Assisting Failure Diagnosis through Filesystem Instrumentation

by

Liang Huang

A thesis submitted to the Faculty of Graduate Studies and Research
in partial fulfillment of the requirements for the degree of

Master of Science

Department of Computing Science

Abstract

With increasing software size and complexity, corrective software maintenance has become a challenging process. When a failure is reported, it takes time and expertise for human operators to collect the right information and pinpoint the root cause. Typically, the operators are overloaded with information generated from many system components, and need assistance.

In practice, however, failures are often recurrent. If they can be identified accurately, the appropriate fix may already be known from prior collected experience about the system. Our approach to diagnose failures is to look at differences in the state of the filesystem and how files are accessed under normal and abnormal situations. In this research, we monitor the behavior of the system through its file-related calls on an instrumented filesystem. When a failure occurs, these calls are abstracted and classified to identify the likely cause.

A diagnostic tool is implemented based on this approach. Through an experiment involving one J2EE Web application, we present the effectiveness of our approach in terms of precision and recall.

Acknowledgements

This research would not have been possible without the assistance of many members of the Department of Computing Science. In particular I would like to thank my supervisor Dr. Kenny Wong who has provided much guidance and support. I would also like to thank Dr. Eleni Stroulia, Dr. Jim Hoover, Mengliao Wang, Zhu Pang, Andrew Neitsch, Marios Fokaefs, Koosha Golmohammadi, Bruce Matichuk, Jeremy Nickurak, Xiaodi Ke, and Aaron Luchko for their thoughts and feedback.

This work has been performed in collaboration with CA Labs and is supported by the Natural Sciences and Engineering Research Council of Canada. Without this I would never be able to embark on this journey.

Table of Contents

List of Tables

List of Figures

List of Symbols

Acronym	**Definition and location of first use**
IP	Internet Protocol, Section 2.1
CMDB	Configuration Management Database, Section 2.1
FUSE	Filesystem in Userspace, Section 2.2
PID	Process ID, Section 2.2
VFS	Virtual Filesystem, Section 2.2
LDAP	Lightweight Directory Access Protocol, Section 4.1
TLS	Transport Layer Security, Section 4.1
OID	Object Identifier, Section 4.4.2
SVM	Support Vector Machine, Chapter 5

Chapter 1

Introduction

In recent years, computer systems have become an integral part of most day to day activities in many organizations. Such systems provide services ranging from enterprise resource planning which automates the integration of management information across an organization, to e-commerce systems that conduct online trade. As the dependence on computer systems to run businesses increases, so does the importance of efficiently analyzing and fixing problems that arise. According to a survey, dealing with problems consumes an estimated 30-70% of IT resources, and failure diagnosis and recovery comprises 1/3-1/2 of the ownership cost of systems [39].

1.1 Motivation

Corrective maintenance is a form of system maintenance that aims to bring a failed system to operational status. It is performed upon receiving a failure report or after noticing a problem in the system. Such problems may include loss of critical data, unscheduled service interruption, or system security issues [11].

The process of corrective maintenance usually starts with collecting information relevant to the noticed problem or reported failure in order to pinpoint the root cause(s) before taking appropriate actions. This first step of corrective maintenance, known as failure diagnosis, remains one of the most time-consuming and expertise-dependent tasks [24], because it involves the exploration of failure symptoms including message patterns and error codes, which differ from one system/application to another. With increasing software size and complexity, human operators are overloaded with the information generated from the many components in such systems, and need assistance.

To assist with failure diagnosis, we propose an unobtrusive and adaptive approach. It does not need access to source code, or recompilation of the software or operating system. Also, it is generic, as it neither assumes the system is based on any particular types of components nor relies on any supporting documents like problem tickets, system models, or existing log files. Our approach determines the failure causes by looking at differences in what and how files are accessed under normal and abnormal situations. For Unix/Linux systems, which regard everything as a file [26], filesystems are often involved in some way when the software misbehaves. To deal with software failures in a multi-filesystem and multi-process system, we monitor how the software behaves through its file-related calls on an instrumented filesystem. When a failure occurs, these calls are abstracted and classified to identify the likely cause.

Software failures may stem from defects in code and configurations. Ideally, corrective maintenance activities such as developing and deploying solutions to address the failures should resolve the problems permanently. In practice, however, failures often recur. As mentioned in [2, 20, 8], most reported software failures are caused by previously reported defects. Maintenance personnel often cannot stop a defect from repeatedly causing the same type of failures, due to several reasons [21] (see Figure 1.1(a)):

1. Corrective maintenance activities, including failure diagnosis and solution development, can take a large amount of time.

2. Solution deployment may be postponed if it requires a scheduled service outage.

3. The deployed solution is ineffective.

4. System administrators do not deploy the available solution in time due to concerns about introducing new problems.

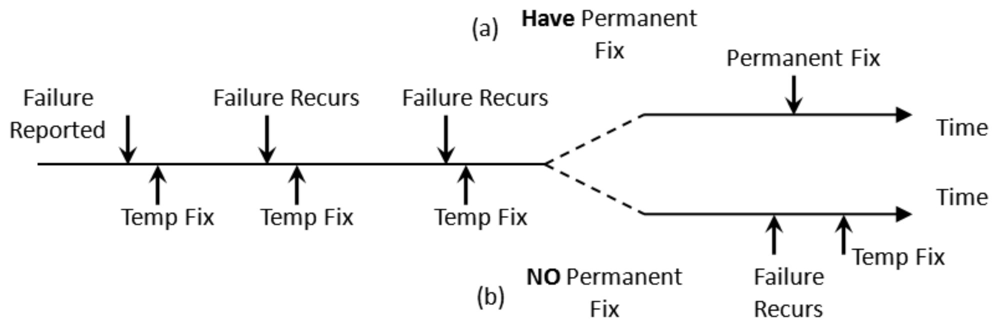


Figure 1.1: When Failures Recur

Moreover, failures are recurrent in some very old software systems where "the resistance of the program to change is at its maximum" (referred to as "software fatigue") [13], because it would be rather expensive to address the problems permanently (see Figure 1.1(b)).

Most of the time, the maintenance personnel have to resort to simple incident resolution to quickly respond to the problems [29] before the deployment of permanent solutions (see Figure 1.1). Those quick and simple incident fixes are usually non-code changes that make the system temporarily operational rather than permanently resolving the problem. For example, a quick incident resolution to a "database table locked" problem is unlocking the table with database administrator privilege, which would not prevent the same problem from happening in another scenario. Therefore, if a recurrent failure can be identified accurately, the appropriate quick fix may already be known from collected experience about past failures with similar causes. With the assistance of our approach, the maintenance personnel would be able to identify the type of recurrent failures. Also, they can look into the faulty filesystem activities to get a clue of where to start the investigation.

1.2 Contribution

The contributions of this work are as follows:

1. We use an unobtrusive and adaptive system profiling technique to keep track of the file-related calls. It does not require source code access or recompilation of software and the operating system; and does not rely on supporting

documents, as it logs its own events by itself. Although it is implemented on Linux in this research, it theoretically works on several operating systems, namely, Linux, FreeBSD, NetBSD, OpenSolaris, and Mac OS X.

2. We study the symptoms and effects in file-related calls when different types of failures occur, and build a classifier to distinguish failures that are non-performance-related.

3. We develop a preprocessing scheme for the file-related call traces. It creates canonical traces with less data redundancy.

4. We set up a system that provides web services to serve as the test bed. We programmatically generate random traffic with a load generator, and manually inject faults in order to quantitatively evaluate the effectiveness of our approach.

1.3 Summary

In industry a large proportion of failures are due to previously reported defects, and failure diagnosis is time-consuming and expertise-dependent. Thus identifying recurrent failures helps the maintenance personnel to determine what prior solutions may be appropriate. Our approach, which is based on file-related calls analysis, assists human operators by suggesting the type of problem when a software failure is reported.

The rest of the thesis is organized as follows. Chapter 2 provides background on failure diagnosis and profiling techniques, and explains why techniques used in this work were chosen. Chapter 3 presents our failure diagnosis approach in detail, followed by Chapter 4 which describes the experiment we conducted to quantitatively evaluate our approach. Chapter 5 relates this work to that of others, and Chapter 6 concludes our work and provides future directions.

Chapter 2

Background

As described in Chapter 1, we collect system run-time information from file-related calls, from which our approach suggests the failure types by classifying the collected information. This chapter provides background knowledge on software diagnosis in Section 2.1 and system profiling tools/techniques in Section 2.2.

2.1 Software Failure Diagnosis

Failures occur in almost all systems due to different types of incidents that cause one or more components to perform in unexpected ways. Software failures in industry are expensive, and the increasingly stringent requirements on performance and reliability of enterprise systems have made the diagnosis of software failures crucial and challenging. The tasks of failure diagnosis, as defined in [18], include figuring out what caused a failure, or identifying the type or origin of a failure. For example, the system warning message in the box at the top of Figure 2.1 indicates the existence of problem(s) with the specified host or its related component(s), but the possible root causes are many. For instance, the hard disk on the specified host failed, the server software running on the specified host failed, or some human operator misconfigured the port number or IP (Internet Protocol) address. To bring the system back to the normal state, diagnosticians need to collect additional information, analyze, and find the root cause(s), which is the process of failure diagnosis.

Manual failure diagnosis in enterprise systems is difficult and inefficient because it takes time for human operators to learn the failure symptoms and adapt to

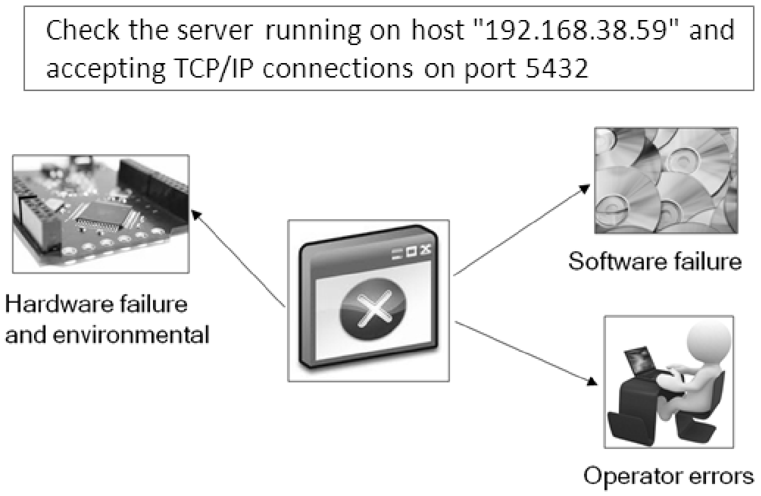

Check the server running on host "192.168.38.59" and accepting TCP/IP connections on port 5432
Hardware failure and environmental
Software failure
Operator errors

Figure 2.1: Failure and Possible Causes: An Example

unfamiliar components. With the emergence of concepts like autonomic computing, the interest on using system runtime information to assist in failure analysis continues to increase [37]. System event traces (e.g., system call traces, method call traces) and log files (e.g., server logs, incident reports, change logs) are two major sources of system runtime information. Documentation (e.g., CMDB (Configuration Management Database) information, bug reports) and configuration files are also involved because they either record the architecture of system or imply the history of system states. To use the above information for software diagnosis, researchers proposed four types of failure diagnosis methods:

1. Rule-based approach (e.g., [15]): Set up and maintain a set of inferences, from which to generate hypotheses for the root causes when a failure occurs. For example, if the command *ping ca-ssa* fails, then the host named *ca-ssa* fails.

