RC24672 (W0810-098) October 16, 2008
Computer Science

IBM Research Report

Automatic Software Fault Diagnosis by Exploiting Application Signatures

Xiaoning Ding[1], Hai Huang[2], Yaoping Ruan[2], Anees Shaikh[2], Xiaodong Zhang[1]

[1]Ohio State University

[2]IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Cambridge - Haifa - India - T. J. Watson - Tokyo - Zurich

Automatic Software Fault Diagnosis by Exploiting Application Signatures

Xiaoning Ding[1], *Hai Huang*[2], *Yaoping Ruan*[2], *Anees Shaikh*[2], *Xiaodong Zhang*[1]
[1] *Ohio State University* [2] *IBM T. J. Watson Research Center*

Abstract

Application problem diagnosis in complex enterprise environments is a challenging problem, and contributes significantly to the growth in IT management costs. While application problems have a large number of possible causes, failures due to runtime interactions with the system environment (e.g., configuration files, resource limitations, access permissions) are one of the most common categories. Troubleshooting these problems requires extensive experience and time, and is very difficult to automate.

In this paper, we propose a black-box approach that can automatically diagnose several classes of application faults using applications' runtime behaviors. These behaviors along with various system states are combined to create signatures that serve as a baseline of normal behavior. When an application fails, the faulty behavior is analyzed against the signature to identify deviations from expected behavior and likely cause.

We implement a diagnostic tool based on this approach and demonstrate its effectiveness in a number of case studies with realistic problems in widely-used applications. We also conduct a number of experiments to show that the impact of the diagnostic tool on application performance (with some modifications of platform tracing facilities), as well as storage requirements for signatures, are both reasonably low.

1 Introduction

Since the advent of the notion of "total cost of ownership" in the 1980s, the fact that IT operation and management costs far outstrip infrastructure costs has been well-documented. The continuing increase in IT management costs is driven to a large extent by the growing complexity of applications and the underlying infrastructure [6]. A significant portion of labor in these complex enterprise IT environments is spent on diagnosing and solving problems. While IT problems that impact business activities arise in all parts of the environment, those that involve applications are particularly challenging and time-consuming. In addition, they account for the majority of reported problems in many environments and across a variety of platforms [12].

Many factors can cause incorrect application behavior, including, for example, hardware or communication failures, software bugs, faulty application configurations, resource limitations, incorrect access controls, or misconfigured platform parameters. Although some of these are internal to applications, i.e., bugs, failures are more commonly caused when an application interacts with its runtime environment and encounters misconfigurations or other types of problems in the system [22]. Troubleshooting these problems involves analysis of problem symptoms and associated error messages or codes, followed by examination of various aspects of the system that could be the cause. Application programmers can leverage signal handlers, exceptions, and other platform support to check for and manage system errors, but it is impossible to anticipate all such failures and create suitable error indications [7]. As a result, solving these application problems requires a great deal of experience from support professionals and is often ad-hoc, hence it is very difficult to automate this process.

In this paper, we present a black-box approach to automatically diagnose several types of application faults. Our system creates a *signature* of normal application behaviors based on traces containing an extensive set of interactions between the application and the runtime environment gathered during multiple runs (or for a sufficiently long run). When an application fault occurs, we compare the resultant trace with the signature to characterize the deviation from normal behavior, and suggest possible root causes for the abnormal operation. Using output from our analysis, a system administrator or user can significantly reduce the search space for a solution to the problem, and in some cases pinpoint the root cause precisely.

We represent an application's runtime behaviors using a variety of information, including its invocation context (e.g., user id, command line options), interactions with the platform during execution (e.g., system calls, signals), and environment parameters (e.g., environment variables, `ulimit` settings, shared library versions). Our approach makes extensive use of the `ptrace` facility [8] to collect system call and related information, and other interfaces to gather additional data. Traces containing such information are created during application runtime. After observing multiple runs of an application, information from these traces are summarized (into signatures) and stored in a signature bank. If the application misbehaves, normal behavior of the application stored in the signature bank is compared with the faulty execution trace to find the root cause.

We evaluate the effectiveness of our tool using a series of real problems from three popular applications. Our case studies show that the tool is able to accurately diagnose a number of diverse problems in these applications, and its accuracy can be improved as our tool observes more traces to increase the number (and diversity) of normal execution paths reflected in the application signatures. For each of the applications we also perform detailed evaluations of the time and space overhead of our approach, in terms of the application response time degradation due to trace collection, and the storage needed to store trace data and signatures. Our initial results showed that the time overhead is very noticeable for the applications we tested, up to 77% in the worst case using standard tracing facilities. However, with some modifications and optimizations, we can reduce this to less than 6%, which is a promising indication that this tool can be used in production environment. In terms of space, we observe that signatures grow to nearly 8MB in some cases, which is quite manageable for modern storage systems. Moreover, the space dedicated to traces and signature data can be controlled according to desired trade-offs in terms of diagnosis accuracy or application importance.

A precise definition of an application *signature* is given in Section 2. In Section 3, we describe the toolset we have implemented to automate the collection of trace information, construction of signatures, and analysis of faulty traces to diagnose application problems. In Section 4, we describe several case studies in which we apply the tool to diagnose realistic application problems in a Linux environment. In Section 5, we propose a technique of optimizing ptrace to significantly reduce the performance overheads incurred by trace collection. In Section 6, we discuss related works. Limitations of this work are discussed in Section 7, and finally we conclude the paper in Section 8.

2 Application Signatures

Our approach heavily relies on our ability to capture applications' various runtime behaviors (ingredients of a signature), and using which to differentiate normal behaviors from abnormal ones. These runtime behaviors can be largely captured by recording how an application interacts with the external environment. In the following sections, we describe how to capture application's runtime behaviors and how they can be used for building a signature, which can be more easily applied for diagnosing application problems than the raw runtime behaviors.

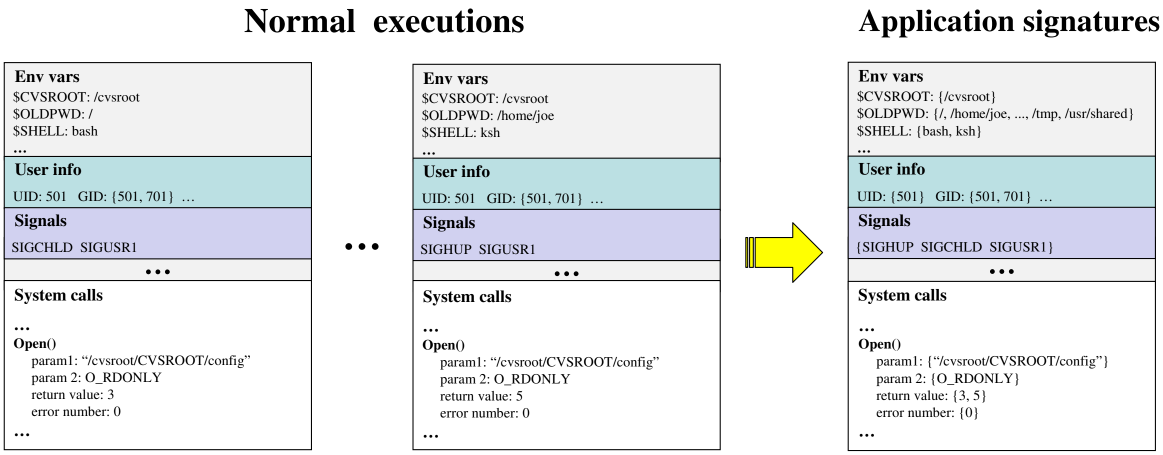
2.1 Capturing Application Behaviors

An application interacts with its external environment through multiple interfaces. A major channel is through system calls to request hardware resources and interact with local and remote applications and services. By collecting and keeping history information on system calls, such as call parameters and return values, runtime invariants and semi-invariants can be identified[1]. Attributes that are invariant and semi-invariant are important in finding the root cause of a problem, as we will see later.

Factors that have an impact on an application's behavior can be mostly captured via information collected from system calls. However, there are some factors that can influence an application's behavior without ever being explicitly used by the application (and therefore, cannot be captured by monitoring system calls.) For example, resource limits (ulimit), access permission settings (on executables and on users), some environment variables (e.g., LD_PRELOAD), etc. cannot be observed in the system call context, but nevertheless, have important implications on applications' runtime behaviors. Additionally, asynchronous behaviors such as signal handling and multi-processing cannot be captured by monitoring system calls, and yet, they are intrinsic to an application's execution behavior. Therefore, to have a comprehensive view of an application's behavior, we collect the following information.

