a known one, which would further lead to its resolution. By applying statistical learning techniques to classifying system call sequences, we show our approach can achieve considerable accuracy of root cause recognition by studying four case examples.

Categories and Subject Descriptors
D.2.5 [**Software Engineering**]: Testing and Debugging – *diagnostics, tracing*

General Terms
Algorithms, Measurement, Experimentation

Keywords
System call sequences, root cause analysis, support vector machine

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
EuroSys'06, April 18–21, 2006, Leuven, Belgium.
Copyright 2006 ACM 1-59593-322-0/06/0004...$5.00.

1. INTRODUCTION
With the increasing complexity of modern computer systems, problem diagnosis has become a major challenge for users and support engineers. A typical process of problem diagnosis is seemingly simple: A human or machine troubleshooter first provides a description of the problem that has happened, then some sort of analysis is performed and the root cause of the problem is identified, and finally a remedy is applied to solve the problem. In short, the paradigm of problem diagnosis is:

Problem description ➔ root cause ➔ solution

In general, the "**root cause ➔ solution**" pairs can be obtained from past problem-solving experiences. These labeled pairs can be called "known problem". In such cases, the problem diagnosis paradigm can be transformed to:

Problem description ➔ known problem

We intend to use the simplified paradigm to automate the process of problem diagnosis, leveraging past problem-solving knowledge to train classifiers for automatic identification of the root cause of a problem. Let us first take a look at the following example:

Example – "The page cannot be displayed" problem in Windows Internet Explorer (IE): *when the computer has a network connection or setting error, the "The page cannot be displayed" error message will be prompted to the user. Through analyzing the problem-solving logs of a technical support center, we identified the top 10 possible root causes for this problem:*

1. *BadIP: IP address is invalid*
2. *BadPort: the specified server port is invalid*
3. *BadProxy: the HTTP proxy is invalid*
4. *BadProxyPort: the HTTP proxy port is invalid*
5. *Disable: the network connection is disabled*

6. *NoDriver: the driver of the network adapter is not installed*
7. *NoPage: the page visited does not exist on the server*
8. *NoProtocol: the TCP/IP protocol is not enabled*
9. *Unplug: the network cable is unplugged*
10. *WinSock2: WinSock2 registry key is corrupted*

This is a very typical problem that needs significant efforts for troubleshooting because for the same symptom "The page cannot be displayed", there are at least 10 possible root causes. Moreover, the root causes may come from diverse sources. For instance, in the above listed 10 root causes, three of them are related to TCP/IP, three related to HTTP, two related to network driver, one related to registry, and another related to hardware.

Currently, there are two kinds of mainstream problem diagnosis methodologies – text-based diagnosis and state-based diagnosis. Using the above example problem, we analyze the pros and cons of these two existing methodologies.

Text-based diagnosis – Traditionally, the user finds solution by text-based search in past problem solution knowledge base. A common experience is that the user summarizes the symptom of a problem with her own words and then search through various data sources like company's online support websites. Thus, the paradigm of text-based diagnosis could be viewed as: **text symptom description ➔ troubleshooting documents**.

The effectiveness of this approach is highly variable. Sometimes it is difficult for the user to describe the symptom. Furthermore, manual query is prone to ambiguity and inaccuracy which would lead to inaccurate search results. Like the above example, frequently there are many root causes that could yield the same symptom, and they would have to be distinguished by hand. In summary, to manually compose accurate problem descriptions is a difficult task for users or even professionals in most cases. Moreover, even if an accurate problem description could be made, the search results are a list of relevant or irrelevant documents due to the inherent inaccuracy of information retrieval techniques. For example, searching "The page cannot be displayed" will return thousands of results from a typical web search engine. It is still a very challenging task for users to figure out the real root cause and solution by browsing through the documents. As a result, text-based diagnosis greatly suffers from its inaccuracy and heavy human involvement, even though it is a straightforward method which has been used for a long time.

State-based diagnosis – Low-level system state information has been used extensively in program debugging and problem diagnosis since it can reveal more detailed context which might indicate the source of a problem. Many tools attempt to automate the fault diagnosis task using system states (such as [28][30]). These tools use some mechanisms to detect abnormal system states, and then relate the states to known problems. For example, to solve configuration problems [28] uses various techniques to narrow down the list of candidate root causes, including persistent state differencing, runtime tracing, intersection and statistical ranking. Then configuration roll-back [31] can be applied to fix the problem. The work in [30] further attempts to automatically find the good state to roll-back to. Thus, the general paradigm of state-based troubleshooting is: **system state information ➔ root cause or known problem**.

For state-based diagnosis, the whole diagnosis process is pretty automatic and requires no or very little human involvement. Its accuracy is also good in the case that the abnormal state is correctly identified. In addition, since abnormal state is directly identified, some unknown problems could be detected and solved by this method. Despite these merits, state-based diagnosis methods have several inherent shortcomings. First, accurately isolating abnormal system states is usually non-trivial, considering that a modern computer system contains so many kinds of states. Tools in [28] can only give a candidate set containing tens of "possible" abnormal states in the ideal cases. Second, in many cases, a state by itself cannot tell if it is normal or abnormal. For instance, a specific IP address could be normal in one machine and abnormal in another machine. Third, state-based diagnosis will not work when the root cause is not contained in the collected system states. If a state-based troubleshooter is designed to only collect registry related states, it can do nothing about those problems with file related root causes. While collecting as many kinds of states as possible can alleviate the problem, it is usually unrealistic to collect all or even most of the state information in a complex computer system. Therefore, the generality of this method is not satisfactory. Fourth, most existing state-based diagnosis tools treat states separately and target to build a kind of direct mapping between single state and root cause or known problem. However, problems caused by multiple abnormal states cannot be dealt with by this strategy.

In this paper, we propose a novel **trace-based problem diagnosis** methodology, which relies on the trace of low-level system behaviors to deduce problems of computer systems. Behavior information is trails of various transient events occurring in the system, such as system calls, I/O requests, call stacks, context switches, etc. We intend to identify the correlations between system behaviors and known problems and then use the learned knowledge to solve new coming problems. Thus the new diagnosis paradigm is: **system behavior information ➔ known problem**.

We study the effectiveness of this method with four example cases. We find that with appropriate processing method and learning algorithm, the root cause of a problem can be identified at a considerable accuracy. Also, problems with various root cause sources could be correctly identified, even when the root causes are not contained in the traces. The result strongly supports that there does exist some asso-

ciation between low-level system behavior and high-level problem and the trace-based problem diagnosis is a promising troubleshooting method with good accuracy and generality.

This paper is organized as follows. In Section 2 we describe the design of the automated diagnosis system. We introduce the event tracing component in Section 3 and describe the classifier we use to learn from system behavior and predict root cause in Section 4. In Section 5 we report our observations on system event traces which help us to design noise filtering and canonicalization rules. In Section 6 we evaluate this approach with four problems having diverse root causes. In Section 7 we discuss some questions about the method. We introduce related work in Section 8.

2. SYSTEM DESIGN

This section describes the design of the automatic troubleshooting system which diagnoses problems with event traces collected from user machines. The illustration of the system architecture is shown in Figure 1.

The goal of the system is to minimize user involvement in problem diagnosis. When a user encounters a problem, for example Internet Explorer cannot display a Web page, she only needs to start the troubleshooter that handles the IE display problem. The steps follow are described below.