2. Model-based approach (e.g., [34]): Build a system model to help generate diagnostic results when a failure occurs. For example, if an application runs on a host and the host fails, then the application also fails.

3. Statistical approach (e.g., [35] and [16]): Find out statistical correlations between causes and failure symptoms, from which to produce rules and models.

4. Integrated (e.g., [23]): Combines at least two of the above into the investigation workflow.

Rather than grab information from existing logs, documentation, or configuration files that vary in format, content, and use, our approach, which is statistical, logs its own event traces in the form of file-related calls to provide a generic way of performing diagnosis (see Chapter 3 for details).

2.2 Profiling Techniques

This section reviews generic profiling techniques to trace software application behavior, and explains why we use FUSE (Filesystem in Userspace) [38] in this research.

Gprof [12] and OProfile [22] are two of the most commonly used Linux profilers. Both of them are open source profiling tools which record the usage of functions at the code level. The primary use is to monitor program performance by collecting information regarding function executions. Using function execution data, the behaviors of software applications can be modeled. However, gprof is unable to monitor software without first instrumenting and recompiling the source code, or work with multi-threaded code. And the problem with using OProfile is that it does not record low-level filesystem activities.

The *inotify* [25] facility is a Linux kernel subsystem. An *inotify* watch point, which restricts the effective scope of inotify, can be added to one directory, or recursively to one directory and its subdirectories. For a watch point, *inotify* monitors all *inode* changes, and thus is able to report all filesystem activities within the effective scope. It is easy to use; however, the PIDs (process IDs) cannot be gathered directly, so the data cannot be decomposed by process. This makes distinguishing filesystem activities difficult in an environment with multiple concurrent processes.

For Linux, the *ptrace* [14] facility can be used for recording filesystem changes. By attaching *ptrace* to a process, programmers can observe the argument(s) and return value(s) of every system call made by the monitored process. A problem with this approach is that all system calls would be trapped, even if they are not

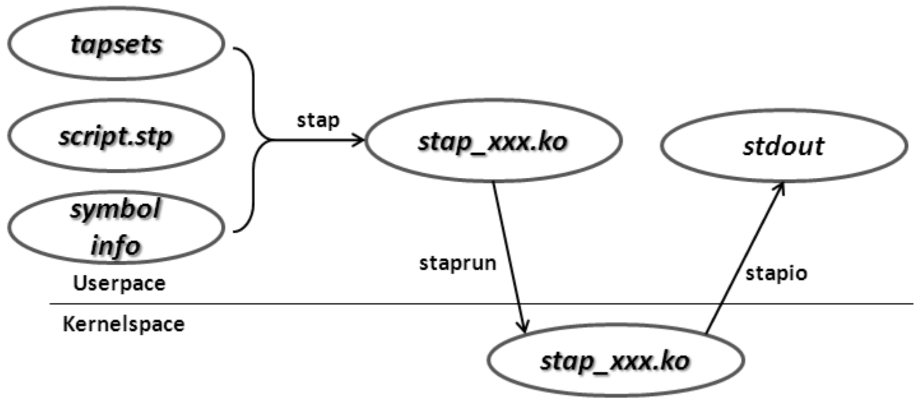


Figure 2.2: The SystemTap Process

filesystem related. This results in overhead on all system operations.

DTrace [6] and SystemTap [17] are kernel profiling systems, which allow programmers to write scripts containing relatively complex logic to decide which events to profile. However the support for DTrace in Linux systems is quite limited compared to Solaris or Mac OS X. For SystemTap, as shown in Figure 2.2 [19], it translates *script.stp*, a script written in SystemTap probe language, to C, and then creates a kernel module (i.e., *stap_xxx.ko*) by running the C compiler. Once the kernel module is loaded and all probes are enabled, the corresponding events would be reported when they occur. The problem with using SystemTap is that it takes time to place probes upon newly created directories, causing some loss of events to be captured.

FUSE [38], which provides an alternative way of monitoring filesystems, is what we use for system monitoring in this work. It is a loadable kernel module that allows "non-privileged users to create their own filesystems without touching kernel code" [38]. In a system without FUSE (see Figure 2.3(a)), different types of concrete filesystems (e.g., EXT, NFS) are plugged into VFS (Virtual Filesystem) interface. When software applications need to access concrete filesystems, it is FUSE's responsibility to talk to the concrete filesystems and complete the file accesses. In a system with FUSE module being loaded (see Figure 2.3(b)), we mount concrete filesystems to FUSE. And at this point, VFS is no longer able to talk to concrete filesystems directly when software applications make a call – what VFS does is to send VFS calls to FUSE, which replaces VFS calls with FUSE calls that

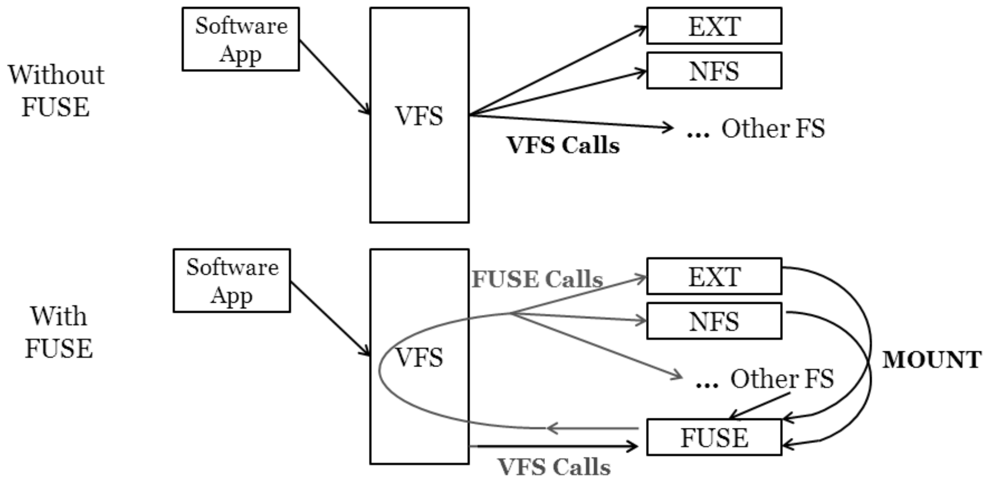


Figure 2.3: How FUSE Works

are implemented by programmers.

We can distinguish VFS calls made by processes that we monitor. FUSE only traps the VFS calls we specify, so it does not result in as much overhead as *ptrace* does. Also, FUSE is unobtrusive as no source code access or recompilation of software is required of . Unlike SystemTap which traces the filesystem operations by trapping all the concerned system calls, FUSE uses a much smaller set of functions to represent activities on the filesystem. For example, in the behavior of reading a file, the variety of available system calls for reading could make data normalization complicated, because reading data from the same file can be implemented by calling different combinations of *read*, *readv*, *pread*, or even *mmap* and *sendfile*. FUSE uses only one function *fuse_read* to perform precisely the same operations.

2.3 Summary

Software diagnosis methods may use system event traces, log files, documentation and configuration files, from which to model the behaviors of systems/applications, and/or identify root causes of failures. Our approach is based on system event traces in the form of file-related calls, which are captured by a FUSE-instrumented filesystem.

The following chapters provide details of our approach, which is subsequently evaluated for its effectiveness.

Chapter 3

Approach

When a failure is reported, our approach determines the failure type by analyzing the file-related calls that are logged by our instrumented filesystem. These calls represent partial behaviors of software. In this chapter, we describe the architectural design and implementation of our approach for capturing the file-related calls, modeling software behaviors and failure diagnosis.

Figure 3.1 shows the high-level architecture of our approach, which mainly contains two parts: off-line training (i.e., steps enclosed by dashed lines) and on-line failure diagnosis (i.e., steps outside the area enclosed by dashed lines). And the instrumented filesystem, which serves as the data collector, is used to capture the file-related calls at run time to provide call sequences for both off-line training and on-line failure diagnosis.

The training process involves the following activities:

1. Data collection and labeling: we keep track of the file-related calls in different abnormal situations caused by artificially injected faults. These call sequences record how the monitored software behaves with respect to the filesystem when failures occur (see step 1 in Figure 3.1). Each trace is labeled by its type of failure once collected.

2. Pattern mining: after preprocessing the raw data, the pattern discovery method is applied to the call sequences to discover frequent call sequences, which are considered as "featured behaviors". These sequences are then used to transform call sequences (collected from both off-line and on-line uses) into

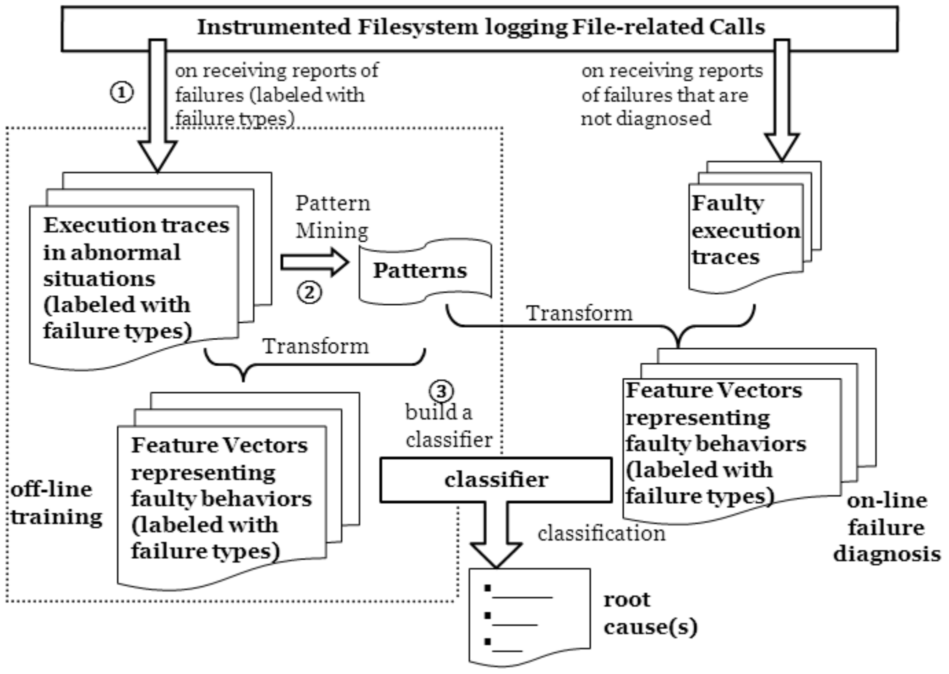

Instrumented Filesystem logging File-related Calls
①
on receiving reports of failures (labeled with failure types)
on receiving reports of failures that are not diagnosed
Execution traces in abnormal situations (labeled with failure types)
Pattern Mining
②
Patterns
Faulty execution traces
Transform
Transform
③
build a classifier
Feature Vectors representing faulty behaviors (labeled with failure types)
Feature Vectors representing faulty behaviors (labeled with failure types)
classifier
off-line training
classification
on-line failure diagnosis
root cause(s)

Figure 3.1: Approach Flow Graph

feature vectors (see step 2 in Figure 3.1).

3. Building a classifier: Research in machine learning shows that Naive Bayes is efficient and accurate in text classification [28, 3]. And in practice, Naive Bayes is widely used in categorizing texts, for example, filtering out the spam, identifying appointment messages, and so on. Since the execution traces we logged are non-numerical (see Section 3.2) and the aim of our approach is to correlate the texts and labels (i.e., failure causes), we build a Naive Bayes classifier to classify transformed call sequences in the on-line failure diagnosis process (see step 3 in Figure 3.1).