- **System call attributes:** we collect system call number, call parameters, return value, and error number. On a number of system calls, we also collect additional information. For example, on an *open* call, we make an extra *stat* call to get the metadata (e.g., last modified time and file size) of the

[1] Invariants are attributes with a constant value, e.g., when an application calls *open* to read its configuration file, the name of the file, given as a parameter to the call, is almost never changed. Semi-invariants are attributes with a small number of possible values, e.g., the return value of the *open* call normally returns any small positive integer but does not have to be a fixed number.


Normal executions

Env vars
$CVSROOT: /cvsroot
$OLDPWD: /
$SHELL: bash
...
User info
UID: 501 GID: {501, 701} ...
Signals
SIGCHLD SIGUSR1
• • •
System calls
...
Open()
param1: "/cvsroot/CVSROOT/config"
param 2: O_RDONLY
return value: 3
error number: 0
...

• • •

Env vars
$CVSROOT: /cvsroot
$OLDPWD: /home/joe
$SHELL: ksh
...
User info
UID: 501 GID: {501, 701} ...
Signals
SIGHUP SIGUSR1
• • •
System calls
...
Open()
param1: "/cvsroot/CVSROOT/config"
param 2: O_RDONLY
return value: 5
error number: 0
...

Application signatures

Env vars
$CVSROOT: {/cvsroot}
$OLDPWD: {/, /home/joe, ..., /tmp, /usr/shared}
$SHELL: {bash, ksh}
...
User info
UID: {501} GID: {501, 701} ...
Signals
{SIGHUP SIGCHLD SIGUSR1}
• • •
System calls
...
Open()
param1: {"/cvsroot/CVSROOT/config"}
param 2: {O_RDONLY}
return value: {3, 5}
error number: {0}
...

Figure 1: **A simple example to show how an application signature is built from its runtime behaviors.**

opened file. Or, on a *shmat* call, we make an extra *shmctl* call.

- **Signals:** we collect the signal number and represent information collected during signal handling separately from the synchronous part of the application. This is discussed along with how to handle multiprocess applications in more detail later.
- **Environment variables:** we collect the name and value of all the environment variables at the application startup time by parsing the corresponding environ file in /proc.
- **Resource limits:** we collect ulimit settings and other kernel-set parameters (mostly in /proc) that might have impacts on applications.
- **Access control:** we collect the UID and GID of the user and access permissions of the application executables.

This is not meant to be a complete list, but from our experience working in the system administration field, we believe this is a reasonable starting point and the information that we are collecting here will be useful in diagnosing most problems. In the next section, we describe how the collected information is summarized to build a signature.

2.2 Building Application Signatures

We use a simple example in Figure 1 to illustrate how signatures are constructed from application's runtime behaviors. These runtime behaviors can be broken down to their elemental form as *attributes*, e.g., an environment variable is an attribute, uid is an attribute, and each of the parameters in a system call and its return value is an attribute. Distinct values that we have seen for an attribute are stored in a set structure, which is what we called a *signature* of that attribute. For example, the environment variable $SHELL in the above example changed from "bash" to "ksh" between runs. Therefore, the signature of the $SHELL attribute is represented as a set {"bash", "ksh"}. On the other hand, the `errno` of the *open* call in the above example is always zero. Therefore, its signature is simply a set with one item {"0"}.

Some attributes always change across runs (i.e., normal runtime variants), e.g., PID, temporary file created using *mkstemp*, the return value of *gettimeofday*, etc. These are not useful attributes that we can leverage during problem diagnosis. We identify non-useful runtime variants with one-sample Kolmogorov-Smirnov statistical test (KS-test) [5]. It is a "test of goodness of fit" in statistics and is often used to determine if values in two datasets follow same distribution. KS-test computes a distance, called D statistic, between the cumulative distribution functions of the values in two datasets. KS-test provides a critical value D_α for a given significance level α, which represents the probability that the two datasets follow same distribution but KS-test determines they are not. If the D statistic is greater than D_α, the two datasets can be considered to have different distributions with the possibility of $1 - \alpha$. Obviously, if an attribute is a runtime invariant (i.e. only one value in its signature), we do not perform KS-test on it.

We apply the KS-test to test only attributes with more than one distinct value. For such attributes, we monitor the changes in their signature size (i.e., the set size). Thus, we have a series of set sizes (as many as the collected values) for that attribute across runs. We then hypothesize that the attribute is a runtime variant, and its value changes in each run. This hypothesis will generate another series of set size, and the set contains all distinct values. We then use the KS-test to determine if the distributions of the set sizes in the two series are same. If so,

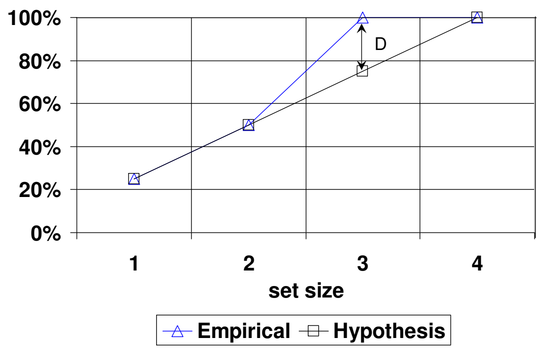


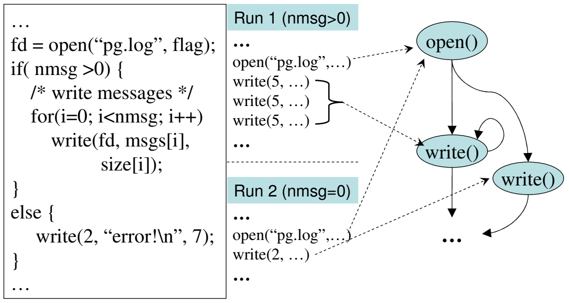
Figure 2: **A simple example illustrating KS-test.**

the attribute is considered to be a runtime variant. As an example, we assume we have collected 4 values (and 3 are distinct) for an attribute. When we build its signature by merging these 4 values into a set one by one, we obtain 4 set sizes (1, 2, 3, 3)—the last 2 values are the same and did not increase the size of the set. If the attribute is a runtime variant, we expect the set sizes are (1, 2, 3, 4). We use KS-test to compare the cumulative distribution functions of the set sizes as shown in Figure 2. For this example, the D statistic is 0.25. If we set the significance level α to 10%, the critical value D_α of KS-test is 0.564. As the D statistic is less than D_α, KS-test determines that the difference between the two distributions is not significant, thus, we consider this attribute as a runtime variant.

When an application fault arises, we compare the values of the attributes collected in the faulty execution against the values in their signature. Attributes that are considered as runtime ~~variant~~ are not used in comparison. If a value of an attribute cannot be found in its signature, the attribute is considered to be abnormal and is identified to be a possible root cause. With the signatures built in Figure 1, if a process receives an extra signal *SIGXFSZ* in a faulty execution, which cannot be found in the signal signature, the signal can be identified to be abnormal. According to the semantics of the signal, only a process writing a file larger than the maximum allowed size receives this signal. Thus one can find the root cause by checking the size of the files used by the application. Since each file accessed is monitored, the over-sized file can be easily identified using our tool.

2.3 Building Signatures for System Calls

We have shown the method of building signatures for attributes in the previous section. However, building signatures for attributes in system calls, e.g., parameters, return value, error number, is not as simple. Before attributes in a system call can be built into signatures, we first need to find other invocations of this system call that are also invoked from the same location within the target application, either in the same run or in a previous run. However, this is a very difficult task when trying to find these correlated system calls among hundreds of thousands of system calls that are collected.



```
…
fd = open("pg.log", flag);
if( nmsg >0) {
   /* write messages */
   for(i=0; i<nmsg; i++)
      write(fd, msgs[i],
            size[i]);
}
else {
   write(2, "error!\n", 7);
}
…
```

```
Run 1 (nmsg>0)
…
open("pg.log",…)
write(5, …)
write(5, …)
write(5, …)
…
Run 2 (nmsg=0)
…
open("pg.log",…)
write(2, …)
…
```

Figure 3: **A sample system call graph built for the system calls shown in the middle.** The program is shown on the left.