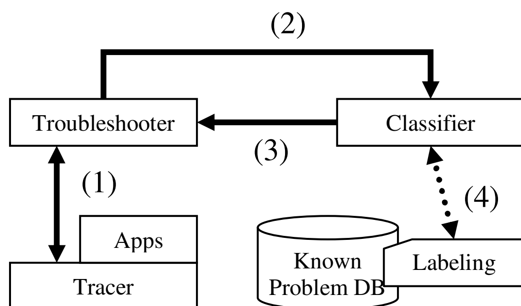


Figure 1. Automatic troubleshooting system
The dotted edge means the step can be done offline.

(1) The troubleshooter will replay the IE browsing operation to reproduce the symptom.[1] At the same time it will start the event tracer to collect the sequence of system events that will be incurred.

(2) Then the trace log is sent to the classifier. After some preprocessing and transformation, the classifier will analyze the input trace and identify the root cause based on its similarity with the previously collected and labeled traces.

(3) The root cause and the corresponding solution will be sent to the troubleshooter, which can present a report of repair instructions to the user or even fix the problem automatically.

Step (4) is responsible for learning the classifier for root cause identification. With a database of known problems and their root causes (from the past accumulation of diagnosis knowledge), a number of event traces can be collected for them offline and labeled with the corresponding root causes. Then the classifier can learned from the event traces and build a prediction model, which is used to forecast the root causes of the traces submitted by end users. It is possible to leverage the user-submitted traces in the learning, which could save some data collection effort.

Although the classifier can be continuously updated if deployed on a support center, it can also (sometimes has to) reside on the user's local machine with a pre-computed prediction model for some frequent problems and networking issues.

In the next two sections we will describe the two main components – tracer and classifier – in detail.

3. TRACER

The tracer will log the sequence of events triggered in the system when the symptom of a problem is being reproduced. The sequence will serve as a representation of the problem and be delivered for analysis and recognition.

There are many kinds of events in a running system and they can occur at various levels. A higher-level event can be an abstraction of some lower-level events. For example, an action on the application's user interface could result in a series of function calls to a dynamic library, which might further ask for system services with system calls. The system calls are finally realized by machine instructions. Tracing events at these levels would require different implementation mechanisms and reveal different degrees of detail. Each level may need diverse effort in obtaining similar coverage of system behavior. And the granularity of behavior characterization would change with the level of observation as well. A series of instructions would tell more about the dynamics in the machine than a function call to a library does. The side effect is that the variation of instruction sequences could also be more significant than function call sequences. Another factor we have considered is the semantics of an event, because we may need to inspect the logged events in the development of the method and a function call is often more intuitive and meaningful for us to understand. Usually, the higher the tracing level is, the richer semantics an event would contain.

Taking the trade-off between granularity and semantic richness into account, we choose to use system call as the type of events to monitor. Since system calls provide core system services and are frequently invoked, at this level most as-

[1] The problems that can not be reliably reproduced are beyond the scope of the paper.

pects of application and system behavior would be covered at a considerable granularity. System calls also have clear semantics about the operating system and thus offer some extent of understandability. In addition it might be one of the most similar event types across popular operating systems, which would be helpful to extend our result to broader environments. There are also known techniques and tools for collecting system calls on various platforms. Therefore, we think it is a good starting point for our study. In the future we plan to try some lower-level events like I/O requests (e.g. network packets) and higher-level events like function calls to various libraries or application-specific events actively generated by developers for the purpose of debugging. Simultaneously investigating events from multiple providers might be feasible as well.

The current tracer collects most of the system calls on Windows XP, which are relevant to various kernel objects like process, thread, registry, file, mutex, semaphore, event,[2] section, access token as well as facilities like security, auditing and local procedure call. It also traces some system calls about Win32 messaging, which constitutes the event-driven model of Windows. For each system call the tracer records the following attributes.

- Sequence number
- Process ID
- Thread ID
- Process name
- Thread name
- System call name
- System call parameters
- System call return value

Sequence numbers denote the order of event occurrences. However, system calls are logged upon exit, so nested calls will appear before the caller. Since this always happens within a thread, the relative order remains deterministic.

Process ID and thread ID are used to distinguish system calls from different tasks. Process name is the image name of the process making the system call. We use the start address of the thread to resolve the thread name. Specifically, we figure out the module[3] containing the thread start address according to the module layout of the process. Then the module name plus the offset relative to the module start address is regarded as the thread name. Process name and thread name are used to seriate the system calls of a trace session in a uniform way, as described in Section 5.1.

System call parameters suggest more specific semantics about a system call. Where possible we always translate a parameter into a meaningful and session-independent form so that it can be compared with each other reasonably. For example, kernel objects can be named and many system calls access kernel objects through handles, so a parameter referring to a kernel object handle will be logged as the object name queried with the handle.

The tracer is implemented as a kernel-mode driver relying on system call hooking techniques [22][25][19] to intercept the system calls. The logging of intercepted events is done through WPP Software Tracing [32] which is a low-overhead mechanism for kernel-mode drivers to log real-time messages. In our experience when the tracer is running on a typical modern machine there is no noticeable performance degradation for most operations. In addition event loss is always reported so we can avoid collecting incomplete traces.

For the reason mentioned in Section 5.1, process/thread beginning and end events are also logged along with system calls by the process/thread creation/deletion callback routines set by the tracer.

The size of a trace log is highly variable, depending on the length of a session (reproduction) and the application. It can generate a log of about 10Mbytes (57K events) for a Web browsing action in Internet Explorer (a browsing button is pressed to the page is fully loaded). However, the current trace format is highly redundant and a compact representation can reduce the size by an order of magnitude (the above log file becomes about 0.7Mbytes after compression). Since trace size will largely determine the efficiency of storage, remote transfer, and analysis, it would be very important to have the option to cut down the amount of information to be logged without sacrificing accuracy. In the meantime, less information to log also means less runtime overhead in tracing. We will discuss this issue in the evaluation part.

4. CLASSIFIER

The classifier is responsible for predicting the root cause (class) of a new trace (test data) based on the previous traces with known root causes (labeled training data). The key component of the classifier is feature representation, i.e. developing effective mathematical form of input data so that different classes can be accurately distinguished by a classification algorithm. We first introduce the n-gram model we use as feature representation of system call sequences and then give some background on the SVM classification algorithm.

4.1 N-gram Based Representation

An n-gram is any N successive symbols from a symbol string. When N=1, it becomes the “bag of symbol” representation. N-gram models were used early for natural language understanding [26] and later text categorization [8]. They are simple and can construct features from sequential data while maintaining its sequential information.

[2] Here event is a kind of kernel object for synchronization or notification between threads in Windows. Event has a more general sense elsewhere in the paper.

[3] In Windows a module is an executable file or dynamic link library. Each process contains one or more modules.

Given a set of system call sequences as training data, we first extract n-grams (features) of all sequences and each of them will represent a dimension of a high-dimensional vector space. Then for any system call sequence (either for testing or training) we represent it as a feature vector of {0, 1} by setting 1 at a component of the vector if the corresponding n-gram is contained in the sequence and setting 0 otherwise. The bit vectors are ready to be used by a classification algorithm for learning and prediction.

4.2 Classification by Support Vector Machines

Support Vector Machines (SVM) is a pattern classification algorithm developed by V. Vapnik [27]. It solves two-class pattern recognition problems based on the *Structural Risk Minimization principle*. Given a training set in the feature space, this method finds the best decision hyperplane that separates the two classes, so that it gives the best expected generalization ability. It has been shown to perform well on high dimensional data sets with small sizes, which is an ideal property for the data types we are dealing with.