As shown in Figure 3.1, the failure diagnosis process, which is started after a failure is reported, relies on three things:

- the file-related call traces that are logged by our instrumented filesystem at run time, which reflect the behaviors of the monitored software running in faulty states;
- the patterns that are mined offline from a collection of labeled call sequences, which are used to transform the run-time trace data into feature vectors; and

- the classifier built in the off-line process upon the labeled traces.

On receiving a failure report of the monitored software, our approach looks into the file-related call trace, starting from a certain time prior to the point that the problem is noticed (the selection of an effective starting point is studied in Chapter 4). The call trace is transformed into a feature vector by the trace transformer, analyzed and ultimately assigned the most probable root cause by the classifier built in the off-line training process.

In the following sections, the technical details of our approach are described: Section 3.1 describes the implementation of our instrumented filesystem and the setup of the data collection environment; Section 3.2 presents the method applied for raw data preprocessing; Section 3.3 depicts the algorithm we used for call sequence pattern discovery; Section 3.4 explains how to transform call traces into feature vectors; and Section 3.5 describes failure diagnosis with the classifier we build.

3.1 Capturing File-related Calls

We use file-related calls to model behaviors of the monitored software. We developed TraceFS with FUSE to instrument a running Linux system to collect information on which files are accessed and how. With the loadable FUSE module, this process only requires restarting the system, and thus is not as obtrusive as those that need to recompile the operating system or software application.

Figure 3.2 shows how TraceFS works. The FUSE module, which resides in the kernel space, redirects normal VFS calls made by the monitored software to TraceFS, where the details of those VFS calls are logged, and precisely the same operations on the target files are made in userspace. For example, suppose the monitored software reads the file */lib/libdl-2.5.so*. The following occurs:

1. The monitored software calls VFS *read* via a function in *glibc* (see step 1 in Figure 3.2),
2. VFS *read* calls a corresponding function in the FUSE module (step 2), which passes the *read* call to TraceFS via the *libfuse2* library (step 3),

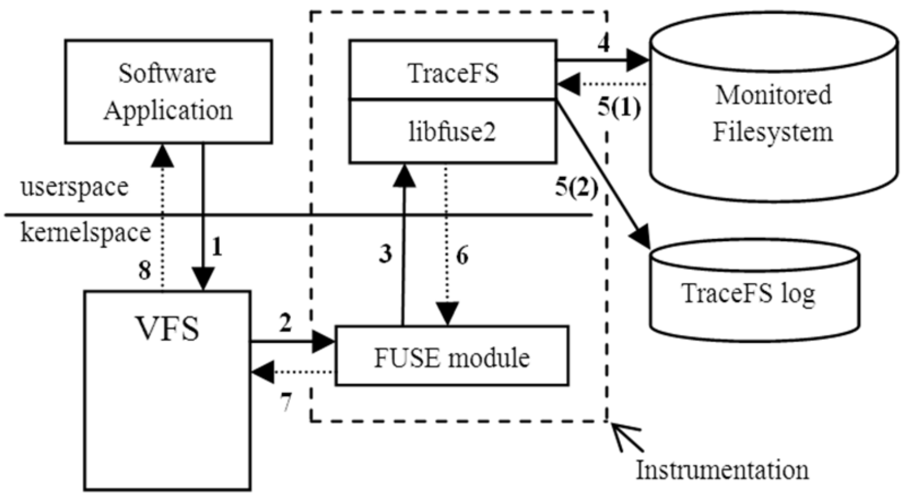


Figure 3.2: Capturing File-related Calls with TraceFS

3. Our implementation in TraceFS collects parameters passed with VFS *read*, simulates VFS *read* by reading the file */lib/libdl-2.5.so* (step 4) via *glibc* (i.e., TraceFS reads data from */lib/libdl-2.5.so* into the specified buffer.), and keeps the collected parameters in the TraceFS log file (step 5). See Figure 3.3.

4. The return value of the *read* call is sent back via TraceFS, *libfuse2*, the FUSE module, VFS, and finally to the monitored software (steps 6, 7, and 8).

With the FUSE module, TraceFS is a fully functional filesystem that resides in the userspace. The file-related calls that TraceFS monitors are listed in Table 3.1. This covers all filesystem operations related to link operations, file/directory property changes, directory operations, and regular file operations.

To get the complete picture of run-time filesystem operations, all filesystems of interest should be mounted to a mount point bound to TraceFS. We used the method in [31], which starts the initial process of the operating system (e.g., */sbin/init* in Centos 5.5) by using the *chroot* call to change the root directory of *init* and all its child processes to that mount point, so as to have the entire operating system run within our data collection environment.

Nevertheless, operations in some directories are not of interest but should still be accessible to the monitored software application running within our environment. These directories, namely, */dev*, */sys*, */proc*, and */tmp*, are special in Linux. Files

```c
/** Read data from an open file*/

int my_read(const char *path, char *buf, size_t size, off_t
    offset, struct fuse_file_info *fi) {
  int retstat = 0;
  time(&now);
  timenow = localtime(&now);

  retstat = pread(fi->fh, buf, size, offset);
  if (retstat < 0) {
    retstat = bb_error("bb_read read");
    log_msg("%d|ERROR(%d)|read|%s|%s", myGetpid(fi), retstat,
        path, asctime(timenow));
  }
  else
    log_msg("%d|NORMAL|read|%s|%s", myGetpid(fi), path, asctime
        (timenow));

  return retstat;
}
```

Figure 3.3: Reading a File in FUSE

in */dev* are interfaces for communication with devices, while files in */sys* and */proc* are either used to record driver classes or information regarding running processes. Directory */tmp* is used for temporary files storage. And keeping track of file-related calls in */tmp* will introduce much noise in the data because the names of some files in this directory may constantly change. Therefore, we remount these special directories to the same relative paths within TraceFS as they were in the original VFS before starting the initial process *init*.

A startup script is used to configure the operating system and start the initial process *init* within our data collection environment by using a "init=$SCRIPT_PATH" option (see Figure 3.4). In this way, information regarding filesystem operations that take place in the system is collected as much as possible.

3.2 Data Preprocessing

The raw logs produced by TraceFS are actually operation-by-operation traces. As shown in Figure 3.5 which contains a clip of a TraceFS log, each file-related call contains 5 fields: PID (process ID), return value of the call, name of the call, tar-

```
#!/bin/sh
TRACEFS=/root/fuse/fuse-tutorial/src/bbfs
TRACEMNT=/root/mnt
FALLBACK=/bin/bash
TRACEPID=""
# list of directories to remount under TraceFS
IGNOREDIRS='/dev /proc /sys /tmp';

fallback(){
    echo $1
    echo "starting fallback shell:\n"
    exec $FALLBACK
}
if [ $$ -ne 1 ]; then
    echo "This script is intended to run as an init process
        only (PID=1)\nExiting..."
    exit
fi

# Make sure the root filesystem is read-write, and setup FUSE
mount -o remount,rw /
    || fallback "Failed to remount / as read-write\n"
/sbin/insmod /lib/modules/2.6.18-194.el5/kernel/fs/fuse/fuse.ko
     || fallback "Failed to load FUSE module\n"
if [ ! -c /dev/fuse ]; then
    mknod /dev/fuse c 10 229
        || fallback "Failed to create /dev/fuse\n"
fi

# Start gathering statistics from / by mounting on $TRACEFSMNT,
# and recording the PID of the statistic tracker fuse
# application.
mkdir -p $TRACEFSMNT || fallback "Failed to create directory
    $TRACEFSMNT for TraceFS mountpoint\n"
$TRACEFS / $TRACEFSMNT & TRACEFSPID=$!
[ -n "$TRACEFSPID" ]
    || fallback "Failed to start \"$TRACEFS\"\n"
sleep 12

# Remount the following directories to ensure we don't see
# their usage in our TraceFS logs.
for DIR in $IGNOREDIRS; do
    mount --rbind $DIR $TRACEFSMNT$DIR
        || fallback "Failed to remount $DIR inside TraceFS\n"
done

# Pass off execution to the system's normal init process
/usr/sbin/chroot $TRACEFSMNT /bin/ls /etc/init.d
    || fallback "Failed to start /sbin/init\n"
exec /usr/sbin/chroot $TRACEFSMNT /sbin/init 3
```

Figure 3.4: Startup Script for the Tracing Environment

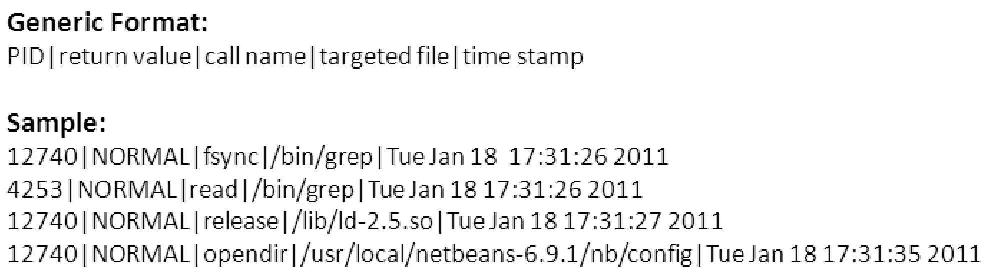
Table 3.1: Monitored File-related Calls

Call	Function
symlink	create a symbolic link
readlink	read the target of a symbolic link into a buffer
unlink	delete a name from the filesystem, and the referred file if no process has the file open
link	create a hard link to a file
mknod	create a file node
rename	rename a file
chmod	change the permission of a file
truncate	change the size of a file
chown	change the owner of a file
utime	change the access or modification time of a file
create	create and open a file
open	open a file
read	read data from an opened file into a buffer
write	write data to an opened file
fsync	synchronize the content of a file
release	release an opened file
access	check access permission of a file
mkdir	create a directory
rmdir	remove a directory
opendir	open a directory
readdir	read a directory structure into a buffer
releasedir	release an opened directory

geted file, and time stamp. Once the calls are captured, some preprocessing needs to be done to the collected traces in order to reduce noise and data redundancy:

- Split the traces by PID: To collect data from an environment where multiple concurrent processes are running, we use the PID to decompose the file-related calls. In other words, we use PID to discard the file-related calls that are made by processes we do not monitor (see rule 1 in Figure 3.7 for example). This reduces irrelevant data from being involved in the analysis, however it also rules out the possibilities that failures are caused by inter-process communications (e.g., signals, message queues).