To understand the difficulty, we use an example shown in Figure 3. In this snippet of code, there are two *write* calls. Either one or the other will be invoked, depending on the value of `nmsg`, but not both. It makes no sense for us to merge the attributes of the first *write* call with those of the second *write* call when generating signatures, as these two *write* calls perform very different functions. The first is to write messages to a file, and the second is to print an error message to `stderr`. Therefore, attributes of the first *write* will only be merged with those of other invocations of the first *write*, i.e., within the `for` loop. One can imagine how difficult it would be to differentiate the first *write* call from the second when looking at a trace of a flat sequence of system calls as shown in Figure 3.

We address this problem by converting a flat sequence of system calls to a graph representation, which we called a *system call graph*. Each node in the graph represents a unique system call in the target application, and the edges are uni-directional and representing the program execution flow from one system call to another. The right part of Figure 3 shows such an example graph, where the two *write* calls are clearly differentiated. As we can see, only those system calls invoked from the same place within the target application are collapsed into the same node, e.g., the *open* from the two runs and the 3 invocations of the first *write* call in the `for` loop from the first run. Attributes associated with each system call are appended to the node in the graph the system call corresponds to, as shown in Figure 4.

A vital step in the construction of the system call graph is to collapse system call invocations that are invoked from the same location in the program to a single node in the graph, either within a single run or across multiple

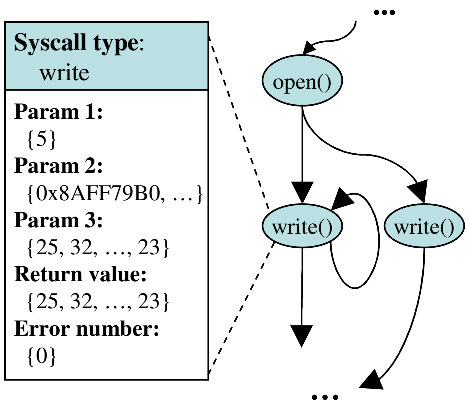
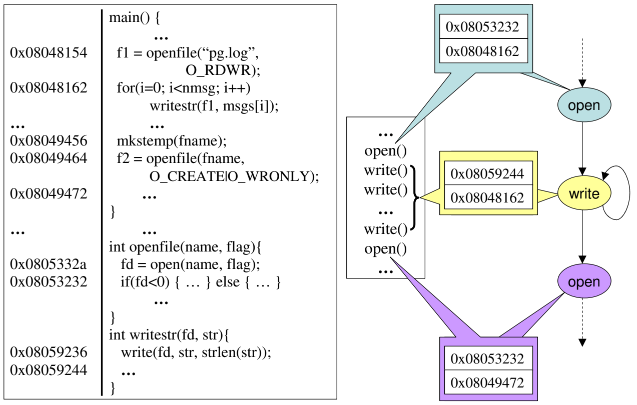


Figure 4: **An example system call graph.**



```
            main() {
                  …
0x08048154    f1 = openfile("pg.log",
                            O_RDWR);
0x08048162    for(i=0; i<nmsg; i++)
                   writestr(f1, msgs[i]);
…                  …
0x08049456    mkstemp(fname);
0x08049464    f2 = openfile(fname,
                   O_CREATE|O_WRONLY);
0x08049472         …
            }
…                  …
            int openfile(name, flag){
0x0805332a    fd = open(name, flag);
0x08053232    if(fd<0) { … } else { … }
                  …
            }
            int writestr(fd, str){
0x08059236    write(fd, str, strlen(str));
0x08059244    …
            }
```

Figure 5: **An example shows building system call graph with return addresses in call stack.**

```
1:  prev_node = NULL
2:  for each system call in the flat sequence do
3:      if the graph is empty
4:          curr_node = NULL
5:      else
6:          search the nodes pointed by edges starting on prev_node using context
7:          set curr_node as the node having the same context or NULL
8:      endif
9:      if curr_node = NULL
10:         add a new node (referred as curr_node)
11:         populate system call attributes in curr_node
12:         add an edge from prev_node to curr_node
13:     else
14:         update system call attributes in curr_node
15:         add an edge from prev_node to curr_node
16:     endif
17:     prev_node = curr_node
18: end
```

Figure 6: **Algorithm to convert a system call sequence to a system call graph.**

runs. Though the locations in an application can be represented by their virtual memory addresses, we use the stack of return addresses by collecting and analyzing the call stack information of the target process during each system call invocation. This gives system calls an *invocation context* in a more accurate way. The program in figure 5 illustrates this point. For the statements in the program, their memory addresses are shown on the left side of them. In the program, *open* and *write* system calls are wrapped in low level functions *openfile* and *writestr*. As these functions are used in different places in the program for different purposes, the system calls wrapped in them are also invoked for different purposes. Take function *openfile* as an example. It is used in two places in the program. In one place, it is to open an log file, and in the other place, it is to open a temporary file. Thus *open* is indirectly called two times for two different purposes. It is necessary to differentiate the *open* invocations for opening the log file and invocations for opening a temporary file because the names of temporary files are randomly generated and change across different runs. To clearly differentiate these two types of *open* invocations, we need not only the address where *open* is called, but also the addresses where *openfile* is called. This example illustrates that a "stack" of addresses of the functions in the calling hierarchy are needed to accurately differentiate the system call invocations.

We show the algorithm for collapsing a flat sequence of system call invocations to a system call graph in Figure 6. On line 6, the algorithm searches a matching node for a system call invocation following the edges in the system call graph. On line 14, when a node in the system call graph is found for the system call invocation, the attributes of the invocation, e.g., parameters, return values, and error numbers, are merged with existing attributes of the node. For each system call attribute, we again use a set to represent its distinct values among different invocations. This is illustrated in a *write* node of Figure 4.

2.4 Dealing with Multiple Processes

Applications, especially server applications, may have multiple processes running concurrently. We collect data for each process separately for two reasons. The first reason is that the causal relations between system calls can only be correctly reflected after separating interleaving system calls. Both building system call graphs and diagnosis require to know correct causal relations between system calls. While building system call graphs need the causal relations to form correct paths; diagnosis ~~need~~ causal relations to trace back to system calls ahead of the anomalies to get more information. For example, if we identify that a *write* call is an anomaly, we want to get the pathname of the file it ~~handles~~ by tracing back to an *open* call with the file descriptor. The second reason is that some attributes like signals, UIDs and GIDs are specific to a process. It is necessary to collect their values in a per-process mode to build accurate signatures for these attributes.

When we build signatures for a multi-process application, we divide its processes into groups based on the roles they play in the application, and build signatures separately for each process group. For example, a PostgreSQL server may create one or more back-end processes, one daemon process, and one background writer in each run. We build a system call graph and form a set of signatures for back-end processes, and we do the same for the daemon processes and background writers. When we build signatures for each process group, we treat the data collected for a process just like that collected in an execution of a single process application, and build signatures in a similar way. To identify which group a process belongs to, we use the stack information (return addresses) of the system call creating the process as a *context* of the process. Processes with same *context* are considered to be in the same group.

For multi-threaded applications, we collect data and build system call graphs and signatures for threads in the same way as we do for processes by treating each thread just like a process. While we can differentiate native threads through ptrace and /proc interfaces, which are managed by OS kernel, we cannot differentiate user-level (green) threads, which are managed at user space and thus transparent to OS kernel. As user-level threads have not been widely used, our current approach does not handle user-level threads.

We handle signal handler functions similarly to child processes, except that we collect only signal number and system call attributes for signal handler functions. When we build signatures for signal handlers, we use a 2-element tuple <process context, signal number> as the *context* of a signal handler. Thus, only data collected for signal handlers that handle same type signals for processes with same context are summarized to form signatures, e.g., we build a set of signatures for the SIGHUP signal handlers in the back-end processes of PostgreSQL.

3 Toolset Design and Implementation

In this section, we describe the architectural design and implementation of our diagnostic toolset for capturing applications' runtime behaviors, building signatures, and using which to find root cause of problems when they arise.

Figure 7 shows the overall architecture of our diagnostic toolset. First, a *tracer* tool (Section 3.1) is used to monitor the runtime behaviors of applications and record a log of these behaviors. Logging is started by having the tracer tool fork-execute the target application, e.g., *'tracer sample_program'*. However, the tracer tool can be used more seamlessly if we attach it to the shell process and have it monitor all of the child processes created by the shell. Since this tool is intended to run alongside of applications at runtime, having low overheads is crucial. We will see a detailed study of time and space overheads in Section 4.