Given a data set $D = \{\mathbf{x}_i, y_i\}_{i=1}^{t}$ of labeled examples, where $\mathbf{x}_i \in R^n$ (in our case the bit vectors), $y_i \in \{-1,1\}$, we wish to select, among the infinite number of linear classifiers that separate the data, one that minimizes the generalization error, or at least an upper bound on it. Vapnik showed that the hyperplane with this property is the one that leaves the maximum margin between the two classes. Given a new data point $\mathbf{x}$ to classify, a label is assigned according to its relationship to the decision boundary (hyper-plane), and the corresponding decision function is:

$$f(\mathbf{x}) = sign(\sum_{i=1}^{t} \alpha_i y_i \langle \mathbf{x}_i, \mathbf{x} \rangle - b)$$

From this equation it is possible to see that the α_i associated with the training point $\mathbf{x}_i$ expresses the strength with which that point is embedded in the final decision function. A remarkable property of this alternative representation is that often only a subset of the points will be associated with non-zero α_i. These points are called *support vectors* and are the points that lie closest to the separating hyperplane.

The nonlinear support vector machine maps the input variable into a high dimensional (often infinite dimensional) space, and applies the linear support vector machine in the space. Computationally, this can be achieved by the application of a (reproducing) kernel. The corresponding nonlinear decision function is:

$$f(\mathbf{x}) = sign(\sum_{i=1}^{t} \alpha_i y_i K(\mathbf{x}_i, \mathbf{x}) - b)$$

where K is the kernel function. Some typical kernel functions include polynomial kernel, Gaussian RBF kernel, and sigmoid kernel. For multi-class classification problem, one-against-all scheme [14] can be used.

We choose the widely used libSVM package [9] as the implementation of the algorithm. LibSVM will learn a prediction model from a specified training set and store it as a persistent file which can be loaded by any new instance of the classifier. We use the linear SVM kernel in our experiment, because it is more robust than nonlinear kernel and does not need to tune any parameter.

Though SVM is very efficient in learning on high-dimensional data, the actual computational complexity would still rely on the dimension of the input data. For example with the linear kernel the basic operation would be the inner product of two vectors. In practice we often need to reduce the dimension to make the classification task more efficient, for example applying the noise filtering rule described in Section 5.2.

A classifier is usually measured by its accuracy in predicting unknown data, i.e. the percentage of the correctly predicted data. We use the cross-validation procedure throughout our experiments to prevent overfitting on a limited data set. In a standard k-fold cross-validation we first divide the training data into k partitions and then repeat selecting one partition of data to test with the classifier trained from the remaining data until all data are tested. The cross-validation we use is a slight variation based on this for controlling the size of the training data. Then the accuracy of the classifier will be averaged over all folds of the cross-validation.

5. NOISE FILTERING AND CANONICALIZATION

The section describes our observations on system behavior variation by comparing the system call sequences with sequence alignment. We designed some rules for noise filtering and canonicalization based on these observations.

Trace comparison is an important primitive for understanding system behavior distinctions under different situations (such as times and machines of tracing, root causes of a problem) and hence for characterizing system behavior from multiple traces.

We use sequence alignment [15] as a basic tool to compare traces. A sequence alignment algorithm tries to find the maximal similarity of two sequences. For example, Figure 2 shows the alignment of two strings, where white spaces are inserted to allow noncontiguous matches. Without any knowledge of the application program, this would be a reasonable method we can rely on for comparison.

```
Original     Aligned
[abcefh]     [abc ef h ]
[bcdfghi]    [ bcd fghi]
```

Figure 2. Sequence alignment example

To compare more sequences simultaneously, we use a simplified multiple sequence alignment algorithm based on pair-wise alignment. Its time complexity only increases line-

arly with the number of sequences used, and the effect is close to the optimal alignment according to our datasets.

However, raw traces are not suitable to be compared directly since irrelevant system calls may disturb alignment. For stable comparison, we attempt to identify and eliminate random effects as much as possible and serialize traces in a uniform manner before alignment.

5.1 Uniform Ordering

System calls from different threads can occur in random orders as a result of thread scheduling effect. Therefore, traces cannot be reasonably compared before they are reordered in a uniform way. Thread interleaving effect can be handled by looking at thread ids of system calls. In case of two threads being assigned the same id in a trace session, we can tell them apart using the thread beginning and end events. Processes are separated in the same way. Thus, a trace can be divided into segments each of which represents the activity of a single unique thread.

After segmentation, we sort the system calls by process name and thread name.[4] Within each thread they are ordered by their sequence numbers. Processes/threads with the same name will be ordered by their first occurrence. The first occurrence of a process/thread is defined as the sequence number of the first captured system call of the process/thread. This is suggested by the observation that it is very likely that such processes/threads occur in a non-random order.

With this ordering scheme the system calls generated by the instances of the same logical thread in different traces can be positioned in a way that reveals similarity of different traces as much as possible.

5.2 System Call Variation

Below we report our study on system behavior changes with different aspects of traces in terms of time and machine of trace collection. This study will help us to focus on the essential parts in traces and design noise filtering and canonicalization rules. All the studies are based on the traces collected for the same operation replayed on some normal machines.

5.2.1 Cross-Time

We first study how system behavior changes with time. The traces are first reformatted with the uniform ordering as introduced in Section 5.1. Then we watch for patterns of continuous system calls in different traces through trace comparison. Because the traces containing all kinds of system calls are usually very long (10,000~100,000 calls), which would be hard for our inspection, we divide the traces into pieces with each containing system calls of the same category (like registry, file) and compare respective pieces instead. We noticed the following patterns of system calls when doing the cross-time trace comparison, as illustrated in Figure 3. Five normal traces of the same operation are aligned in the figure, where each column represents a trace, dark area denotes the same system calls as the first trace at the same positions, gray area is the system calls not occurring in the first trace (but probably occurring in other traces), and white area means inserted spaces.

Some system call patterns occur in all traces (category A in Figure 3). This might be the behavior that the system always performs in the situation, though it could lie in some processes other than the known relevant process. Sometimes certain processes known to be noisy also incur such patterns.

Some patterns appear in a minority, half, or majority of the traces (category B). They may be generated due to the slight change in the underlying environment that the process in the context depends on. Since they are not always generated, they are not strongly relevant to the problem.

This leads us to design a *cross-time noise filtering* rule that if a pattern occurs in more than a threshold percentage of the total traces for a root cause, we keep it as a feature and otherwise we regard it as a noise to be discarded.

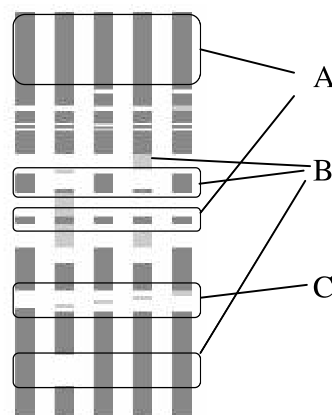


Figure 3. Cross-time system call patterns

Some system calls occurs uniquely in one trace (category C). By comparison we find that sometimes such system calls do not align with each other because their parameters are partially different in each run of the problem. Specifically, many system calls operating on named objects but the names could change each time the call is invoked even though it is doing the same work (which can be inferred from the similarity of its surrounding calls in other traces and also confirmed by the slight difference in most of such names). For example, the system calls in category C of Figure 3 are CreateEvent with event named like \BaseNamedObjects\CTF.ThreadMIConnectionEvent.000001AC.00000000.000002E3, E4, or E5 etc. This problem is referred to as *object name canonicalization* problem. The object names need to be translated into such uniform ones

[4] In practice we found the offset part of a thread name is not very stable, so we remove it and leave the module name only.

so that the corresponding system calls can be properly compared. Currently we have some rules to transform an object name to a session-independent format (while retaining the meaning of the name as much as possible) based on a few such misalignments we discovered.