- Shorten the sequences containing consecutive successful *access* calls generated during path resolution. For example, checking the permission of */bin/ls* will generate three *access* calls, i.e., */*, */bin*, and */bin/ls*. In this case, we use



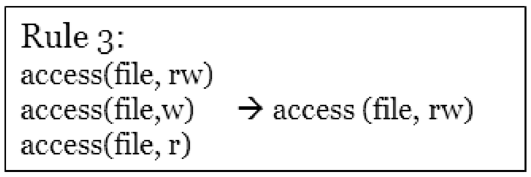
Generic Format:

PID|return value|call name|targeted file|time stamp

Sample:

```
12740|NORMAL|fsync|/bin/grep|Tue Jan 18  17:31:26 2011
4253|NORMAL|read|/bin/grep|Tue Jan 18 17:31:26 2011
12740|NORMAL|release|/lib/ld-2.5.so|Tue Jan 18 17:31:27 2011
12740|NORMAL|opendir|/usr/local/netbeans-6.9.1/nb/config|Tue Jan 18 17:31:35 2011
```

Figure 3.5: Format of Collected Traces



Rule 3:

access(file, rw)
access(file,w) → access (file, rw)
access(file, r)

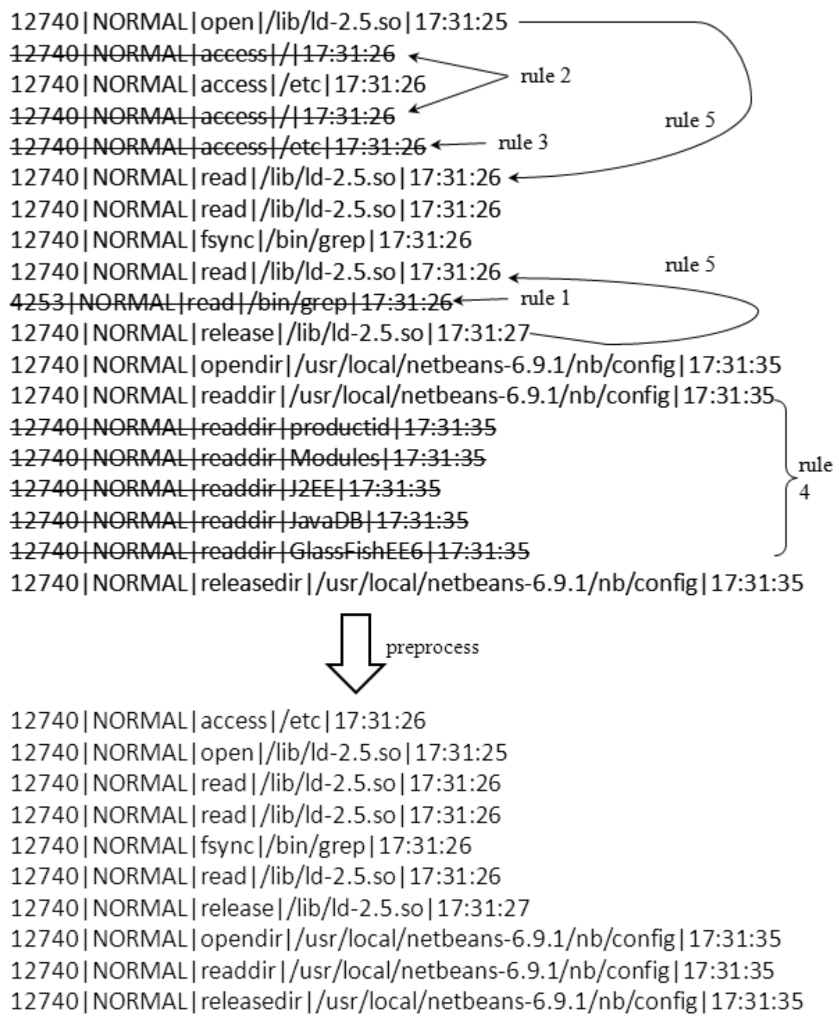
Figure 3.6: Combine Multiple *access* Calls into One

the call for */bin/ls* to represent all three (see rule 2 in Figure 3.7).

- Shorten the sequences containing consecutive *access* calls with identical return values. To reduce data redundancy, we combine repeated consecutive access calls returning the same value with one call (see rule 3 in Figure 3.7), as containing consecutive *access* calls with identical return values can always be combined into one call (see Figure 3.6).

- Shorten the sequences containing *readdir* calls that aim to read the structure of a directory into the buffer. In the trace data, a directory associated with *readdir* includes the one that *readdir* aims to read into the buffer, and its subdirectories. We discard the *readdir* calls for the subdirectories to reduce the level of data redundancy (see rule 4 in Figure 3.7).

- Move successful *open* calls downward to the position just before the closest *read* or *write* call targeted on the same files, and successful *release* calls upward to the position just after the closest *read* or *write* call targeted on the same files. A file must be opened prior to any *read* and *write* operations. In other words, reading and writing can be applied to a file as long as it is open, and the time that it was opened is not important. Similarly, when reading and writing (and synchronization) are finished, calling *release* sooner or



```
12740|NORMAL|open|/lib/ld-2.5.so|17:31:25
~~12740|NORMAL|access|/|17:31:26~~
12740|NORMAL|access|/etc|17:31:26
~~12740|NORMAL|access|/|17:31:26~~
~~12740|NORMAL|access|/etc|17:31:26~~
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
12740|NORMAL|fsync|/bin/grep|17:31:26
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
~~4253|NORMAL|read|/bin/grep|17:31:26~~
12740|NORMAL|release|/lib/ld-2.5.so|17:31:27
12740|NORMAL|opendir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
12740|NORMAL|readdir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
~~12740|NORMAL|readdir|productid|17:31:35~~
~~12740|NORMAL|readdir|Modules|17:31:35~~
~~12740|NORMAL|readdir|J2EE|17:31:35~~
~~12740|NORMAL|readdir|JavaDB|17:31:35~~
~~12740|NORMAL|readdir|GlassFishEE6|17:31:35~~
12740|NORMAL|releasedir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
```

```
12740|NORMAL|access|/etc|17:31:26
12740|NORMAL|open|/lib/ld-2.5.so|17:31:25
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
12740|NORMAL|fsync|/bin/grep|17:31:26
12740|NORMAL|read|/lib/ld-2.5.so|17:31:26
12740|NORMAL|release|/lib/ld-2.5.so|17:31:27
12740|NORMAL|opendir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
12740|NORMAL|readdir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
12740|NORMAL|releasedir|/usr/local/netbeans-6.9.1/nb/config|17:31:35
```

Figure 3.7: The Preprocessing of a Clip of Call Traces

later does not make any difference to the data stored in the file. So moving successful *open* and *release* calls reduces noise and creates a more canonical sequence of calls with the same effect (see rule 5 in Figure 3.7).

- For the same reason of moving *open* and *release* calls, move successful *opendir* calls downward to the position just before the closest *readir* call targeted on the same directory, and successful *releasedir* calls upward to the position just after the closest *readir* call targeted on the same directory.

Once the call traces are preprocessed following the above strategy, they are used for pattern discovery, or transformed into feature vectors.

3.3 Pattern Discovery from File-call Traces

This section presents the pattern discovery process in this work. Section 3.3.1 defines the terms in our pattern discovery problem, and Section 3.3.2 provides the pseudo code.

3.3.1 Definitions

A **call sequence** S is an ordered list of file-related calls, denoted by $c_1c_2...c_n$, where c_i $(1 \leq i \leq n)$ is one call having 5 data fields (i.e., PID, return value of the call, name of the call, targeted file, and time stamp). Two file-related calls c_m and c_n are considered equal (denoted as $c_m = c_n$) if they have identical PID, return value, name, and targeted file (although they may have different effects in different contexts). From the definition we know that each call trace is a call sequence, and constituent calls are not necessarily distinct.

The number of calls in a call sequence is called the **length** of the call sequence. If call sequence S contains l calls, S is an l-sequence, i.e., $|S| = l$.

Given two call sequences $S_a = a_1a_2...a_m$, $S_b = b_1b_2...b_n$. S_a is properly contained by S_b if $m < n$ and there exist an integer i $(1 \leq i \leq n - m + 1)$ such that $a_1 = b_i$, $a_2 = b_{i+1}$, ..., $a_n = b_{i+n-1}$. S_a is called the **call subsequence** of S_b, and S_b the **call super-sequence** of S_a (denoted as $S_a \sqsubset S_b$), if S_a is contained in S_b.

We use T to denote the training set which is used as the input of the pattern discovery task. T contains the call sequences (labeled by the type of failure causes).

The **support** of a call sequence S_a in T, denoted by $sup(T, S_a)$, is the number of times that S_a appears as a call subsequence of the call sequences in T. And S_a is considered as a **frequent call sequence** in T if $sup(T, S_a) \geq min\_sup$, where min_sup is a support threshold.

If S_a is a frequent call sequence and there does not exist any call super-sequence of S_a with the same support, i.e., $\neg\exists S_b$ such that $S_a \sqsubset S_b$ and $sup(T, S_a) = sup(T, S_b)$, we call S_a a **maximal frequent call sequence**.

The problem of pattern discovery is to find the complete set of maximal frequent call sequences, denoted by FCS with a given min_sup from a set of call sequences.

In this work, the length of each preprocessed call sequence increases by at least one hundred calls per minute. To reduce the run time of our pattern mining task, we set a threshold $lmax$ for the maximum length of call sequence patterns. Thus the problem of pattern discovery in this work becomes finding the complete set of maximal frequent l-sequences, $3 \leq l \leq lmax$, with a given min_sup from the call traces.

```
PatternDiscovery($T$, $min\_sup$, $lmax$) : $FCS$
Input: the set of all call traces $T$, the support threshold $min\_sup$, the pattern length threshold$lmax$
Output: the set of maximal frequent call sequences $FCS$
  $l = lmax$
  $FCS = \emptyset$, $tmpList = \emptyset$
  while $l \geq 3$ do
      for each call trace $S$ in $T$ do
          $pos = 0$
          while $pos \leq |S| - l$ do
              $S\_tmp = l$ consecutive calls starting from the position $pos$ in $S$
              if $\neg\exists S\_tmp$ in $tmpList$ then
                  add $S\_tmp$ to $tmpList$, $sup(T, S\_tmp) = 1$
              else
                  $sup(T, S\_tmp) = sup(T, S\_tmp) + 1$
              end if
              $pos = pos + 1$
          end while
      end for
      for each $S_a$ in $tmpList$ do
          if $sup(T, S_a) \geq min\_sup$
          and isContain($S_a$,$FCS$,$T$)=false then
              add $S_a$ into $FCS$
          end if
      end for
      $l = l - 1$
      $tmpList = \emptyset$
  end while
  return $FCS$

isContain($S_a$, $FCS$, $T$) : $contain$
Input: call sequence $S_a$, the set of maximal frequent call sequences $FCS$, the set of all call traces
Output: a boolean value $contain$ indicating whether or not there exist a call super-sequence of $S_a$ with the same support in
$FCS$
  $contain = false$;
  for each call sequence $S_b$ in $FCS$ do
      if $S_a \sqsubset S_b$ and $sup(T, S_a) = sup(T, S_b)$ then
          $contain = true$
          break
      end if
  end for
  return $contain$
```

Figure 3.8: Pattern Discovery Pseudo Code

3.3.2 Algorithm

As stated at the end of Section 3.3.1, the aim of our pattern discovery method is to find all maximal frequent call subsequences from the call sequences. As the

```
**TraceTranfomation**($S$, $lmax$, $FCS$) : $V(S, FCS)$)
**Input**: a to-be-transformed call trace $S$, the pattern length threshold $lmax$, the set of maximal frequent call sequences $FCS$
**Output**: a feature vector $V(S, FCS)$
  $l = lmax$
  $n = |FCS|$
  $V(S, FCS) = \langle f_1, f_2, ..., f_n \rangle, f_i = 0, \forall f_i, 1 \leq i \leq n$
  while $l \geq 3$ do
      $pos = 0$
      while $pos \leq |S| - l$ do
          $S\_tmp = l$ consecutive calls starting from the position $pos$ in $S$
          if $\exists S\_tmp$ in $FCS$ then
              $pos =$ pattern ID of $S\_tmp$
              $f_{pos} = f_{pos} + 1$
          end if
      end while
      $l = l - 1$
  end while
  return $V(S, FCS)$
```

Figure 3.9: Trace Transformation Pseudo Code

pseudo code for pattern discovery given in Figure 3.8 shows, each call sequence is scanned for $(lmax - 3)$ times. Each time, every call sequence is scanned by a sliding window of length l, and the $support$ of all l-sequence are counted. Once a scan finishes, the set of frequent l-sequences is merged into the set of maximal frequent call sequences FCS. A frequent l-sequence is added to FCS only if it does not have any call super-sequence in FCS with the same support. Ultimately the complete set of maximal frequent call sequences is obtained.