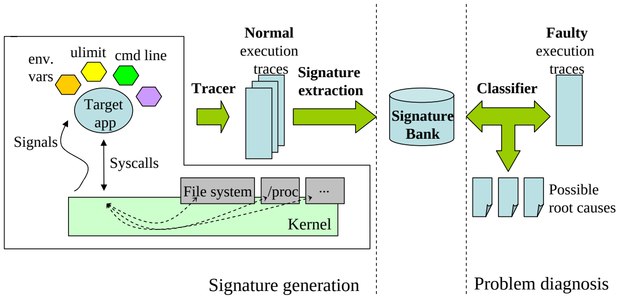


Figure 7: **System architecture**

On each run of the target application, the tracer tool will record and summarize its runtime behavior into a trace file. Multiple traces are then aggregated into a *signature bank*, a central repository where the target application's runtime signatures are distilled and built. We give an in-depth explanation of the steps involved in building runtime signatures in Section 3.2.

The last part of the toolset, called the *classifier* (Section 3.3), is used when an application is misbehaving. It is used for comparing the faulty execution trace (collected by the tracer) with the application's signature bank and classifying what differing features of the faulty trace from those in the signature bank might be the root cause of the problem. It is possible that sometimes multiple differing features are found. Since there is usually only one root cause, others are false positives. In Section 4.5, we discuss how to reduce the number of reported false positives.

3.1 Application Tracer

The tracer tool monitors an application's runtime behaviors via the *ptrace* interface, which is widely implemented on Linux and most UNIX variants. This approach has the benefit of not requiring instrumenting the target application or having access to its source code, and also does not need kernel modifications. Each time the target application invokes or finishes a system call or receives a signal, the application process is suspended and the tracer is notified of the event by the kernel and collects related information, e.g., call number, parameters, and return value. For a small set of system calls, we also collect some additional information that might be useful during problem diagnosis. ~~These~~ information ~~are~~ collected usually by having the tracer ~~making~~ extra system calls. For example, when *open* is called on a file, we make an extra *stat* call on the opened file to get its last modified time, which will become a part of the information we collect for that *open* call. In addition to files, we also collect additional information for other system objects such as shared memory, semaphore, sockets, etc. As explained in Section 2.3, to construct a system call graph from a sequence of system calls, the tracer also takes a snapshot of the call stack of the target application in the context of each system call.

As mentioned in Section 2.1, not all runtime behaviors can be captured by monitoring system calls, e.g., environment variables, ulimit, uid/gid of the user, etc. These information are collectively obtained by the tracer at the startup time of the target application, and they may be updated by monitoring system calls such as *setrlimit*, *setuid*, etc. at runtime.

For a single-process application, tracer puts all the monitored data into a single trace file. The trace file is logically separately into multiple sections to hold different categories of runtime data, similarly to that shown in Figure 1. The largest section by far is usually the system call section. To reduce space overheads, instead of saving a flat sequence of system calls, we convert it into a system call graph on-the-fly using the algorithm described in Figure 6. The conversion removes much redundant information by collapsing multiple system calls invoked in a loop into the same system call graph node. To reduce I/O overheads, the system call graph is kept in tracer's memory space via memory mapping of the trace file.

For a multi-process application, we keep one trace file per process (by detecting *fork/exec*), so we can separate the interleaving system calls made by different processes and maintain process-specific state information in each trace file. Ancestry relationships between processes are also kept in the trace file so we know exactly how the trace files are related and also at which point in the parent process the child process is spawned. Signal handlers are handled the same way by the tracer as ptrace can also trap ~~on~~ signals.

If a long-running application has large variations in its execution, its trace files may be filled with large volumes of data collected for runtime variants. By not saving these data into traces, we can reduce space overhead without influencing diagnosis. For a attribute having been considered as a runtime variant, we set an upper limit (512 in our current implementation) on the size of the set holding its distinct values. Thus new values of a runtime variant are not collected into traces or merged into signature bank when the set size ~~has reached~~ 512. 512 is chosen so it is sufficient to cover semi-invariants with large number of distinct values, yet small enough for it not to be a storage burden.

3.2 Signature Bank

For single-process applications, signature bank is simply an agglomerate of one or more normal execution trace files. When adding the first trace file to an empty signature bank, the trace file simply becomes the signature bank. As illustrated in Figure 8, when adding new trace files to the signature bank, values of attributes (e.g., an environment variable) in theses traces are compared to those in the signature bank. If the value of an attribute in the new trace is different from that of the signature bank, the new value is added to the set of possible values of that attribute in the signature bank. Otherwise, the signature bank remains unchanged. Since most attributes do not change between runs, the size of a signature bank grows very slowly over time. When merging the system call graph in a trace file into the signature bank, we use a similar algorithm as that described in Figure 6.

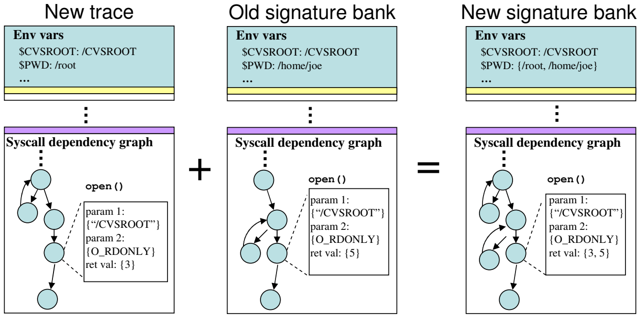


Figure 8: **An example showing how a trace file is merged with the signature bank.**

All attribute values and system call graph paths are versioned in the signature bank. This is useful when a faulty execution trace is inadvertently added to the signature bank. Versioning allows this action to be easily reverted.

For a multi-process application, its signature bank is consisted of multiple sub-banks, each of which describing a separate process group. These sub-banks are organized to reflect the ancestry relationships between the processes they are associated with. Merging of the trace files of a multi-process application into the sub-banks is performed following the algorithm described in Figure 9.

Our current approach has to re-build application signatures after some administrative changes. For example, updating the application or the shared libraries changes the return addresses of the functions which invoke system calls directly or indirectly. Because these addresses are used as context to build system call graphs and to match system call invocations, system call graphs and signatures built before an update cannot be used any more after the update because we cannot find the signatures correctly with the return addresses in a new context.

```
FUNCTION aggregate(trace_file, sub_bank)
    add trace_file into sub_bank;
    FOR each child process DO
        let trace_file_child be its trace file
        look for a child of of sub_bank using the context
            of the child process (referred as sub_bank_child);
        IF such child does not exist
            create a new sub_bank (referred as sub_bank_child);
            make sub_bank_child a child of sub_bank;
        END IF
        aggregate(trace_file_child, sub_bank_child);
    END DO
END FUNCTION

IF signature_bank is empty
    create a new sub_bank (referred as sub_bank_root);
END IF
trace_file_root = trace file of the main process;
sub_bank_root = root of the sub_bank tree;
aggregate(trace_file_root, sub_bank_root);
```

Figure 9: **Algorithm to aggregate trace files into signature bank for a multi-process application.**

3.3 Fault Diagnosis

When an application fault occurs, a *classifier* tool is used to compare the faulty execution trace with the application's signature bank. The comparison is straightforward. Application and system states in the faulty execution trace are first compared with those in the signature bank. Mismatched attributes are then identified. The system call graph in the faulty execution is next compared with that in the signature bank, a node at a time. For each node, its attributes are compared with those on the corresponding node within the signature bank. We do not list all the mismatched attributes as potential root causes—this might result in too many false positives.

To ~~highlight~~ the more likely root causes to the person diagnosing the problem, the classifier ranks the results. If an attribute from the faulty execution mismatches a signature that is either an invariant or has a very small cardinality, it is more like to be the root cause than if the signature were to a higher cardinality value. Additionally, among the mismatched attributes found in the system call graph, we give more weight to those attributes located closer to the "head" of the graph. The reason being, due to causal relationship, the mismatched attributes that are closer to the top of the call graph are likely to be the cause of the mistaches found toward the bottom.

4 Case Studies

In this section, we evaluate our approach using real-world application problems. We would like to observe how effectively and accurately the tool is able to handle these problems and also identify some of its limitations.

4.1 Experimental Methodology

Our evaluation covers three popular applications: Apache web server [1], CVS version control system [15], and PostgreSQL DBMS server [14]. Rather than injecting contrived faults to test our system, we evaluated actual problems faced by users of these applications, drawn from problem reports on Internet forums and from bug reporting tools such as Bugzilla. Our target problems include configuration files, environment variables, resource limitations, user identities, and properties ~~keep persistent information such as log files.~~ Software bug detection is not our goal in this work. We describe a subset of our experiments in this section, with the representative problems shown in Table1.