In addition, there are cases that the object name is the same within a single running session but will change after the application is restarted. For example, we find Internet Explorer always accesses "minpos1400*1050(1).x" (and a series of relevant entries) in registry from a different location in new instances. So we decide to discard the registry path and leave the name part only.

5.2.2 *Cross-Machine*

In this study the traces from the same machine are merged into one after cross-time noise filtering and canonicalization. Then the merged traces from different machines are compared with each other.

Figure 4 shows the alignment of the merged traces from 11 machines. It can be observed that similar patterns appear as in cross-time trace comparison. The distinction may be caused by the different running environment on the machines. Similar to cross-time noise filtering, we also use *cross-machine noise filtering* to control if a pattern like this should be regard as noise. Its definition is the same as the former one.

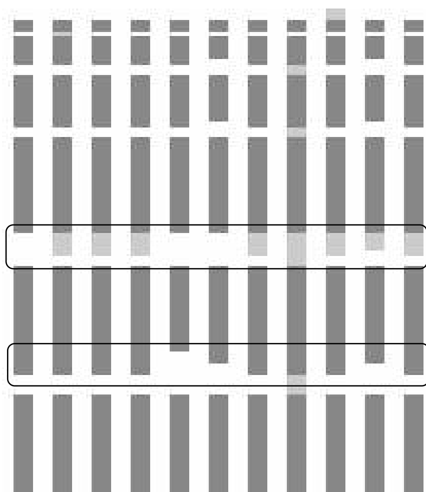


Figure 4. Cross-machine system call patterns

Canonicalization problem also exists here. Some object names can be different on each machine. For example, files of the same purpose can be created at different locations on different machines, which will result in different full path names occurring with system calls. The canonicalization rule we use is simply discarding the path and leaving the file name only. In addition, the tracer also makes the cross-user canonicalization for registry paths at runtime (translating the user-specific prefix "\REGISTRY\USER\S*" to "HKCU" standing for the registry hive of the current user).

6. EVALUATION

In this section we evaluate how accurate our approach identifies the root causes for real problems. We first introduce the four problems we choose for evaluation and how the data are collected. We report the results of cross-time and cross-machine analysis, including the effect of a number of variations that can impact the accuracy. We use cross-time analysis and results as a basis for cross-machine study.

6.1 Problem Selection

There is no existing diagnostic dataset containing system call sequences that we are interested in. Most of today's user feedback mechanisms do not collect system behavior information when users report that they have encountered problems with their machines. Therefore, we need to collect data through fault injection. Before that we first select a set of problems to work on.

Table 1. Selected problems for evaluation

Root cause id	Description
IeDisplay	
BadIP	IP address is invalid.
BadPort	The specified server port is invalid.
BadProxy	The HTTP proxy is invalid.
BadProxyPort	The HTTP proxy port is invalid.
Disable	The network connection is disabled.
NoDriver	The driver of the network adapter is not installed.
NoPage	The page visited does not exist on the server.
NoProtocol	The TCP/IP protocol is not enabled.
Unplug	The network cable is unplugged.
WinSock2	WinSock2 registry key is corrupted.
SharedFolder	
BadIP	IP address is invalid.
Disable	The network connection is disabled.
NoClient	Client for Microsoft Networks is disabled.
NoDriver	The driver of the network adapter is not installed.
NoHost	The host does not exist.
NoPath	The shared path does not exist on the host.
NoPermission	The permission is not enough to access the folder.
Unplug	The network cable is unplugged.
OeOpen	
DbxDamaged	The mailbox file is damaged.
DbxNoPermission	The permission is not enough to access the mailbox file.
DbxReadOnly	The mailbox file is read-only.
IdCorrupted	The registry key corresponding to the user identity is corrupted.
FfDisplay (see IeDisplay)[5]	

Our criteria of deciding whether a problem is suitable for the study are based on its popularity, the diversity of its root causes and the ease of reproducing its symptom. After going through some sources of PC diagnosis knowledge we decided to use four problems for evaluation as described below.

- **IeDisplay**. Internet Explorer cannot display a Web page. There are 10 potential root causes to this symptom.

[5] Firefox actually can detect invalid proxy settings, though it does not distinguish between BadProxy and BadProxyPort. We include the two root causes for it just as an aggressive test of our approach.

- **SharedFolder**. Cannot open a shared folder on the local network. There are eight potential root causes.
- **OeOpen**. Cannot open Outlook Express. There are four root causes.
- **FfDisplay**. Mozila Firefox cannot display a Web page. The root causes are similar to those of IeDisplay.

The first three problems are selected based on the frequency of occurrence recorded in Microsoft PSS (Product Support Service) service request logs. They also have corresponding entries in Windows XP's built-in Help and Support Center. We added the last problem in order to validate the method on a broader set of applications.

Table 1 gives the description of the root causes. It can be seen that the root causes are quite diverse, ranging from the faults in configuration, installation, server, to security. For each problem, we will treat normal status as an additional class to be distinguished, whose "root cause id" will be "Normal."

6.2 Data Collection

We collected the traces by reproducing the problems on a number of daily-used machines. For each problem and each machine we first reconstruct the problem context, make sure the problem does not exist already and then inject the fault into the machine. Next the tracer is started to capture the trace during the symptom reproduction. For example, when collecting traces for the problem IeDisplay-WinSock2 (Internet Explorer cannot display a Web page because of the corrupted WinSock2 registry key), we verify Internet Explorer can display web pages properly and then corrupt the WinSock2 registry key. Next we start the tracer and try browsing a Web page with Internet Explorer. After the symptom shows up, the tracer is stopped. For normal cases we just redo the actions as for other root causes without actually injecting any fault.

As described in Section 2, we require the troubleshooter to reproduce the symptom automatically. Therefore, the training data should be collected in the same way as the testing data. In our experiment the symptom reproduction is performed with the UI automation tool AutoMate [3]. We wrote a replay script for each problem, which will be executed by AutoMate to redo the actions and wait for the symptom to appear. Similarly, our implementation of the troubleshooter will call AutoMate to reproduce the symptom.

When collecting traces for each root cause of IeDisplay and FfDisplay, we launch the browser four times and each time followed by four navigations to a different Web page. The tracer is started before navigation starts and is stopped after the Web page is loaded completely or after an error message shows up. For SharedFolder we select four shared folders and open each folder for four times. For OeOpen Outlook Express is launched eight times. We only use a subset of the traces for each evaluation to reduce any potential dependencies between reproductions as much as possible.

Two sets of data are collected, for single-machine analysis and cross-machine analysis, respectively. The details are mentioned in Section 6.3 and 6.4. All the machines run Windows XP SP2.

6.3 Single-Machine Results

We first focus on how well the method can work on a single machine. In this setting the test data and the training data are from the same machines (but different times). The evaluation is intended as an initial study on the method, which would be helpful when we want to extend it to a broader domain. The results would serve as a baseline as well.

All the data used in the section are collected from five machines across six days for two problems (IeDisplay and SharedFolder). The results are obtained from sixfold cross-validations where four traces are used for each day and the data of three days are taken as training data from the six days in turn.

All the data are processed on a Dual 3.1 GHz Intel Pentium 4 Xeon with 4GB memory running Windows Server 2003. When there is time measurement, we always ensure the load is light before processing.