3.4 Call Sequence Transformation

With the complete set of maximal frequent call sequences, we transform a call sequence into a vector of integers (see Equation 3.1). Compared with transforming into a vector of bits (0s and 1s), integers indicates the times that the corresponding maximal frequent call sequence is present, and preserve richer information regarding the call sequence.

Let p denote an element of the set of maximal frequent call sequences (i.e., $p \in FCS$), let $\mathbb{S}$ denote the set of call traces to be transformed, and an individual call trace $S \in \mathbb{S}$ can be transformed into a feature vector

$$V(S, FCS) = \langle f_1, f_2, ..., f_n \rangle, \tag{3.1}$$

where $n = |FCS|$, and f_i denotes the number of times that p_i appears in S (i

denotes the index of p in FCS, $1 \leq i \leq n$).

Example: Denoting each unique call with a different letter, we have 3 call traces in Table 3.2. Given $min\_sup = 4$ and $lmax = 6$, the complete set of maximal frequent call sequences comprises 4 elements, i.e., FCS ={ABCAAA ($sup = 4$), ABCDA ($sup = 4$), AAA ($sup = 6$), ABC ($sup = 8$)}. Accordingly, with FCS, 3 call traces in Table 3.2 are transformed into 4-bit feature vectors $V(1, FCS) = \langle 1, 1, 1, 2\rangle$, $V(2, FCS) = \langle 1, 1, 2, 2\rangle$, and $V(3, FCS) = \langle 2, 2, 3, 4\rangle$, respectively. ■

The pseudo code of call sequence transformation is given in Figure 3.9.

Table 3.2: An Example Set of Call Sequences

ID	Trace Data
1	A B C D A B C A A A
2	A B C A A A B C D A A A
3	A B C D A B C A A A A B C D A B C A A A

3.5 Learning Faulty Software Behaviors

As discussed at the beginning of Chapter 3, call sequences are collected to build a classifier, which is later used for diagnosis (i.e., classification) when a failure occurs. In practice, the call sequences collected under faulty states also contain normal behaviors of the monitored software. Thus knowing how the software behaves in its normal state is necessary, as it helps to distinguish the events that only happen when failures occur and in this way improve the performance of classifier we build.

Since patterns in FCS are considered as featured behaviors of monitored software, we transform call sequences with the pattern set ($FCS_{faulty} - FCS_{normal}$), where FCS_{faulty} denotes the set of maximal frequent call sequences from call sequences collected under abnormal situations, and FCS_{normal} denotes the set of maximal frequent call sequences from the normal state. In other words, patterns that appear in the normal state are not used to build the classifier.

In this work, each call sequence is assigned only one class label, i.e., a failure type that has been already observed. We use Naive Bayes to build a classifier with the labeled call sequences. The classifier estimates the probabilities of failure types that a new call sequence could belong to, and the call sequence is assigned a failure type with the highest probability. For a feature vector $V(S, FCS) = \langle f_1, f_2, ..., f_{|FCS|} \rangle$, the decision rule is

$$class(V(S, FCS)) = P(R) \prod_{i=1}^{|FCS|} P(f_i|R), \tag{3.2}$$

where R denotes the failure type, i.e., the class label, and $P(f_i|R)$ is estimated assuming a Gaussian density function with mean μ_R and standard deviation σ_R computed from the i-th attribute values in class R.

3.6 Summary

To use file-related calls to model the software's behaviors, we implement TraceFS and set up the data collection environment. In order to distinguish different types of faulty states, we apply our pattern discovery algorithm on the preprocessed call traces collected by TraceFS, and obtain the complete set of maximal frequent call sequences, from which to transform the trace data and build a classifier.

To present the performance of this approach, Chapter 4 details the experiment design and experimental results.

Chapter 4

Evaluation

This chapter describes an experiment to evaluate our approach. The first three sections describe the design of our experiment, including the architecture of our test bed (Section 4.1), and the strategies applied in the normal state simulation (Section 4.2), fault injection, and training data collection (Section 4.3). The experimental results are presented in Section 4.4. Section 4.5 analyzes the limitations of this work and threats to validity of the experimental results. And Section 4.6 summarizes the whole chapter.

4.1 Test Bed

An enterprise software system is composed of a number of components, for instance, one or more web application servers for the deployment of web applications, database servers for storing the transaction data, file servers for data archives, an LDAP (Lightweight Directory Access Protocol) server for the user authentication, and so on. Each component in the system is a source of the file-related traces.

We observe the performance of our approach in a system that provides Web services. Our test bed, as shown in Figure 4.1, is composed of four major components.

1. Web application server: We use Glassfish v3 as the Web application server, where Duke's Bank, an open source J2EE application that emulates a web-based personal online banking system, is deployed.

2. Database server: We use Postgresql 8.3 to store the transaction records, bank account and customer-related information.

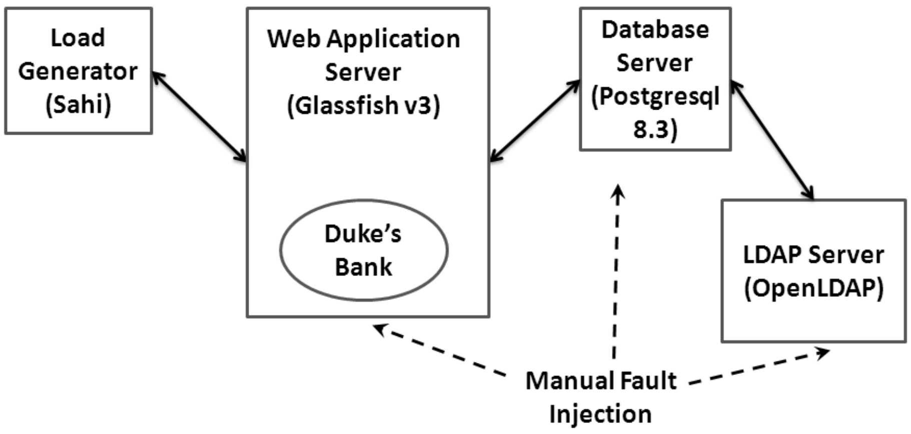


Figure 4.1: The Test Bed

```
1  function login_viewAccount(){
2      _setValue(_textbox("j_username"), "200");
3      _setValue(_password("j_password"), "javaee");
4      _click(_submit("Submit"));
5      _click(_link("Account List"));
6      _click(_link("Checking"));
7      _setSelected(_select("sortOption"), "Description");
8      _click(_submit("Update"));
9      _assertExists(_textbox("total"));
10     _assert(_isVisible(_textbox("total")));
11     _assertEqual( 1 8 5 0  , _getValue(_textbox("total")));
12 }
```

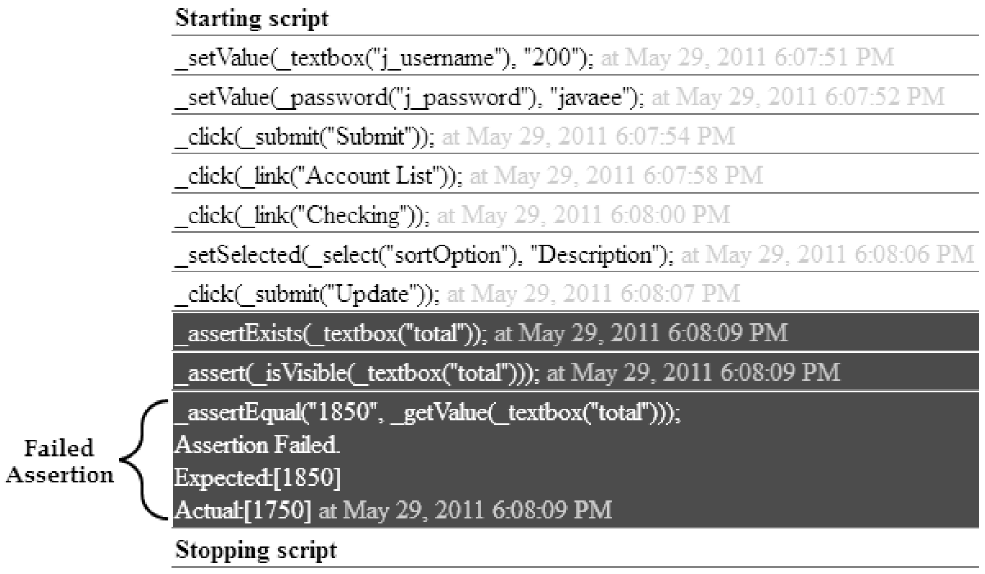
Figure 4.2: A Clip of a Sahi Script

3. LDAP server: OpenLDAP is installed for the user authentication of our database server. The user name and password pair is transmitted between the database server and LDAP server with TLS (Transport Layer Security) encryption enabled.

4. Load generator: We use Sahi [1], an open source web automation and test tool, for traffic generation (see Section 4.2 for the strategy of generating traffic in detail).

   Sahi is able to store execution profiles of web applications in the form of scripts by working as the proxy server of browsers that access the web applications. Line 1 to 8 in Figure 4.2 show an example of a Sahi script which records the process of “user login” and “view bank account information”.

Another feature of Sahi is the playback of recorded execution profiles. With Sahi scripts that store use cases of the web application (i.e., Duke's Bank), we programmatically use the playback feature to simulate interaction with a running system to produces trace data. We can add assertions (see Line 9 to 11 in Figure 4.2) to the end of page requests in Sahi scripts to check if any unexpected events happen (e.g., the displayed values are incorrect or improperly displayed). Figure 4.3 shows an example of a Sahi playback log, with the assertion checks highlighted. Failed assertions are recorded along with the expected and actual values. Those failed assertions are considered as failures in our experiment. And their time stamps indicate the exact time that failures happen.



```
Starting script
_setValue(_textbox("j_username"), "200"); at May 29, 2011 6:07:51 PM
_setValue(_password("j_password"), "javaee"); at May 29, 2011 6:07:52 PM
_click(_submit("Submit")); at May 29, 2011 6:07:54 PM
_click(_link("Account List")); at May 29, 2011 6:07:58 PM
_click(_link("Checking")); at May 29, 2011 6:08:00 PM
_setSelected(_select("sortOption"), "Description"); at May 29, 2011 6:08:06 PM
_click(_submit("Update")); at May 29, 2011 6:08:07 PM
_assertExists(_textbox("total")); at May 29, 2011 6:08:09 PM
_assert(_isVisible(_textbox("total"))); at May 29, 2011 6:08:09 PM
_assertEqual("1850", _getValue(_textbox("total")));
Assertion Failed.
Expected:[1850]
Actual:[1750] at May 29, 2011 6:08:09 PM
Stopping script
```

Figure 4.3: Sahi Logs: a Failed Assertion

4.2 Normal State Simulation

As discussed in Section 3.5, for off-line training, not only do we collect call sequences containing faulty behaviors but also keep track of the file-related calls when the system is running normally. Since the web application involved in this experiment is small and simple, we collect call sequences from the system running for 48 hours normally to obtain a set of featured behaviors that can well represent the normal state.