For each application, our general approach was to first collect traces by running it with a series of standard operations or workloads that represent its normal usage and operation. In some cases, we also change some system settings to emulate administrators tuning the system or modifying configurations. For example, when collecting traces for CVS, we perform the commonly used CVS operations such as import, add, commit, checkout etc. multiple times on different modules in both local and remote CVS repositories. The CVS repositories are changed by resetting shell environment variable $CVSROOT. We integrate these normal operation traces into the signature bank to generate the runtime signatures of the application. After these two steps, we inject the selected fault manually and collect the faulty execution trace for each problem scenario. Afterward, the system is returned to the non-faulty state. Finally, we use the classifier to identify possible root causes by comparing the faulty execution traces with the application's signatures.

In each case we discuss the ability of the classifier to effectively distinguish erroneous traces from normal signatures to aid in diagnosing the problem. In addition, since the applications being diagnosed (and their threads) must be launched from our tracer tool, the performance impact as well as space overhead due to trace and signature storage are important measures of the feasibility of our diagnosis approach. Therefore, for each application we estimate overhead in execution time or response time slow down by repeating the execution without tracer. We also record the size of the individual traces and the signature bank. Trace file size is less important than the size of the signature bank since trace files can be deleted after they are inserted into the signature bank. However, if the size of trace files is reasonably small, we can retain several recent traces and batch the aggregation operation to amortize the cost of insertion into the signature bank.

All the experiments below are performed on a Dell Dimension 3100 desktop computer with a 3GHz Intel Pentium 4 CPU and 1GB memory running Red Hat Enter-

Apache problems		
Index	Symptom	Root cause
1	Intermittent failures of httpd processes	Log file size is getting too large, close to 2GB
2	Httpd cannot start	File system containing httpd log files is mounted as read-only
3	Httpd cannot start, ~~being~~ unable to load share libraries in correct version.	Paths are re-ordered in $LD_LIBRARY_PATH.
4	Httpd crashes when the number of connections is large.	Httpd loaded by crond has more restrict resource limit.
5	Web clients cannot access contents pointed by symbolic links	User removed the *FollowSymLinks* directive from httpd.conf

CVS problems		
Index	Symptom	Root cause
6	User cannot checkout a specified CVS module	$CVSROOT is pointing to a directory that is not a CVS repository
7	CVS client cannot connect to CVS server	A non-default ssh port number is specified in /etc/ssh/ssh_config
8	Accesses to CVS repository are denied	User is not added to CVS group
9	User cannot connect to CVS server with an error message ”temporary failure in name resolution”	Network cable is disconnected

PostgreSQL problems		
Index	Symptom	Root cause
10	DBMS accepts only connections from local machine	Config file pg_hba.conf is mistakenly changed
11	Server cannot start	A stale postmaster.pid file is left undeleted after improperly shutting down the server

Table 1: Description of the problem symptoms and their root causes for Apache, CVS, and PostgreSQL.

prise Linux WS v4 and Linux kernel version 2.6.9.

4.2 Apache

For tests with Apache, we use WebStone 2.5 [19] to emulate multiple clients which concurrently access web pages through Apache. Besides generating workloads, we also use WebStone to measure the average response time of Apache. The web pages served by Apache are generated by LXR [13] (Linux Cross-Reference), which is a widely used source code indexer and cross-referencer. We use LXR to serve user queries for searching, browsing, or comparing source code trees of three versions of Linux kernels.

We repeat the following operations ten times to generate ten corresponding traces of Apache server: start the server with tracer, run WebStone on another machine for 45 minutes generating HTTP requests, and stop the server.

We use the signature bank built from the traces to diagnose the Apache problems listed in Table 1. Both problems 1 and problem 2 are related to log files. Because the contents and the sizes of log files usually change frequently, problems related to log files are difficult to diagnose by directly comparing persistent states without capturing the run-time interactions of the application. Our classifier identifies the root causes by finding out abnormal system calls in the faulty execution traces, *write* for problem 1 and *open* for problem 2. The abnormally behaved system calls are identified because their error numbers do not match their signatures captured in the signature bank. Figure 10 illustrates the difference between the values of these attributes in the faulty execution and their signatures in signature bank for problem 1. In problem 1, system call *write* in faulty execution cannot write access logs into log file *access_log* successfully. The root cause is revealed from its error number (*EFBIG*, which means file is too large). Similarly, in problem 2, system call *open* in faulty execution cannot open file *error_log* successfully. The root cause is revealed from the return value(-1, which means the system call fails) and its error number (*EROFS*, which means read-only filesystem). In addition to abnormally behaved system calls, the classifier also identifies that some httpd processes receive *SIGXFSZ* signals in the faulty execution in problem 1. The *SIGXFSZ* signal is only thrown by the kernel when a file grows larger than the maximum allowed size.

Figure 11 illustrates the command used and the output of our tool in diagnosing problem 1. The classifier command usually has two parameters, the faulty exe-

```
[sigexp@sysprof ~]$ classifier sigbank/Apache traces/Apache_problem1.trace
** Record_ID: 58        Node_ID: 58     Graph_ID: 6     System call: write
Fails to write file /m/logs/access_log.
Note: File too large.
** Signal SIGXFSZ received by process 8259, 8260, 8261, 8262, 8267, 8268, 8269, 8271
Signal appears only in faulty execution.
Note: application appends a file larger than maximum allowed size.
```

Figure 11: **Command line and console output of Classifier diagnosing Apache problem #1**

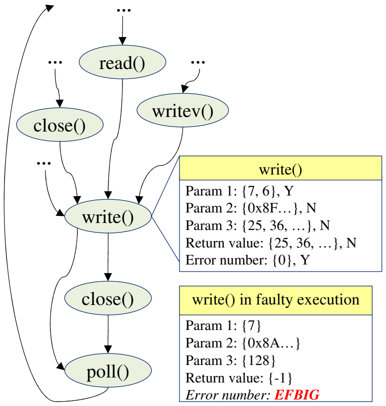


Figure 10: **Signatures of the attributes in a *write* system call and the values of these attributes in faulty execution in problem 1.** The "Y" or "N" after each signature (set) indicates whether the signature is an invariant or a semi-invariant that passes KS-test and thus can be used for diagnosis. The abnormal attribute in faulty execution is in italic font.

cution trace ("traces/Apache_problem1.trace"), and the signature bank of Apache (stored in a file named "sigbank/Apache"). The messages under the command are console output of the classifier. The first line of the console output shows one of the possible root causes of this problem—the abnormal *write* system call invocation. Record_ID, Node_ID and Graph_ID indicate where the signatures are located in the signature bank so users can manually check the entire system call graph if necessary. The second and the third lines show how the system call invocation behaves abnormally. The remainder of the console output reports a second possible root cause, namely the new signals which don't appear in normal executions.

Problem 3 of Apache is caused by a modified environment variable. The classifier identifies the environment variable ($LD_LIBRARY_PATH) by comparing the value of the environment variable in the faulty execution trace against those in the signature. When Apache performs normally, paths in the environment variable are in the right order, and Apache can load correct libraries. Since this variable usually does not change in normal executions, we capture the value of $LD_LIBRARY_PATH as a signature in the signature bank. In the fault execution, paths in $LD_LIBRARY_PATH are reordered. As a result, when comparing the faulty execution trace against the signatures, the classifier finds the new value does not match the value in the signature bank and reports it as a possible root cause. Besides the changed environment variable, the classifier further identifies that the fault is caused by opening incorrect files in faulty execution because the pathnames of these files are different with those in the signatures. Based on the pathnames, administrators may identify these files are shared libraries.

Problem 4 is caused by a restricted resource limit setting on the maximum number of processes owned by the same user. Our classifier diagnoses this problem by observing the abnormal return values and error numbers of the *setuid* system calls made by the httpd processes. The *setuid* system call increases the number of processes owned by the user which Apache runs as. The return values indicate that the system calls did not succeed, and error numbers indicate that the failure was caused by unavailable resources. In addition, since we keep resource limit as an attribute of the shell environment signature. The new resource limit value in the faulty execution differs with that in the signature, which is another indication of the root cause.

Problem 5 is caused by a change in a config file httpd.conf. In building application signatures, file metadata such as file size, last modification time are collected, usually when an *open* call happens. When comparing the faulty execution trace to the signatures, our classifier discovered that attributes of httpd.conf such as file size, last modification time etc. do not match those in the signatures. Thus our classifier can attribute the application failure to the change in httpd.conf.

In these experiments, the response time of Apache observed by WebStone is increased by 22.3% on average. The performance overheads are non-negligible. We propose a method to reduce performance overheads in Section 5.