Threshold. We first investigate the effect of noise filtering threshold on the accuracy of the 1-gram classifier. We will take account of the sequential property of the event traces with higher-gram classifiers in the experiments below.

Figure 5 shows the results of the two problems when we vary the threshold. The accuracy of IeDisplay mostly increases with the threshold. For SharedFolder the accuracy does not change much, though machine2 and machine4 have slight decrease. This is because that the filtering could risk eliminating features that can be discriminative when combined with some other features, though it can remove true noises as well. The effect on the accuracy could be dependent on the problem. In spite of this, increasing the threshold can reduce the average dimension of the feature vectors corresponding to the training data by almost two orders of magnitude. The time needed to carry out the entire cross-validations achieve the similar savings because the evaluation of the linear kernel function of SVM consists of the inner product of two vectors whose complexity depends on the dimension of the input data. The efficiency improvement by dimension reduction would significantly speed up the process of learning the classifier and predicting the class of a new trace, even though at the cost that the accuracy could be sacrificed.

The threshold of the remaining experiments in this section will be fixed to 0.8. Since it seems to be orthogonal to other variables, we can always try to make more optimization from it when necessary.

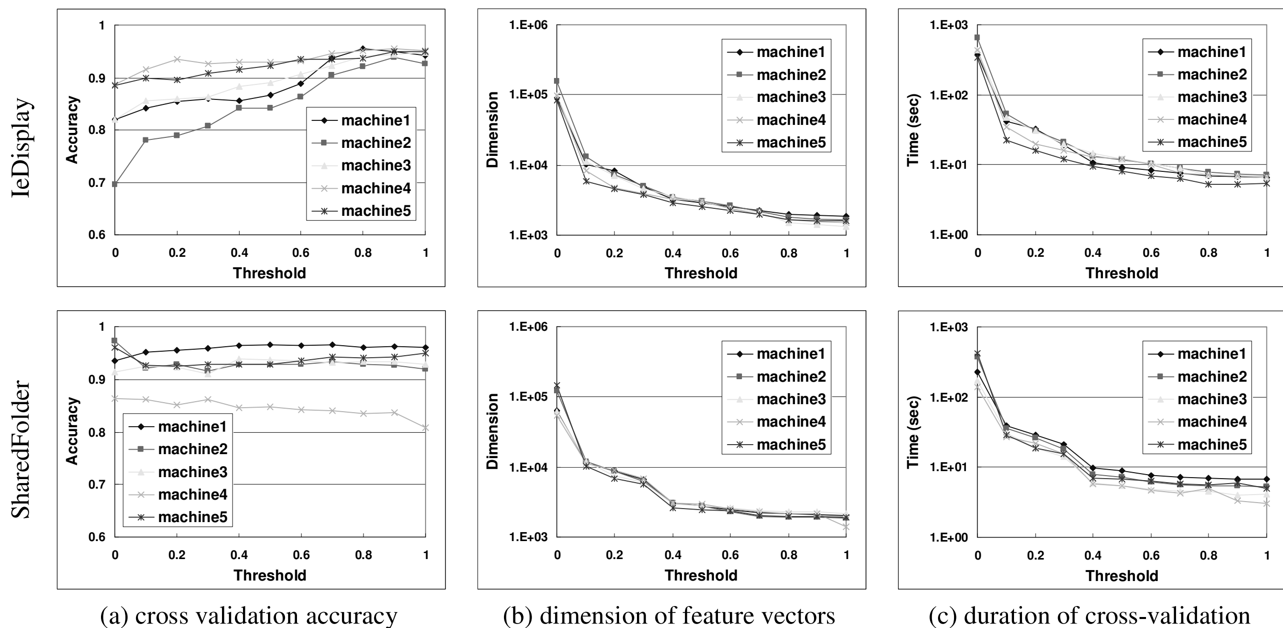


Figure 5. How the accuracy and efficiency of the classifier change with the threshold of noise filtering for single-machine cross-time system call sequences of the two problems

Table 2. Effect of canonicalization on single machines

	IeDisplay		SharedFolder	
	Before	After	Before	After
1	95.45%	95.58%	96.14%	95.83%
2	92.05%	93.18%	92.90%	93.52%
3	94.07%	93.94%	93.36%	93.06%
4	95.20%	95.45%	83.49%	82.41%
5	93.69%	93.56%	94.14%	94.44%

Canonicalization. We apply the canonicalization rules to the data at the preprocessing stage and the results of classification accuracy are shown in Table 2. It can be seen that the difference between the results with and without canonicalization is insubstantial. This might be because the object names are rather stable across time on single machines.

Since canonicalization does not bring any negative impact and canonicalized events are usually more compact, we will include it in the next experiments.

Table 3. Results of 2-gram and 4-gram classification on canonicalized single-machine data

	IeDisplay		SharedFolder	
	2-gram	4-gram	2-gram	4-gram
1	96.34%	93.56%	95.99%	95.06%
2	92.42%	90.78%	91.82%	91.67%
3	93.18%	93.81%	91.51%	91.51%
4	94.57%	94.32%	81.17%	81.17%
5	95.08%	94.82%	92.28%	89.04%

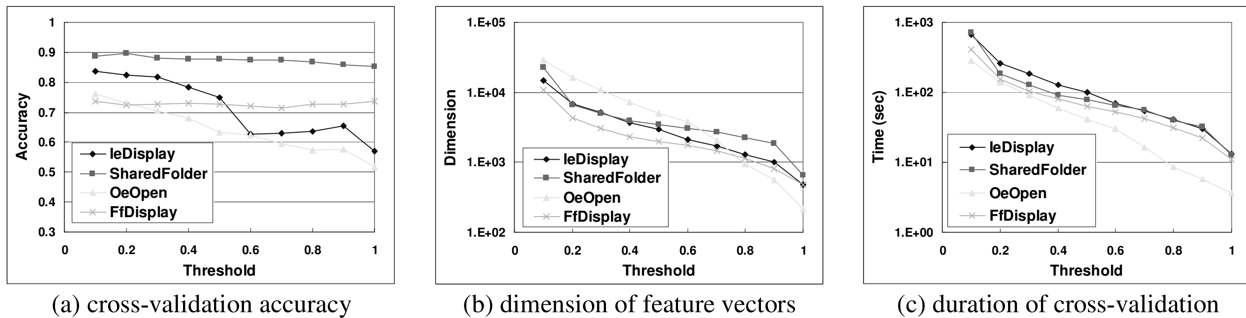
Higher n-grams. 1-gram algorithm considers a whole sequence of events as a set of events and does not use their sequential order. Next we study some higher n-gram classifiers which capture the local order of events in different degrees. The cross-validation results of 2-gram and 4-gram for the data are shown in Table 3. Compared to the 1-gram results in Table 2 the higher-grams do not provide any improvement.

Comparatively, the work on intrusion detection using system call sequences reported patterns with length greater than 1 give useful results [16]. However, the system call records used there are from a single process and have no parameter information. Our system call records are with parameters and from all processes in the system, which would significantly increase the size of the alphabet of the sequences and therefore raise the chance of shorter patterns being discriminative that might be missing by considering long patterns only.

No thread names, parameters and return values. We also want to study how well the classifier can work on traces without logging any information about thread name, parameters and return values. On the one hand, this would reduce the runtime overhead of the tracer and save the storage and transfer overhead. On the other, the availability of thread names and some parameters like those related to object names might be platform-dependent.

Parameters like object names may play a critical role in the classification. Without them, events can become less specific and therefore less discriminative. In this case the sequential information might be valuable. Figure 6 shows the results of different n-gram classifiers for the two problems.