During this 48 hours, we generate the incoming user requests at a rate mimicking the curve of incoming traffic rate at *hulk02*, a web server in Princeton University Data Center [32] (see Figure 4.4). Since *hulk02* is not a server responsible for large file transfers, we assume the rate of user requests it receives fits the curve of incoming traffic rate. To simulate the rate of user requests normal state, in the first 24 hours we generate the user request rate mimicking the curve between Friday 4 am and Saturday 4 am at *hulk02*, and in the second 24 hours the curve between Saturday 4 am and Sunday 4 am. The rate of generated web page requests is adjusted every 15 minutes following

$$f(t) = \begin{cases} b + \frac{k_1}{\sqrt{18\pi}} e^{-\frac{(t-9)^2}{18}}, & 0 \le t \le 9 \\ b + \frac{k_1}{\sqrt{18\pi}}, & 9 < t \le 12 \\ b + \frac{k_1}{\sqrt{18\pi}} e^{-\frac{(t-12)^2}{18}}, & 12 < t \le 16 \\ b + \frac{k_1}{\sqrt{18\pi}} e^{-\frac{8}{9}}, & 16 < t \le 19 \\ b + \frac{k_1}{\sqrt{18\pi}} e^{-\frac{(t-15)^2}{18}}, & 19 < t \le 24 \\ b + \frac{k_2}{\sqrt{18\pi}} e^{-\frac{(t-36)^2}{18}}, & 24 < t \le 48 \end{cases}$$

where x denotes the time in hours. The maximum web page request rate that our load generator (i.e., Sahi) can achieve depends on the web server response time, the performance of the computer on which Sahi is running, and the content of Sahi scripts. In our environment, Sahi can generate less than 800 page requests per hour. So in the experiment, we let $b = 140$, $k_1 = 4000$, $k_2 = 1700$, and get the distribution of web page requests received by our web application server shown in Figure 4.5.

4.3 Fault Injection and Data Collection

Pertet et al. [33] performed a study on failures in enterprise Web Service systems and concluded that 80% of the failures are due to software failures and human errors. In order to get trace data from our test bed when failures occur, we choose 9 out of the 10 most common failure types summarized in [33]'s "software failures and human errors" list (the type of failures not chosen is supposed to occur in a

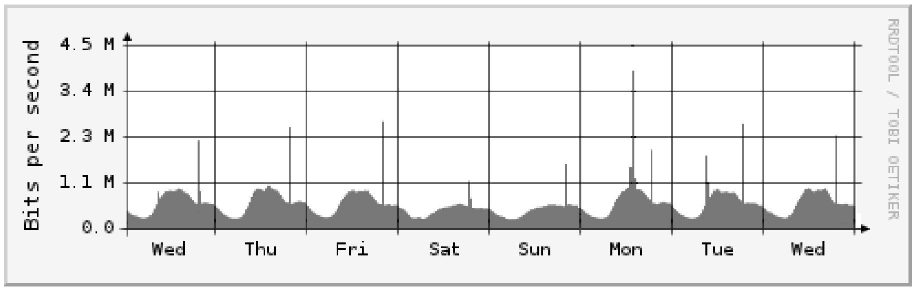


Figure 4.4: Incoming Traffic Rate at *hulk02*

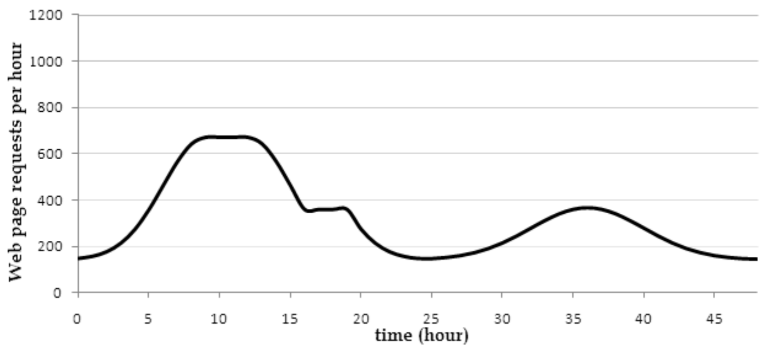


Figure 4.5: The Rate of Web Page Requests at the Normal State

cache server, which we do not have). We manually inject faults to the test bed to simulate those failures (see Table 4.1).

Table 4.2 shows how faults are simulated in this experiment. And Figure 4.6 illustrates our fault injection and data collection strategy:

- Faults are injected into our test bed while the web server receives a random rate of web requests, i.e., the rate of web requests is a random value that changes between the lowest and highest rate bound in "normal system simulation".

- We allow only one fault, which is randomly chosen, to be injected to the test bed each time. Once a failure is noticed, we collect the call sequence that starts from the time that fault injection finishes and ends when the failure is noticed (also the time associated with the corresponding failed Sahi assertion, as mentioned in Section 4.1).

- Recovery procedures are applied in order to make the test bed normal again. Data collected during fault injection and system recovery are discarded, be-

Table 4.1: Injected Faults

Faulty Component	Index	Root Cause
Web application server	1	Port used by Glassfish for the database server connection is changed
	2	The number of connections exceeds the upper limit of connection pool
Web application	3	Insufficient memory
Database server	4	One table in the database is locked
	5	The number of connections exceeds the upper limit of database server
	6	One table in the database is deleted
	7	Port used for connecting LDAP server is misconfigured
LDAP server	8	Some of the configurations are set to default values due to unsuccessful software upgrade
	9	Authentication fails due to the LDAP server outage

cause those are not part of the "problems". The time between the completion of system recovery and the next fault injection and is at least 3 minutes.

In this experiment, we only analyze file-related calls made by processes started by the web application server, the database server, and the LDAP server. We collect 20 traces for each type of fault. That makes 180 traces in total for the system running abnormally.

4.4 Experimental Results

This section presents the effectiveness of our approach from 3 aspects: the performance of our classifier in distinguishing different types of faults, the capability of our approach in dealing with finer-grained faults, and the overhead of the instrumented filesystem.

4.4.1 Performance of the Classifier

The performance of our classifier is evaluated using 5-fold cross validation in $precision$, $recall$, and *F-Measure*

Table 4.2: Faults Simulation

Fault Index	How to Simulate
1	Change the port number used by web application server for database connection
2	Reduce the maximum size of connection pool of web application server
3	Reduce the amount of memory that web application server can access, so the web application fails when loaded
4	Use database administrator account to lock one of the tables
5	Reduce the number of connections permitted in *postgresql.conf*, and restart Postgresql
6	Use database administrator account to delete one of the tables
7	Change the port used for LDAP connection specified in *pg_hba.conf*, and restart Postgresql
8	Change the encryption method of LDAP to default
9	Stop the LDAP server

$$precision = \frac{tp}{tp + fp} \tag{4.1}$$

$$recall = \frac{tp}{tp + fn} \tag{4.2}$$

$$F - Measure = \frac{2 \times precision \times recall}{precision + recall} \tag{4.3}$$

where tp denotes true positive (i.e.,"correct classification", where the result of classification is the same with the type of injected failure), fp denotes false positive, and fn denotes false negative.

Figure 4.7 shows the *precision* and *recall* that our classifier obtains with a given maximum length of call sequence (i.e., $lmax = 25$). The curves reveal that the performance of the classifier increases with the support threshold (i.e., min_sup) changing from 4 to 12 and precision peaks when $min\_sup = 12$, whereas it starts to decline after min_sup exceeds 12. This implies that, at least in our system, the frequently appearing call sequences are not necessarily best for distinguishing call traces with different failure causes. On the contrary, the most useful call sequences appear only at most one dozen times. Therefore, in our system, a relatively small support threshold should be used, and in a custom environment, it needs to be adjusted to help the classifier to achieve good performance.

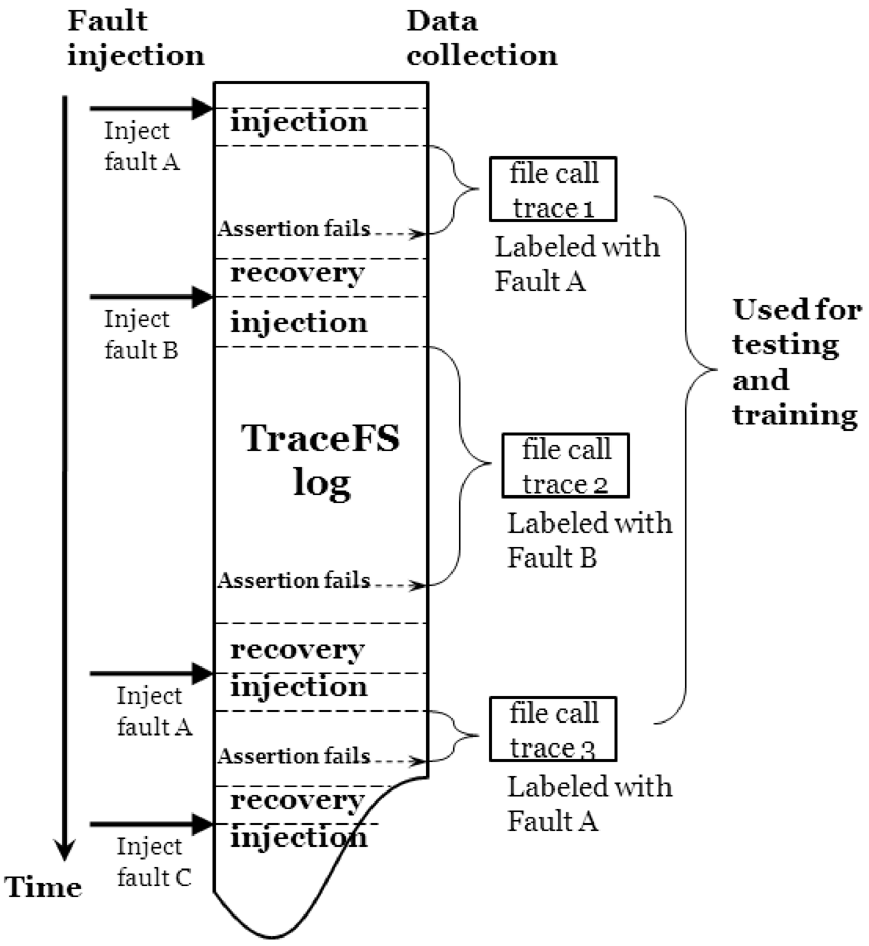


Figure 4.6: The Strategy of Failure Injection and Data Collection

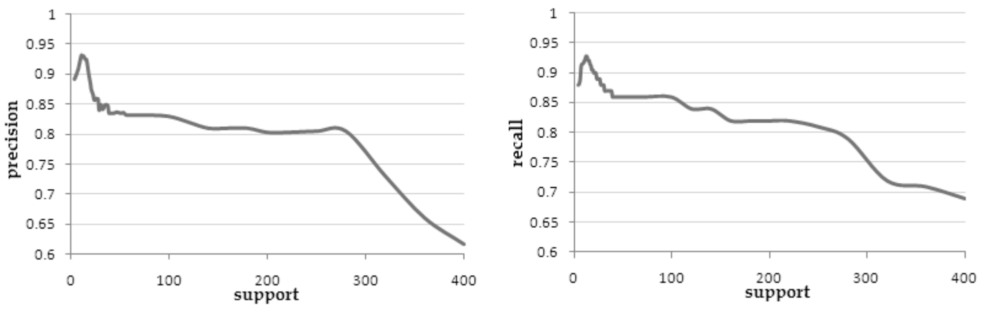


Figure 4.7: Performance of Filesystem Activity Based Approach ($lmax = 25$)

One possible reason that the $precision$ and $recall$ are not inversely related is some of the highly frequently appearing call sequences are common across all faulty states and they lower the $precision$ of our classifier when the support threshold increases.