Figure 12 shows the change in size of an Apache trace in a 45 minutes period when Apache is serving requests. In the first a few minutes, the system call graphs are small and the value sets for the attributes do not include so many distinct values. The trace grows quickly as new system call graph nodes and new values are added into the trace. Afterward, the growth slows down with the system call graphs becoming more and more complete and the value sets covering more variations of the at-

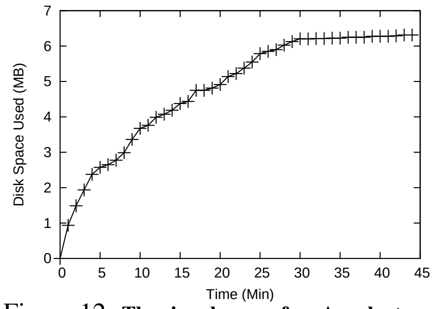


Figure 12: **The size change of an Apache trace.**

tributes. This trend is apparent especially after the 30th minute due to redundancy across requests. At the end of the execution, the trace occupies 6.3MB space, recording nearly 11 million system call invocations.

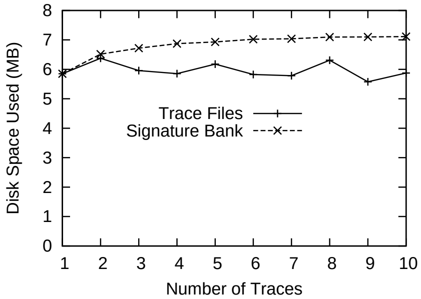


Figure 13: **Sizes of traces and the signature bank for Apache.**

Figure 13 shows the size of the traces, and the change in the size of the signature bank after aggregating each of the traces. Though the size of each trace is around 6MB, the size of the signature bank grows very slowly when a new trace is inserted because redundant data are merged.

4.3 CVS

As we have explained, we collect traces of commonly used CVS operations on different modules including the source code of our diagnostic tools, *strace*, *Gnuplot*, and *PostgreSQL* in both local and remote CVS repositories.

Similar to problem 3, problem 6 is also caused by a modified environment variable. The symptom of problem 6 is that a user cannot checkout a specified CVS module. Figure 14 illustrates the command used and the console output of our tool. The first line of the console output shows one of the possible root causes of this problem—a new $CVSROOT's value has been used in the faulty execution. When CVS performs normally, the environment variable $CVSROOT has been changed several times and pointed to different repositories. These repositories include a local one at */home/cvs/repository* and several remote ones, from which we checked out the source code of *strace*, *Gnuplot* and *PostgreSQL*. Though this variable has been changed multiple times, our tool determines with KS-test that this attribute is not a runtime variant, and uses its signature in diagnosis because the D-statistic of this variable is 0.42, which is far above the corresponding critical value $D_\alpha = 0.22$. In the faulty execution, $CVSROOT is changed to /home/cvs. As a result, the classifier finds the new value does not match the signature and reports it as a possible root cause.

Our tool also discovers (2-4 line of the output in Figure 14) an abnormal *access* system call invocation. The *access* system call is made by CVS to check the access permission of the CVSROOT directory. In normal executions, the *"pathname"* parameter is "/home/cvs/repository/CVSROOT", the return value is 0, and error number is 0. However, in the faulty execution, the *access* call has a different *"pathname"* parameter ("/home/cvs/CVSROOT") because $CVSROOT has been changed to /home/cvs. /home/cvs/CVSROOT is a non-existent directory. Thus the system call returns -1 and the error number is set to ENOENT accordingly. Our classifier interprets the semantics of the return value and error number so users can understand easily.

This simple example demonstrates how our tool helps to pinpoint the root cause of the problem and reveals detailed information for users to examine and verify, while the error message printed by CVS is simply "cannot find module 'strace' - ignored", which is not very descriptive and may be misleading.

Problem 7 is about a failed CVS server connection because of a non-default SSH port number in the configuration file. CVS usually makes connections with the remote CVS server via SSH using its default port number (number 22). In this scenario, the configuration file of the SSH client, /etc/ssh/ssh_config, has been modified to use a customized port number. Therefore, all SSH client requests will use this customized number instead of the default port number. However, the SSH server on the CVS server is not changed accordingly to accept this new port. Our tool identifies the config file to be one of the root causes in a similar way as in problem 5. When comparing the faulty execution trace to the signatures, our classifier discovers that the file was modified when the application is doing an *open* call, since the file size, last modification time etc. do not match. Beside the config file, our classifier also reports that a *connect* system call invocation is having a different port number as its parameter. This information indicates the cause might be a bad port number.

Problem 8 is one of the problems used to evaluate AutoBash [17], we revisit this problem with our approach. AutoBash solves this problem by looking for the causality between the group identifiers (*gids*) of the user and the access permissions of CVS repository. Our approach builds a signature for *gids* used in CVS normal executions. In our signature bank, the signature of this attribute

```
[sigexp@sysprof ~]$ classifier sigbank/CVS traces/CVS_problem6.trace
Environment variable $CVSROOT has been changed to a new value "/home/cvs".
** Record_ID: 158       Node_ID: 95     Graph_ID: 1     System call: access
Faulty execution checks user permission of a file/directory "/home/cvs/CVSROOT".
System call fails.
Note: No such file or directory.
```

Figure 14: **Command line and console output of Classifier diagnosing CVS problem #6**

always takes one value since the CVS client always uses the CVS group. When comparing the faulty execution trace against the signatures, the classifier cannot find the *gid* of CVS group in the set of *gids* used by the faulty execution, thus it classifies it as the root cause. Similar to problem 6 and problem 7, the classifier observes abnormally behaved system calls in faulty execution trace and prints out diagnosis messages of the errors.

From the problems we present here, the only problem for which the classifier cannot provide accurate diagnosis is problem 9. The classifier observes the abnormal behavior of *poll* system call recorded in the faulty execution trace and concludes that ~~system call~~ *poll* gets timeout as the root cause. The classifier fails to identify the real root cause because we do not collect information about hardware states of network card. Though the classifier cannot exactly locate the root cause, it discovers that the anomaly was caused by timeout on network communications. The information may be helpful because it can reduce the scope of investigation for the exact root cause.

While the tracer slows down CVS operations by different percentages, we observe an average slowdown of 29.6%. The smallest slowdown is less than 1%. It is observed when we checkout Gnuplot from the remote repository *gnuplot.cvs.sourceforge.net* because network latencies dominate the delays. The greatest slow-down is 77.1%, which is observed when we commit a version of a small module to the local repository. We collected 26 traces for CVS in total. Their sizes range from 0.1MB to 1.6MB. They record about 1.8 millions system call invocations, and the largest trace file records over 219 thousands system call invocations. The size of the signature bank is 6.5MB after these traces are aggregated.

4.4 PostgreSQL

For PostgreSQL, we collect 16 traces as it processed queries generated by the TPC-H [18] benchmark for decision support systems.

In PostgreSQL, access control configurations are specified in pg_hba.conf. PostgreSQL loads this config file when it is started, and also does a reload when receiving a SIGHUP signal. Thus with a reload command which sends PostgreSQL a SIGHUP signal, users may make ~~the~~ changes to pg_hba.conf take effect immediately without restarting PostgreSQL. In evaluating problem 10, we injected faults by modifying pg_hba.conf when PostgreSQL was running and let PostgreSQL reload pg_hba.conf with reload command. We run reload commands to let PostgreSQL load pg_hba.conf in its signal handler, which we have exercised in normal execution. Our classifier ~~can identify the~~ root causes in a similar way as it diagnos~~es~~ problem 5 and problem 7. The console output is shown in Figure 15.

In problem 11, the shell script which loads PostgreSQL checks for the existence of the postmaster.pid file. If the file exists, it stops loading PostgreSQL assuming it has been started already. In normal executions, an *access* system call is used to check the existence of this postmaster.pid file, and usually returns -1 with the error number set to *ENOENT*. In faulty execution, the system call returns 0 indicating the existence of the file. Our classifier discovers the root cause by comparing the error numbers and return values of the *access* call.

We observed that, using the tracer, the queries are slowed down by 15.7% on average. Tracing causes less performance overhead for PostgreSQL than for the other two applications because most TPC-H queries are computation-intensive and thus PostgreSQL makes system calls infrequently. The traces are from 0.6MB to 2.1MB, and the signature bank is 3.2MB after aggregating the traces.