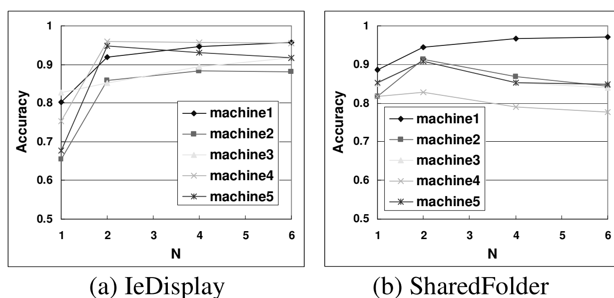
Two kinds of trends can be observed from the results. In some cases the accuracy keeps increasing with N while in



(a) cross-validation accuracy (b) dimension of feature vectors (c) duration of cross-validation

Figure 7. Accuracy and efficiency of the classifier with the increasing of noise filtering threshold for cross-machine system call sequences of the four problems

the other cases the accuracy tops at 2-gram and then does not change much or even goes down with higher n-grams. For example, the accuracy of machine3 on SharedFolder drops from 90.9% of 2-gram to 85.2% of 4-gram. Specifically, from the predicted classes of each root cause we find that two root causes Disable and NoPermission which were properly classified by 2-gram start to be frequently misclassified to NoClient by 4-gram. This means there are some important 2-gram features whose remaining discriminative would depend on the diversity of its containing 4-grams. Therefore, on some machines where such diversity is high, the feature will be missing while on other machines where such diversity is low the feature will be retained. In addition the dissimilarity of the overall trends for the two problems might suggest the optimal n-gram could be problem-specific.



(a) IeDisplay (b) SharedFolder

Figure 6. Accuracy of the data without thread names changes with N-gram classifiers of different N

In summary, we have evaluated a variety of parameters that can affect the result of classification in a single-machine cross-time setting. We found that applying canonicalization and higher n-gram has not brought any obvious positive impact. The accuracy impact of the techniques like noise filtering might be problem-dependent. However, it does improve classification efficiency significantly. Sequential order will help classification when thread name, parameters and return value are not available. For the same problem, the effect of some variations can differ across machines, which may pose a challenge for cross-machine analysis. We will compare these results with the cross-machine results in the next section.

6.4 Cross-Machine Results

Cross-machine evaluation is of realistic importance because it is unlikely a user machine is used for collecting training data. In a cross-machine setting the training data and the testing data are from two disjoint sets of machines. Similar to the single-machine studies, we will measure what accuracy the classification method can achieve under various conditions.

The data used in the section are collected from 20 machines for the four problems. Unless specified, the results in this section are obtained from 10-fold cross-validations where 10 machines are taken in turn from the 20 machines as training machines and four traces are used for each machine. We still start with the 1-gram classifier.

Threshold. Figure 7 illustrates the accuracy, average feature vector dimension, and time for cross-validation change with the noise filtering threshold. We start with threshold 0.1 since threshold 0 will generate vectors with dimension ranging from 160,000 to 540,000 which is too large for the cross-validation to complete in a reasonable time.

Figure 7(a) shows that the accuracy of SharedFolder and FfDisplay keeps almost unchanged, while that of IeDisplay and OeOpen drops with the increasing of the threshold.

Figure 7(b) and (c) show that the dimension of feature vector and the time spent by cross-validation decrease exponentially with the threshold, which makes it appealing to set a proper threshold without sacrificing much accuracy. The threshold in the remaining experiments in this section will be fixed at 0.2.

Table 4. Effect of cross-machine canonicalization

	Before	After
IeDisplay	82.45%	86.77%
SharedFolder	89.72%	89%
OeOpen	73.10%	85.20%
FfDisplay	72.25%	84.61%

Canonicalization. Table 4 compares the classification accuracy before and after canonicalization. We can see that the canonicalization is very effective in improving the cross-machine accuracy compared to the single-machine result.

SharedFolder can achieve equal accuracy before canonicalization, which might be because its traces do no refer to object names as diverse as those of the other three problems. For example, Internet Explorer, Outlook Express and Firefox may access numerous local files whose names can be highly machine-dependent, while accessing a shared folder may not need to do so. The remaining experiments in this section will all use canonicalized data.

Higher n-grams. The results of 2-gram and 4-gram method as shown in Table 5 do not have any obvious improvement over the 1-gram result.

Table 5. Effect of higher n-grams for cross-machine classification

	2-gram	4-gram
IeDisplay	87.64%	87%
SharedFolder	88.19%	88.25%
OeOpen	79.05%	71.55%
FfDisplay	83.52%	82.09%

We also tried another preprocessing method before extracting n-grams. It simply sorts the trace records (already grouped by threads) by the system call name. Stable sort is used to retain the original relative order. The method is based on the observation that the successive identical system calls are likely responsible for the same aspect of a task and therefore could be more coherent. Table 6 summarizes the results with this method. The method does make some improvement over the original results of 2-gram and 4-gram, especially for the OeOpen case. However, the new accuracy still has no notable difference.

Table 6. Effect of higher n-grams with sorting traces by system call name before extracting n-grams

	2-gram	4-gram
IeDisplay	87%	88%
SharedFolder	88.58%	88.72%
OeOpen	82.50%	80.60%
FfDisplay	85.16%	83.14%

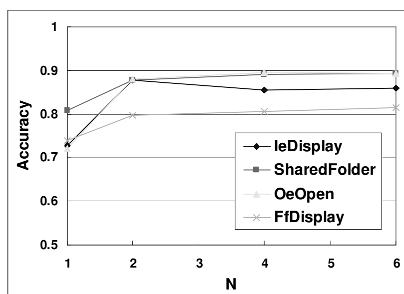


Figure 8. Cross-machine accuracy of the trace data without thread names, parameters and return values using various n-gram classifiers

No thread names, parameters and return values. The result of these data classified with various n-gram classifiers is given in Figure 8. It can be observed that after making some improvement from 1-gram by 2-gram, the n-gram only yields slight changes of accuracy with greater N, which might suggest higher n-gram only capture marginal difference between root causes.

Convergence. One important question related to cross-machine study is how many machines are needed for a classifier to achieve an optimal accuracy, which will determine the effort of data collection for the classifier to be useful. In practice we may only need to obtain a sufficiently high accuracy. Intuitively the accuracy improvement with more machines can be diminishing. This is supported by Figure 9, where (a) is based on the results of the 1-gram classifier (on full trace record) and (b) is based on the 2-gram classifier which work on the data with trace record containing process name and system call name only. Each data point corresponds to the 10-fold cross-validation accuracy with the specified number of machines used for training and the rest ones used for testing. Here, four traces are used for each machine. As can be seen, all the four problems follow the similar trend that the accuracy converges to a value and becomes steady even when more training machines are added. With 10 machines for training, the accuracy is usually close to optimal.