We compare the performance of this filesystem activity based approach against

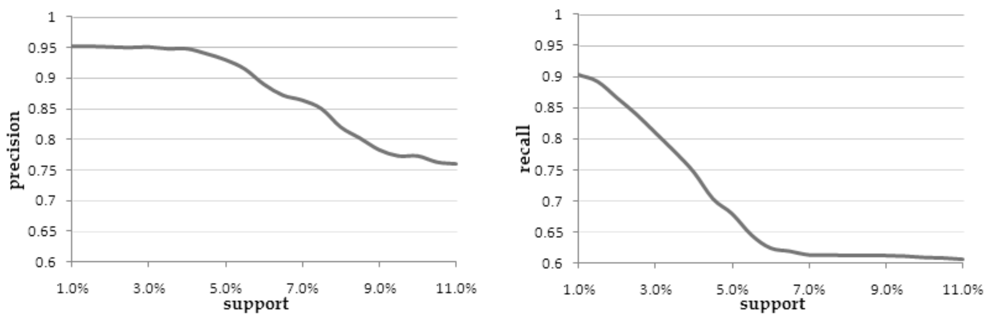


Figure 4.8: Performance of Log Based Approach

a log-based failure diagnosis approach proposed in [16], where message patterns derived from the training data set are used to aid in classifying log files. The log files it analyzes are generated and collected from our test bed along with the collection of file-related call traces. In other words, each faulty state of our test bed in this experiment is represented by one file-related call trace and one log file. The curves in Figure 4.7 and Figure 4.8 show that these two approaches perform equally well when highest performances are compared, although they use different data sources. For the log-based approaches, existing log files from all components where the diagnosis is performed are required. Those logs are generated by logging modules of the application/server software under monitoring, and vary in quality on which the effectiveness of log-based approaches is dependent. And log-based approaches cannot diagnose software having no logs. Our approach, however, analyzes only low-level filesystem events, and does not rely on logging from the software components..

In our experiment, when the highest performance was achieved, our pattern discovery program spends 260 seconds on the mining. It processed 8.3 MB preprocessed trace data and found 5190 patterns for the abnormal states. Building the Naive Bayes classifier takes only a few seconds.

The maximum length of frequent call subsequence (i.e., $lmax$), as described in Section 3.3, is another manually selected parameter. Figure 4.9 shows 3 sets of *F-Measure* values the classifier obtained with 3 given support thresholds and an $lmax$ increasing from 3 to 60. The curves again reveal that the frequently appearing call subsequences (i.e., subsequences that have high support) are not as good as those

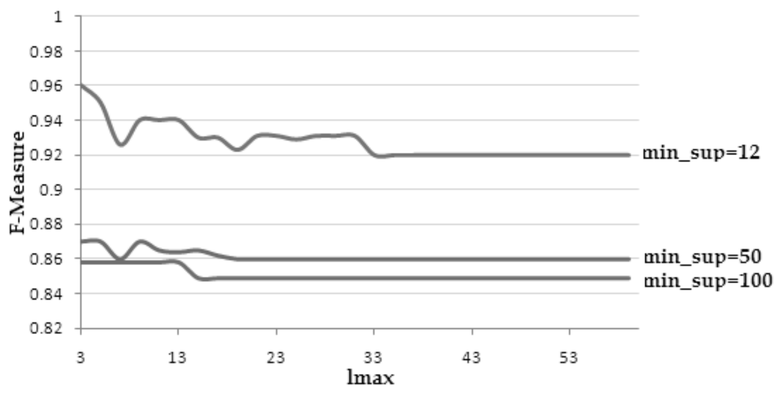


Figure 4.9: Maximum Length of Call Subsequences (*lmax*) and *F-Measure*

that appear a dozen times. In general, the performance of our classifier fluctuates as $lmax$ increases from 3 to 31 ($min\_sup = 12$), from 3 to 17 ($min\_sup = 50$), and from 3 to 13 ($min\_sup = 100$). Then the performance remains stable as $lmax$ keeps increasing. This implies that long call subsequences (e.g., 30 calls or more) are not useful in classifying call traces no matter how frequently they appear. In summary, shorter sequences at lower support do better in classification.

We can use the classifier built upon previous faulty call traces to analyze any incoming traces. But once a failure is noticed, how many file-related operations to consider or "look backward" in time for diagnosis remains an issue. An appropriate number of operations helps to obtain good classification accuracy. Intuitively, the number should be neither too small (maybe we would miss important call sequences) nor too big (the trace would contain too much irrelevant data). To investigate the appropriate number of operations to look backward, we examine the performance of our classifier by changing the number of operations for call traces involved in testing (the traces involved in classifier training are unchanged). Figure 4.10 illustrates the effectiveness of our classifier in classifying traces of varying number of operations to look backward (given that $lmax$ = 10 and min_sup = 12). The results suggest that 1000 to 1300 are good choices for the number of operations to look backward, which roughly corresponds to 30-35 seconds in time when our server receives 600 web page requests per hour.

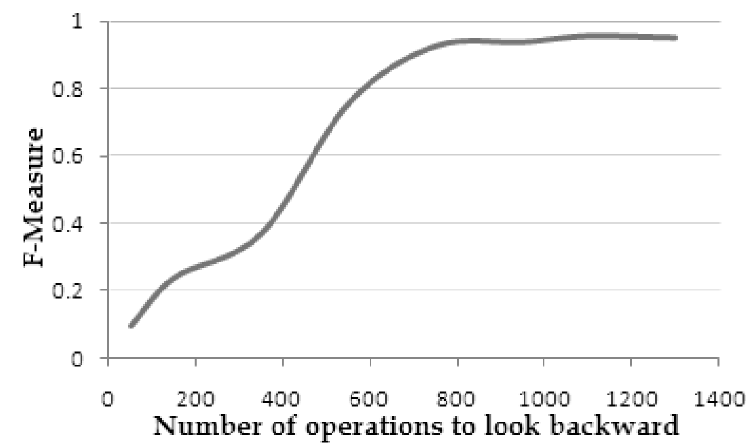


Figure 4.10: Number of Operations to Look Backward

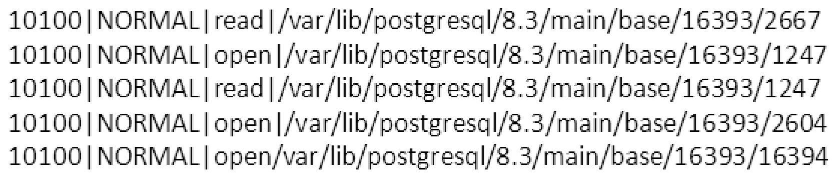

```
10100|NORMAL|read|/var/lib/postgresql/8.3/main/base/16393/2667
10100|NORMAL|open|/var/lib/postgresql/8.3/main/base/16393/1247
10100|NORMAL|read|/var/lib/postgresql/8.3/main/base/16393/1247
10100|NORMAL|open|/var/lib/postgresql/8.3/main/base/16393/2604
10100|NORMAL|open/var/lib/postgresql/8.3/main/base/16393/16394
```

Figure 4.11: A Frequent Subsequence from a Faulty Call Trace

4.4.2 Providing Finer-grained Failure Causes

Our approach uses the file-related calls logged by the instrumented filesystem to determine the type of fault, for example, database table locked. For the maintenance personnel, they may find a clue from the call traces for performing corrective actions. For example, the frequent call subsequence in Figure 4.11 indicates the monitored software touches a table or tables in the database whose OID (Object Identifier) is 16393. And the table name(s) can be determined if the OID creation mechanism and data storage mapping mechanism of Postgresql is known. So, if this frequent call subsequence happens to appear in a call sequence collected from a faulty state, the table and database touched by the monitored software could be one of the places to start investigation.

For maintenance purposes, finer-grained diagnostic results are more helpful (e.g., the specific locked table being localized if the "database table lock" fault has been determined). So we look into the call traces for two injected faults "database table locked" and "database table missing" to investigate how effective our approach is when it is used to localize faults. As Section 4.3 describes, only one

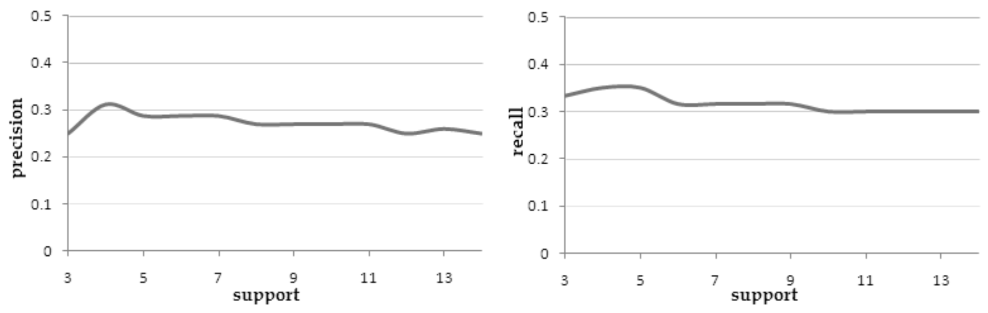


Figure 4.12: Performance of Classifier in Providing Finer-grained Failure Causes

of three tables is locked/dropped when "database table locked" or "database table missing" is injected. We label the collected call traces with failure causes plus the names of locked/dropped tables, using the strategies described in Chapter 3 and 5-fold cross validation to build and evaluate the classifier.

As Figure 4.12 shows, the logged calls cannot effectively suggest finer-grained failure causes. We find some files strongly correlated with a failure are opened and cached long before the failure is noticed (can be longer than 10 minutes), and thus calls targeted to those files do not appear in the call sequence close to the time of the failure report.

4.4.3 Overhead of the Instrumented Filesystem

The overhead of our instrumented filesystem is tested on a computer with a 2.4 GHz CPU, 1 GB memory, and one 20 GB hard disk running Centos 5.5 and Linux kernel version 2.6.18.

In the test bed, our instrumented filesystem produces an average of 5,047 KB logs (before preprocessing) per 100 page requests. We firstly estimate the time overhead by running two extremes: a CPU intensive program which involves almost no file I/O, and 4 I/O intensive shell scripts which contain 100 *tar*, 200 *cp*, 400 *rm*, and 400 "*mkdir+rmdir*" commands respectively. They are run for 10 times and we use the average run time of each program/script to calculate the slow-down. The results, as shown in Table 4.3, reveal that the filesystem instrumentation hardly slows down the running of the CPU intensive program (less than 0.01%), whereas the I/O intensive shell scripts are slowed down significantly (between 50% and

Table 4.3: Time and Space Overhead

Time Overhead		
I/O Intensive Shell Scripts	*tar*	79.1%
	cp	59.4%
	rm	50.1%
	mkdir+rmdir	53.2%
CPU Intensive Program		0.01%
Duke's Bank Application		40.1%

Space Overhead
An average of 5,047 KB logs per 100 web page requests

80%).

To determine the time overhead for a web-based application, we conducted an experiment by playing back 500 Sahi scripts on our test bed with and without the FUSE module being loaded, and used the lengths of running time to calculate how much time overhead the instrumented filesystem caused. The result shows that the slow-down of our system (i.e., 40.1%) is between the I/O intensive and CPU intensive extremes. Since part of the time overhead is due to collecting traces, the performance of web services will be less deteriorated if the traces are written to a separate hard drive.