4.5 Accuracy and Effectiveness

Our approach identifies root causes of problems by comparing a faulty execution with the application's normal runtime signatures. Having "good-quality" runtime signatures is critical to the identification of root causes. From our experience, identifying the root cause is usually not difficult using our approach as we are comprehensively capturing the interactions between the application and the system states, whether or not they are persistent or non-persistent (the root causes of the above problems are all correctly identified using our tool). In addition to being able to identify root causes, it is also important, if not more, to limit the number of false positives. Having too many false positives will render the tool useless ~~to its user~~ in practice.

False positives are generally caused by two reasons. One reason is related ~~with~~ KS-test. Some normal runtime variants may not be ruled out ~~from~~ diagnosis if the significance level is set too high. ~~User~~ may reduce false positives by decreasing the significance level. However,

```
[sigexp@sysprof ~]$ classifier sigbank/postgresql traces/postgresql_problem10.trace
** Record_ID: 45287     Node_ID: 11     Graph_ID(SIGHUP): 3     System call: open
File /home/pgsql/db/pg_hba.conf has been changed since last run.
```

Figure 15: **Command line and console output of Classifier diagnosing PostgreSQL problem #10.**

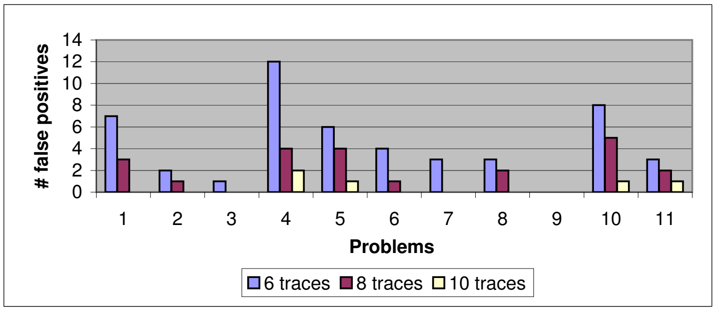


Figure 16: **Number of false positives decreases when more traces are aggregated into signature bank.**

if the level is set too low, attributes useful for diagnosis may be mistakenly identified as runtime variants and thus lead to false negatives. From our experience, setting the level to 10% works well for all the problems in our experiments (the numbers of false positives in diagnosing the problems are as shown in Figure 16). Nevertheless, we have set the significance level ~~as~~ a knob, in case users may need to adjust it in real-world environments to reduce false positives without causing false negatives. The other reason is that signature bank cannot cover all the possible normal variations of the attributes. For example, in problem 7, if the client has never connected to a CVS server before, the signature of $CVSROOT does not include the name of the new repository. Thus the name of that new repository in $CVSROOT may be identified as one of the possible root causes false-positively. Aggregating more traces may make signature bank more "complete", and thus is helpful in reducing such false positives. To illustrate this, for each problem, we also show the number of false positives in Figure 16 when we increase the number traces aggregated into the signature bank.

5 Optimization

Our experiments in Section 4 show that the performance overheads of tracing are quite noticeable when using on real systems. In this section, we propose a technique of optimizing ptrace to significantly reduce these overheads.

Most of the performance degradation comes from information collection and trace file updating when a system call happens. To reduce the context switches and memory copies introduced by updating trace files, we have used direct memory-mapping to map trace files into the memory space of the tracer. However, for each each system call made by the traced application, the following overheads are still incurred.

- 4 additional context switches, switching from kernel to tracer and back from tracer to kernel both at system call entry and exit. Time consumption is about 20.2 microseconds in total.
- Getting system call number, return value, error number, or each parameter would incur two additional context switches of 0.9 microseconds.
- Peeking into the user stack of the target application to get the content of its stack frames would require the OS to read the application's page table to resolve virtual addresses. Each of these operations takes about 2.0 microseconds.

Since most system calls usually take only a fraction of a microsecond, in the same time scale or even shorter than these activities, these overheads may significantly slow down the traced application. To reduce these overheads, we modified several ptrace primitives and added two primitives in Linux kernel. These improvements only require slight modifications to the current ptrace implementation. Less than 300 lines of new code are added. The new ptrace actions/primitives we added are:

- PTRACE_SETBATCHSIZE: Set the number of system calls to batch before notifying the tracer.
- PTRACE_READBUFFER: Read and then remove data collected for the system calls in same batch from a reserved buffer space.

The improved ptrace interface reduces overheads by decreasing the number of ptrace system calls the tracer needs to call and the number of context switches. This is done by having the kernel reserve a small amount of buffer space for each traced process (40KB in the current implementation) so it can be used by ptrace to store data it has collected on behalf of tracer without interrupting the traced application on every system call. Instead, the traced application is only interrupted when (i) the buffer space is approaching full, (ii) a user-defined batch size (of system calls) is reached, or (iii) a critical system call is made, e.g., *fork*, *clone*, and *exit*. By batching the collection of information on system calls, the costs of context switches and the additional ptrace system calls are dramatically reduced.

We repeated the trace collection operations for *Apache*, *CVS*, and *PostgreSQL* in Section 4 with the improvements introduced above. The slowdowns of these applications are shown in Figure 17 with the batch size varying from 1 to 64.

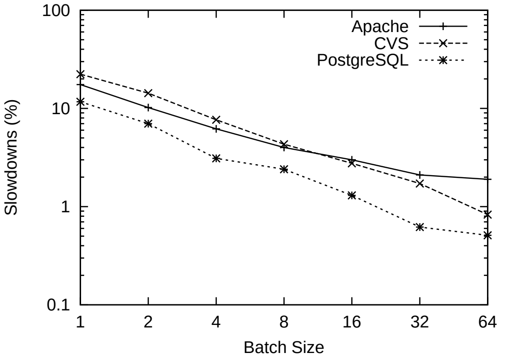


Figure 17: **Slowdowns for *Apache*, *CVS*, and *PostgreSQL*, with batch size varying from 1 to 64.**

Even when batch size is equal to 1, the applications have smaller slowdowns with improved ptrace than they do with original ptrace. There are two reason. One reason is that OS invokes tracer only once with improved ptrace for each system call on its exit, instead of twice with original ptrace on both system call entry and system call exit. The other reason is that the tracer needs only one improved ptrace system call (PTRACE_READBUFFER primitive) to get the required data, instead of multiple ptrace system calls with original ptrace. With the ~~increasing of~~ batch size, ~~the~~ slowdowns are reduced significantly for all applications. When batch size is increased to 64, the slowdowns of *Apache*, *CVS*, and *PostgreSQL* with improved ptrace are reduced to 1.9%, 0.8%, and 0.5% respectively. For normal applications, such small slowdowns are acceptable.

6 Related Work

As systems are becoming more complex and problem diagnosis is taking longer and requiring more expertise, quite a few number of related works, that we describe in Section 6.1, have attempted to automate problem diagnosis and resolution. The general approach we have taken to automate problem diagnosis in this work—capturing and utilizing application's runtime behavior—has also been applied to other areas such as debugging and intrusion detection, which we cover in Section 6.2 and Section 6.3, respectively.

6.1 Problem Diagnosis and Resolution

A general approach to diagnosing and solving application problems, especially those caused by misconfiguration, is to regularly checkpoint system states and keep track of state changes. For example, Strider [23] takes periodic snapshots of the Windows Registry. When a problem occurs, recently changed or new registry entries are presented as potential root causes. Chronus [25] and FDR [20] also take into account of changes in other system states, not just in the Windows Registry. FDR actually records every event that changes the persistent state of a system. While such system-wide approach is generally fairly comprehensive when it comes to recording changes, filtering out noises (i.e., unrelated changes) and pin-pointing the exact root cause can sometimes be difficult. On the other hand, the approach we have taken is very application-specific. We only consider those changes that are known to have an impact on the application that we are diagnosing.

Yuan *et al.* [26] is the ~~closest work~~ related to ours. They try to match the system call sequence of a faulty application with that of a set of known (Top100) problems. When a match is made, the pre-cooked solution to that problem is presented to the user. One problem with this approach is that there is a huge number of different applications, and for each application, there are many possible problems. As a result, the 80-20 rule might not hold true here, which means building a knowledge base of only the Top100 problems might not be sufficient. Additionally, there are a few problems with comparing only system call *sequences*, which we have discussed in Section 2. In our work, we address these problems by converting system call sequences to graph structures. PeerPressure [22] is closely related to the Strider work, also looking at the Windows Registry. It goes a step further and uses statistical methods to compare application-specific Windows Registry entries across many machines to detect abnormal entries. However, this work is limited to only Windows platform and problems caused by misconfiguration in the Windows Registry.