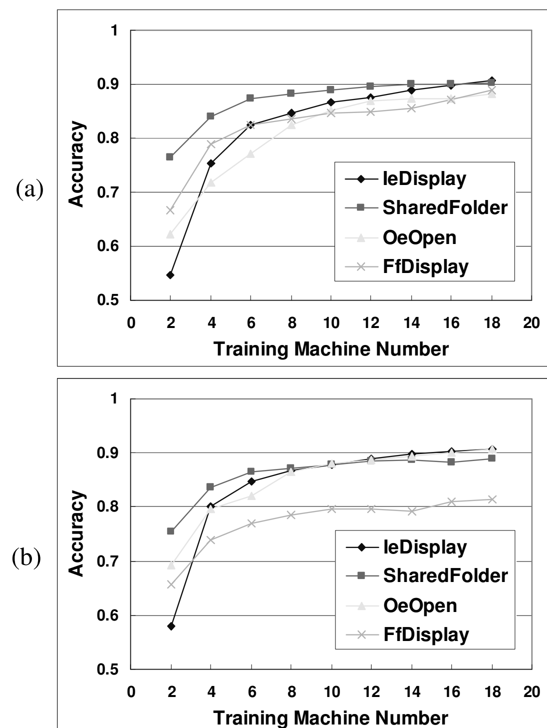


Figure 9. Accuracy grows more and more slowly with increasing number of machines used for training

The highest accuracy the two methods can achieve in this convergence test is listed in the following table. Comparing the highest accuracy of the two classifiers, we can see that the first one gives slightly better results, which is attributed

to the relatively low accuracy on FfDisplay of the second classifier.

	IeDisplay	SharedFolder	OeOpen	FfDisplay
(a)	90.68%	90.28%	88.25%	88.98%
(b)	90.68%	89.03%	90.75%	81.48%

To summarize, our main findings in this section are: (1) Canonicalization is very effective on cross-machine classifications when there are parameters containing object names in trace records; (2) 1-gram classifier is usually good enough for such data. Using higher n-grams does not make notable improvement; (3) when only process names and system call names are available in traces higher n-gram classifiers play a critical role in providing competitive accuracy with (2); (4) The average accuracy converges with more machines used for training. Overall, we find the two methods we investigated can achieve considerably high accuracy.

7. DISCUSSION

In this section we discuss some fundamental questions about this approach.

7.1 Error Reporting by Programs

A direct argument against our post-priori approach is that if it is possible to distinguish the root causes of a problem from the running behavior of the program, the developer should be able to change the program instead to handle the probably overlooked error and give more detailed report. However, it may not always be possible for the program to determine the root cause of an error and its resolution. And, sometimes the error detection and handling may reside in another component beyond the control of the component making use of it. Furthermore, for some low-level components shared by high-level applications (such as networking), handling of their errors in each application would result in many duplications of the same code. In addition, our method can support already released products without the need to update deployed instances.

Another argument against using system behavior information to characterize problems is that it could be more accurate and direct to use program-generated signatures such as error codes to make correlations to problem knowledge. Besides the analogous counter-arguments mentioned above, the decision on where to generate signatures would also require effort of developers, especially for those problems not caused by system errors. In addition, we also plan to apply this method to performance troubleshooting in which root causes are even harder to detect within programs.

7.2 Automatic Symptom Reproduction

We use automatic symptom reproduction not only for user convenience but also to restrict the way reproduction occurs. The user can browse a Web page in many ways, such as click a hyperlink, type a URL, go to the homepage, etc. Each way of reproduction may have different low-level consequences though the high-level symptom could be the same. Moreover, the difference of the consequence could depend on root causes. Instead of enumerating all possible ways of reproduction and take them into account in data collection, we believe using a restricted but automatic reproduction is more reasonable and hopefully the accuracy could also be more stable. Of course the overhead is that an automation script needs to be authored for each problem and sometimes it is also non-trivial to reliably reproduce a symptom. The good effect is that training data can be more readily obtained. And, where such automation is not possible, we can still resort to manual reproduction.

8. RELATED WORK

8.1 Automated Diagnosis

Perhaps the work closest to ours in spirit is [20]. The authors argue that a global knowledgebase associated with rich system information will help to automate diagnosis process. Though several types of automatically generated data are proposed, there is no data analysis method and quantitative experiment showing how the data can be useful for diagnosis. [13] presents a method for generating signatures from system states to help identify recurrent problems and leverage existing diagnosis knowledge. In this paper we focus on using system event traces to help identify root causes of known problems, but we believe our study would also be valuable in advocating other behavior information. Our goal of automated diagnosis is also similar to Autonomic Computing [17] that attempts to make system self-healing by detecting, diagnosing, and repairing problems automatically.

[5] attempts to automate the diagnosis process that is usually performed by human experts: system health monitoring and error detection, component sanity checking, and configuration change tracking. The authoring of the detection and checking rules and logics would require the knowledge of human experts. For example, deciding if the Outlook Express mailbox file is damaged (i.e. DbxDamaged in Table 1) would need the same level of understanding of the mailbox file format as the developers have.

8.2 Learning from System Behavior

Pinpoint [10][11] employs statistical learning techniques to diagnose failures in a Web farm environment. After the traces with respect to different client requests are collected, some data mining algorithms are applied to determine the components most relevant to a failure. The main difference from our work is that Pinpoint intends to recognize what part of the existing traces contributes most to a failure. But our goal is to predict what class of failure a new trace belongs to from the knowledge of the failure categories of prior traces.

Magpie [7][6] aims to analyze performance of distributed systems. A behavior model annotated with resource usage information is constructed after the event traces are clus-

tered by edit distance, which can then be used to understand where the performance bottlenecks are. On the contrary, the traces in our work are used for classification instead of clustering.

Aguilera et al. [1] attempts to locate the node of performance bottlenecks in a distributed system with only inter-node communication traces so as to avoid application-level instrumentation required by the above methods. The main work there is to infer the causal paths in the system from such traces such that the node causing extraordinary delay can be detected.

Cohen et al. [12] also uses learning classifier to make prediction of compliance with performance objectives and help analyze the cause of the violation in Web server systems. However, the input data used there are statistical metrics summarizing the system behavior or state rather than the behavior itself as used in our approach. Another difference is that the target of our classification is the root cause of the problem among many possible candidates, while [12] uses classifiers to make binary decision on whether a criterion might be violated and determine the root cause from the induced model of dependencies between the metrics. Similar to our work, the candidate root causes would be limited, in that case to the observed metrics. Finally, the high-level goal of automating diagnosis by learning from observations of the system with statistical methods is common.

Some host-based intrusion detection systems (IDS) use sequences of system calls [16][18][29] (or a subset [2]) as well. Besides having the common theme of taking advantage of past system behaviors to recognize new behavior as our method, there are a number of differences. First, the main objective of IDS is to detect anomalous behavior resulting from external, probably unknown attacks. An attack can happen at any moment, so it should be detected as soon as possible. The goal of this paper is to distinguish different kinds of abnormal behaviors caused by various unintended but known faults in a cooperative environment. Here, it is known (or assumed) that the application has already behaved abnormally and what caused it is of more interest. In addition, the abnormal behavior usually happens only when the user requests a specific service from the application. Second, training data used in IDS are usually collected from normal executions of the target application. In this paper training data are created from various faulty runs as well as normal runs of the application. Third, the difference in the goals and input data also makes the algorithms used by the two methods different, although the model used to represent the input data, that is, local pattern of system calls, is similar. In IDS a model of the application is built first and then new system calls are checked for consistency with the model. Since our scheme can be formulated as a classification problem, we choose to use the popular classification algorithm SVM. Fourth, false positive rate is an important metric to evaluate IDS. For problem diagnosis we pay attention to accuracy of fault prediction instead because it is not the responsibility of our system to tell if the application is normal or not. However, in the paper we did include normal behavior among other faults in the evaluation and the results show that the normal traces can be recognized accurately as well.

8.3 Other Fault Localization Techniques

[4][21] exploit program running traces to localize bugs by comparing an error trace with correct traces. The idea can be applied in this scenario to isolate the component that is relevant to the deviation of the trace from a normal trace. Since the traces are usually obtained in a black box way, the granularity of isolation would depend on how detailed the system running behavior is revealed. Furthermore, the success of the method still relies on the assumption that the fault lies exactly at the branching point of an abnormal trace, which does not hold when the control flow change is not caused by an immediate state change.

Fault localization in computer networks has a close goal to find the root cause from a set of observed symptoms [24]. The codebook technique [33] is similar to our classification approach in exploiting the correlation of the observed events and the root causes. However, the fault localization there requires a prior specification of causality graph which requires the knowledge of dependencies among system components. In our approach we use the passively and automatically monitored events to build the correlation.

9. CONCLUSION

We have proposed a new method to diagnose known problems with system behavior information. Specifically, we correlate traces of system calls captured during symptom reproductions to problem root causes and apply statistical learning technique on traces and correlations to predict root causes of new occurrences of problems with their corresponding traces in context.

We have evaluated our approach with four problems with diverse root causes and find it can achieve considerable accuracy. With the noise filtering and canonicalization rules devised based on the observations of system call change patterns, we experimented with various methods and options on the traces collected from a number of real machines and the resulting accuracy of root cause detection is nearly 90%.

We think our approach is a necessary step to enable automated diagnosis of known problems. The system information in the problem context should not be missed in the diagnosis process. By mining these data with statistical methods, we can draw useful features that would significantly improve the efficiency of today's diagnosis process.

Besides system calls chosen in this paper, I/O requests such as network communications are also an important aspect of system behavior. We plan to extend the tracer to capture additional kinds of events and evaluate their contribution to diagnosing other problems with our approach.

Currently, the symptom of the problem needs to be reproduced before the root cause detection. Sometime it is not convenient for the user or it may not be possible. The step can be avoided with an always-on tracer. However, the challenge then would be pinpointing the most relevant piece of the trace.

10. ACKNOWLEDGEMENTS

We would like to thank Hu Chen for collecting trace data in the earlier experiments and Yankai Xu for writing automatic data collection scripts. We are also grateful to our shepherd, Leendert Van Doorn, and the anonymous reviewers for their helpful comments.

11. REFERENCES

[1] M.K. Aguilera, J.C. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In the 19th ACM Symposium on Operating Systems Principles, October 2003

[2] F. Apap, A. Honig, S. Hershkop, E. Eskin, and S.J. Stolfo. Detecting Malicious Software by Monitoring Anomalous Windows Registry Accesses. In Proceedings of the Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002). Zurich, Switzerland, October 2002

[3] AutoMate. http://www.unisyn.com/automate/

[4] T. Ball, M. Naik, and S. Rajamani. From Symptom to Cause: Localizing Errors in Counterexample Traces. POPL 2003

[5] G. Banga. Auto-Diagnosis of Field Problems in an Appliance Operating System. USENIX Annual Technical Conference, San Diego, California, USA, June 2000

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using Magpie for Request Extraction and Workload Modelling. 6th Symposium on Operating System Design and Implementation (OSDI), December, 2004

[7] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie: Online Modelling and Performance-aware Systems. 9th Workshop on Hot Topics in Operating Systems, 2003

[8] W.B. Cavnar and J.M. Trenkle. N-gram Based Text Categorization, Proceedings of the Third Annual Symposium on Document Analysis and Information Retrieval, April 1994

[9] C.-C. Chang and C.-J. Lin. LIBSVM: a library for support vector machines. September 2002. Available at http://www.csie.ntu.edu.tw/~cjlin/papers/libsvm.pdf

[10] M. Chen, A. Accardi, E. Kiciman, A. Fox, D. Patterson, and E. Brewer. Path-based Failure and Evolution Management. USENIX/ACM Symposium on Networked Systems Design and Implementation, San Francisco, CA, March 2004

[11] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem Determination in Large, Dynamic Systems. International Conference on Dependable Systems and Networks, IPDS track, Washington, DC, June 2002

[12] I. Cohen, J. Chase, M. Goldszmidt, T. Kelly, and J. Symons. Correlating Instrumentation Data to System States: A Building Block for Automated Diagnosis and Control. 6th Symposium on Operating Systems Design and Implementation (OSDI '04), December 2004

[13] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, Indexing, Clustering, and Retrieving System History. In the 20th ACM Symposium on Operating Systems Principles, October 2005

[14] T.G. Dietterich and G. Bakiri. Error-correcting output codes: a general method for improving multiclass inductive learning programs, in the proceedings of AAAI-91, pages 572-577. AAAI press / MIT press, 1991

[15] D. Gusfield. Algorithms on Strings, Trees and Sequences: Computer Science and Computational Biology. Cambridge University Press, January 1997

[16] S. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion Detection Using Sequences of System Calls. Journal of Computer Security, Vol. 6, pp. 151-180, 1998

[17] J.O. Kephart and D.M. Chess. The Vision of Autonomic Computing. IEEE Computer, January 2003

[18] W. Lee and S. Stolfo. Data Mining Approaches for Intrusion Detection. In Proceedings of the Seventh USENIX Security Symposium, San Antonio, TX, January 1998

[19] J.R. Lorch and A.J. Smith. The VTrace Tool: Building a System Tracer for Windows NT and Windows 2000. MSDN Magazine, October 2000

[20] J.A. Redstone, M.M. Swift, and B.N. Bershad. Using Computers to Diagnose Computer Problems. 9th Workshop on Hot Topics in Operating Systems (HotOS IX), Lihue, Hawaii, May 2003

[21] M. Renieris and S. Reiss. Fault Localization with Nearest Neighbor Queries. ASE 2003

[22] M. Russinovich and B. Cogswell. Windows NT System Call Hooking. Dr. Dobb's Journal, January 1997

[23] D.A. Solomon and M.E. Russinovich. Inside Microsoft Windows 2000, 3rd Edition. Microsoft Press, September 2000

[24] M. Steinder and A.S. Sethi. A Survey of Fault Localization Techniques in Computer Networks. Science of Computer Programming, Special Edition on Topics in System Administration Vol. 53, 2 (Nov. 2004), pp. 165-194

[25] Strace for NT. http://www.bindview.com/Services/RAZOR/Utilities/Windows/strace_readme.cfm

[26] C.Y. Suen, N-Gram Statistics for Natural Language Understanding and Text Processing., IEEE Trans. on Pattern Analysis and Machine Intelligence, Vol. 1, No. 2, April 1979

[27] V. Vapnik. Principles of risk minimization for learning theory. In D.S. Lippman, J.E. Moody, and D.S. Touretzky, editors, Advances in Neural Information Processing Systems 3, pp. 831-838. Morgan Kaufmann, 1992

[28] Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, H.J. Wang, C. Yuan, and Z. Zhang. STRIDER: A Black-box, State-based Approach to Change and Configuration Management and Support. Proc. Usenix LISA, pp. 159-172, Oct. 2003

[29] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: Alternative data models. IEEE Symposium on Security and Privacy, 1999

[30] A. Whitaker, R.S. Cox, and S.D. Gribble. Configuration Debugging as Search: Finding the Needle in the Haystack. 6th Symposium on Operating System Design and Implementation (OSDI), December, 2004

[31] Windows XP System Restore. http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwxp/html/windowsxpsystemrestore.asp

[32] WPP Software Tracing. http://www.microsoft.com/whdc/devtools/tools/EventTracing.mspx

[33] A. Yemini and S. Kliger. High Speed and Robust Event Correlation. IEEE Communication Magazine 34,5 (May 1996), 82-90