AutoBash [17] is a set of interactive tools to deal with misconfiguration problems. It uses OS-level speculative execution to track causal relationships between user actions and their effect on the application. Fundamentally different from other related works in this section and ours, AutoBash does not monitor historical changes in system and application states in order to find root cause. Instead, it relies on the user to have sufficient ~~amount of~~ expertise in finding the root cause and records the actions taken, in case the same problem occurs again in the future. Users are also required to define predicates specifying what is the correct behavior of an application. These can sometimes be difficult and time consuming to define. In our approach, the correct behavior of an application is already captured by its runtime signatures.

6.2 Debugging

Capturing and discovering program runtime invariants are important to programmers when debugging. Various tools [2, 9, 10, 16, 7] are developed for this purpose. Daikon [2] detects invariants based on the values of a set of tracked expression at various program points such as reading or writing a variable, procedure entries and exits. DIDUCE [9] hypothesizes invariants that a program obeys in its execution and gradually relaxes the hypothesis when it observes a violation. These tools usually instrument an application at a very fine granularity to track its "internal" problems. As a result, slowdown can be as much as a hundred times slower or more, which is still acceptable during debugging.

Our tool focuses on diagnosing problems after an application has been released and works while the application is being used. Therefore, low overhead is the key for such tool to be pragmatic, which we have demonstrated in the evaluation of our tool. Furthermore, we do not require having application's source code and monitor the application using a black-box approach. This allows our tool to work also with commercial software which almost always do not have accompanying source code available.

6.3 Intrusion Detection

In security area, system calls are commonly traced to detect intrusions [4, 11, 24, 21, 3], where patterns detected in a system call sequence are most important, and other information, such as return value, parameters, and error code, are less so. Intrusion patterns are relatively easier to detect than that of a functional problem of an application, which can happen anywhere in the application and caused by almost anything. Therefore, for problem diagnosis, more detailed information and more types of information are needed to perform accurate diagnosis. And, at the same time, we need to incur as little ~~overheads~~ as possible; like intrusion detection systems, our tool is meant to run alongside of applications. David and Drew build non-deterministic pushdown automata for system calls made by applications [21], which are very similar to system call graphs in our approach. However, they build the automata to have a complete coverage of all the possible execution paths to avoid false alarms. In our approach, we only need to have common execution paths in our signature bank to detect anomalies.

7 Limitations

Our application diagnosis approach and implementation does have a number of limitations. For example, we do not currently address the problem of how to label a particular application execution trace as faulty. Currently, we rely on a manual indication from the user that invokes the problem diagnosis process. We also adopt a somewhat conservative approach in the amount of information that is collected in application traces. More analysis is needed to identify the minimum set of data that provides a high degree of accuracy for diagnosing common problems. More complete information in signature data is likely to improve the chances that new problems can be diagnosed. Finally, our results, while representative of widely used applications and real problems, are nevertheless limited to a few case studies.

Despite these limitations, we believe that this approach for problem diagnosis represents a promising step toward automating application problem solving, and could lead to significant time (and hence, cost) savings in enterprise IT environments.

8 Conclusions and Future Work

We have proposed an automatic approach to diagnose application faults. Our approach finds problem root causes by capturing the run-time features of normal application execution in a *signature* and examining the faulty execution against the signature. We have implemented our approach in a user level tool and evaluated it using real application problems that demonstrate that the approach can accurately diagnose most of these problems. We have tested both the space and time overheads of deploying the diagnosis tool, and though the impact on application response time is high, we have proposed and tested a method that significantly reduces it.

Currently our approach builds application signatures on each individual computer system. It is difficult for a user to obtain complete signatures for an application. By exchanging and sharing signatures built on multiple computer systems, users can have more complete signatures to cover more problems. As future work, we plan to explore approaches to share signatures across hosts (e.g., inspired by [22]). Paper [22] has introduced a method to build signatures for Windows registries, which can be shared across hosts, by identifying and excluding information subject to change across hosts. However, the method can hardly be applied with our approach. When we build signatures for an application on each host, much information specific to that host is included into its signature, such as UID and GID the application is running as, size and last modification time of its configuration files, etc. To share signatures across hosts, some conversion is required. For example, if GID has been considered as a piece of signature in a host, and we want to share the signature to another host, we have to replace it with the GID on that host.

We have evaluated our approach with a number of real problems in a testbed setting, but also plan to evaluate its effectiveness and costs in live deployments, such as campus computer labs.

References

[1] The Apache Group. *The apache HTTP server project*. URL: http://httpd.apache.org/.

[2] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. *IEEE Trans. Software Eng.*, 27(2):99–123, 2001.

[3] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong. Anomaly detection using call stack information. In *SP '03: Proceedings of the 2003 IEEE Symposium on Security and Privacy*, page 62, Washington, DC, USA, 2003. IEEE Computer Society.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *SP '96: Proceedings of the 1996 IEEE Symposium on Security and Privacy*, page 120, Washington, DC, USA, 1996. IEEE Computer Society.

[5] J. Frank J. Massey. The kolmogorov-smirnov test for goodness of fit. (253):68–78, 1951.

[6] J.-P. Garbani, S. Yates, and S. Bernhardt. The evolution of infrastructure management. Forrester Research, Inc., October 2005.

[7] J. Ha, C. J. Rossbach, J. V. Davis, I. Roy, H. E. Ramadan, D. E. Porter, D. L. Chen, and E. Witchel. Improved error reporting for software that uses black-box components. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, pages 101–111, New York, NY, USA, 2007. ACM.

[8] M. Haardt and M. Coleman. *ptrace(2)*, 1999.

[9] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *ICSE '02: Proceedings of the 24th International Conference on Software Engineering*, pages 291–301, New York, NY, USA, 2002. ACM.

[10] M. J. Harrold, G. Rothermel, K. Sayre, R. Wu, and L. Yi. An empirical investigation of the relationship between spectra differences and regression faults. *Software Testing, Verification & Reliability*, 10(3):171–194, 2000.

[11] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[12] H. Huang, R. Jennings, Y. Ruan, R. Sahoo, S. Sahu, and A. Shaikh. PDA: A tool for automated problem determination. In *Large Installation System Administration Conference (LISA 2007)*, Dallas, TX, December 2007.

[13] LXR. Linux cross-reference. URL: http://lxr.linux.no/.

[14] PostgreSQL Global Development Group. *PostgreSQL: The world's most advanced open source database*. URL: http://www.postgresql.org/.

[15] D. R. Price. *CVS: Concurrent versions system*, 2006. URL: http://www.nongnu.org/cvs/.

[16] M. Renieris and S. Reiss. Fault localization with nearest neighbor queries, 2003.

[17] Y.-Y. Su, M. Attariyan, and J. Flinn. Autobash: improving configuration management with operating system causality analysis. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 237–250, New York, NY, USA, 2007. ACM.

[18] Transaction Processing Performance Council. *TPC-H*. URL: http://www.tpc.org/tpch/.

[19] G. Trent and M. Sake. WebSTONE: The first generation in HTTP server benchmarking, 1995. URL: http://www.mindcraft.com/webstone/paper.html.

[20] C. Verbowski, E. Kiciman, 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 *OSDI '06: Proceedings of the 7th symposium on Operating systems design and implementation*, pages 117–130, Berkeley, CA, USA, 2006. USENIX Association.

[21] D. Wagner and D. Dean. Intrusion detection via static analysis. In *SP '01: Proceedings of the 2001 IEEE Symposium on Security and Privacy*, page 156, Washington, DC, USA, 2001. IEEE Computer Society.

[22] H. J. Wang, J. C. Platt, Y. Chen, R. Zhang, and Y.-M. Wang. Automatic misconfiguration troubleshooting with peerpressure. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 17–17, Berkeley, CA, USA, 2004. USENIX Association.

[23] 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. In *LISA '03: Proceedings of the 17th USENIX conference on System administration*, pages 159–172, Berkeley, CA, USA, 2003. USENIX Association.

[24] C. Warrender, S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: Alternative data models. In *IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[25] A. Whitaker, R. S. Cox, and S. D. Gribble. Configuration debugging as search: finding the needle in the haystack. In *OSDI'04: Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation*, pages 6–6, Berkeley, CA, USA, 2004. USENIX Association.

[26] C. Yuan, N. Lao, J.-R. Wen, J. Li, Z. Zhang, Y.-M. Wang, and W.-Y. Ma. Automated known problem diagnosis with event traces. In *EuroSys '06: Proceedings of the ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375–388, New York, NY, USA, 2006. ACM.