4.5 Limitations and Threats to Validity

Some limitations of this work are:

1. Our approach can only diagnose recurrent problems, because it relies on a classifier built upon file-related call traces collected at faulty states. Also, this approach is not effective in providing finer-grained failure causes.

2. The overhead due to filesystem instrumentation slows down file operations.

3. This approach is unable to distinguish failures that have no manifestation or very similar manifestations in file-related calls. Performance-related problems could also be hard to detect, as a software system would still be functional but with degraded performance.

4. In practice, when a failure is noticed, it is difficult to decide how many past file-related operations to consider for training purposes, as a failure may not be noticed immediately after its occurrence.

5. The time of file-related calls being made, and data transmitted between files and virtual memory are not used by this approach.

6. The strategy of dealing with multiple failures and failure interactions is not studied.

7. This approach can only be migrated to Linux/Unix systems which FUSE are compatible with. The strategy of collecting and analyzing filesystem activities in other systems is not investigated.

8. The suggested support threshold and number of look-backward operations are based on our test bed. A further study is needed to find the optimum values systematically in a different environment.

Threats to the validity of the experimental results are as follows:

1. An enterprise system is usually large-scale, while our experiment is performed on only one machine, although theoretically this approach can be extended to an environment that comprises multiple machines. Similarly, the size and complexity of the software system involved in the experiment is also one of the threats, because most enterprise systems are larger and more complicated than our the software system in our test bed.

2. Duke's Bank is not a real banking application. Also, the rate of user requests we generated in the experiment mimics the incoming traffic of a web server in Princeton rather than business banking systems.

3. Failures are artificial and manually injected, and thus may not represent real, spontaneous failures. For example, a locked table in the database may be due to problems with the source code rather than an operator using the command line to lock the table by mistake.

4. We allow only one fault to be injected at a time, so the effectiveness of this approach is not studied when multiple failures occur, and the interactions between faults are not studied either.

4.6 Summary

The experimental results obtained in the evaluation, in particular the highest overall performance of our approach (93% in *precision* and 92% in *recall*) in determining the fault types, and the overhead of our instrumented filesystem in time and space, confirm that file-related call traces can be used to diagnose the observable functional failures. By experimenting on a system that provides web services, we show that our approach can be successful in diagnosing systems that have multiple processes and filesystems.

Chapter 5

Related Work

To frame our work in its proper context, the related work is reviewed in this chapter.

The most closely related work to our approach is Ding et al. [10]. They collect the environmental variables, user information, signals, and system calls from a system's normal execution at run time, and build an "application signature bank" which represents normal behaviors of the software. The "signature bank" assists the diagnosis in a dictionary-like way. When a failure occurs, the human operators start the diagnosis process: a classifier tool would search the signature bank to find the most similar model to the current state and therefore determine the type of the failure. While Ding et al. use a relatively more intrusive technique to reverse engineer the software behaviors with all system call traces and arguments, our approach focuses on how to model the software behaviors using the information contained in and returned by file-related calls.

The study conducted by Dickinson et al. [9] proposed the idea of clustering event traces/execution profiles to assist software diagnosis by clustering traces/profiles. One of their conclusions is that a significant number of execution profiles with failures are distributed in small clusters. The diagnosis that Dickinson et al. perform is a process of similarity estimation between traces and no classifiers are involved. While Dickinson et al. do not need to label the traces, they cannot determine failure causes as our work does.

Event traces are also the key information for the root cause analysis in the approach proposed by Yuan et al. [41], while the difference is that they record all system call traces for the Windows operating system, and SVM (Support Vector

Machine) is employed to classify call sequences.

Rather than determining the failure causes, Chen et al. [7] use real-time requests to pinpoint the misbehaved software components. They keep track of software components involved in all requests and the corresponding return states, using them to find correlations between software components and failures. Compared with our approach working at the file level, the diagnosis approach proposed by Chen et al. works at the software component level.

Besides the work reviewed above, there also exist failure diagnosis approaches that use other sources of information. Log files contain implications of system behaviors and representations of failure symptoms, and thus are one of the major sources used for problem diagnosis. Both Reidemeister et al. [35, 36] and Mariani et al. [27] are similar to the work in this thesis in the architectural design. With patterns mined from logs, they transform log files into feature vectors, from which classifiers are built to distinguish failures with different causes. While their approaches rely on log files and clustering methods, our approach logs file-related calls with the tool we developed and uses a Naive Bayes classifier.

In this area, post-error reports (e.g., [4]), problem tickets (e.g., [15]), incident reports (e.g., [30]), call-stacks (e.g., [5]), and system configuration files (e.g., [40]) are also used for problem diagnosis. Our approach is complementary and considers filesystem events, which are not sensitive to the ambiguity that may be found in information sources such as problem tickets or bug reports.

Chapter 6

Conclusion and Future Work

In this thesis, we propose an automatic approach to diagnose recurrent software failures based on the assumption that filesystems are involved in some way when software misbehaves. Unlike some autonomic systems, this approach is unobtrusive, because it does not re-architect the system. Our approach does not require source code access, recompilation of software or the operating system, or any supporting information regarding the monitored software. Also, it is more generic, and does not assume the system is based on any particular types of components.

By capturing and analyzing file-related calls through an instrumented filesystem, our approach suggests failure causes, which can assist maintenance personnel in performing quick and simple incident fixes before the potential deployment of a permanent solution. We evaluated this approach in a web service system, where faults are injected.

The results show that our approach performs well in determining failure types while the I/O intensive processes are slowed down (80% at worst) due to the filesystem instrumentation. Nevertheless, it is not effective in providing finer-grained failure causes due to caching. Moreover, finding clues for corrective maintenance from the call traces requires background knowledge of the system and monitored software applications.

This approach collects and analyzes the file-related call names, targeted files, PIDs, and return values. With the instrumented filesystem, we can collect more information such as call parameters other than file names, and data transmitted between files and buffers in virtual memory. Also, failure interaction and the strategy of dealing with multiple failures will be investigated. This approach only deals with faults that lead to an observable functional failure rather than a performance degradation. Thus the analysis of time that file-related calls are made and returned will also be part of the future work.

This approach currently works on one machine, and it will be extended in an environment that comprises more machines, where the approach will be evaluated with a greater variety of injected failures. Also, a more systematic way of choosing parameters (e.g., the support threshold) of our approach in a custom environment will be studied.

Bibliography

[1] Web automation and test tool. http://sourceforge.net/projects/sahi/.

[2] E.N. Adams. Optimizing preventive service of software products. *IBM Journal of Research and Development*, 28(1):2–14, 1984.

[3] L.D. Baker and A.K. McCallum. Distributional clustering of words for text classification. In *Proceedings of the 21st Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 96–103. ACM, 1998.

[4] K. Bartz, J.W. Stokes, J.C. Platt, R. Kivett, D. Grant, S. Calinoiu, and G. Loihle. Finding similar failures using callstack similarity. In *Proceedings of the 3rd Conference on Tackling Computer Systems Problems with Machine Learning Techniques*, pages 1–6. USENIX Association, 2008.

[5] M. Brodie, S. Ma, L. Rachevsky, and J. Champlin. Automated problem determination using call-stack matching. *Journal of Network and Systems Management*, 13(2):219–237, 2005.

[6] B.M. Cantrill, M.W. Shapiro, and A.H. Leventhal. Dynamic instrumentation of production systems. In *Proceedings of the 2004 USENIX Annual Technical Conference*, page 2. USENIX Association, 2004.

[7] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic systems. In *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604. IEEE Computer Society, 2002.

[8] R. Chillarege, S. Biyani, and J. Rosenthal. Measurement of failure rate in widely distributed software. In *Proceedings of the 25th International Symposium on Fault-Tolerant Computing*, pages 4–24. IEEE Computer Society, 1995.

[9] W. Dickinson, D. Leon, and A. Podgurski. Finding failures by cluster analysis of execution profiles. In *Proceedings of the 23rd International Conference on Software Engineering*, pages 339–348. IEEE Computer Society, 2001.

[10] X. Ding, H. Huang, Y. Ruan, A. Shaikh, and X. Zhang. Automatic software fault diagnosis by exploiting application signatures. In *Proceedings of the 22nd Conference on Large Installation System Administration Conference*, pages 23–39. USENIX Association, 2008.

[11] H. Eto and T. Dohi. Analysis of a service degradation model with preventive rejuvenation. *Service Availability*, pages 17–29, 2006.

[12] S.L. Graham, P.B. Kessler, and M.K. Mckusick. Gprof: A call graph execution profiler. *ACM Sigplan Notices*, 17(6):120–126, 1982.

[13] P. Grubb and A.A. Takang. *Software maintenance: concepts and practice*. World Scientific Publishing Co. Pte. Ltd., 2003.

[14] M. Haardt and M. Coleman. ptrace (2). *Linux Programmer's Manual*, 1999.

[15] H. Huang, R. Jennings III, Y. Ruan, R. Sahoo, S. Sahu, and A. Shaikh. PDA: a tool for automated problem determination. In *Proceedings of the 21st Conference on Large Installation System Administration Conference*, pages 1–14. USENIX Association, 2007.

[16] L. Huang, X. Ke, K. Wong, and S. Mankovskii. Symptom-based problem determination using log data abstraction. In *Proceedings of the 2010 Conference of the Center for Advanced Studies on Collaborative Research*, pages 313–326. ACM, 2010.

[17] B. Jacob, P. Larson, B. Leitao, and S. Silva. *SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems*. IBM Red Paper, 2009.

[18] S. Jiang and R. Kumar. Failure diagnosis of discrete-event systems with linear-time temporal logic specifications. *IEEE Transactions on Automatic Control*, 49(6):934–945, 2004.

[19] M.T. Johnes. Linux introspection and SystemTap. *http:// www.ibm.com /developerworks/linux/library/l-systemtap/?ca=drs-*, 2009.

[20] I. Lee and R.K. Iyer. Software dependability in the Tandem GUARDIAN system. *IEEE Transactions on Software Engineering*, 21(5):455–467, 1995.

[21] I. Lee and R.K. Iyer. Diagnosing rediscovered software problems using symptoms. *IEEE Transactions on Software Engineering*, 26(2):113–127, 2000.

[22] J. Levon and P. Elie. Oprofile: A system profiler for linux. *http://oprofile. sourceforge. net*, 2005.

[23] J. Li, P. Martin, W. Powley, K. Wilson, and C. Craddock. A sensor-based approach to symptom recognition for autonomic systems. In *Proceedings of the 5th International Conference on Autonomic and Autonomous Systems*, pages 45–50. IEEE, 2009.

[24] J. Li, T. Stålhane, J.M.W. Kristiansen, and R. Conradi. Cost drivers of software corrective maintenance: An empirical study in two companies. In *Proceedings of the 26th IEEE International Conference on Software Maintenance*, pages 1–8. IEEE, 2010.

[25] R. Love. Kernel korner: Intro to inotify. *Linux Journal*, 2005(139):8, 2005.

[26] R. Love. *Linux System Programming*. O'Reilly Media, Inc., 2007.

[27] L. Mariani and F. Pastore. Automated identification of failure causes in system logs. In *Proceedings of the 19th International Symposium on Software Reliability Engineering*, pages 117–126. IEEE, 2008.

[28] A. McCallum and K. Nigam. A comparison of event models for naive bayes text classification. In *Proceedings of the 1998 AAAI Workshop on Learning for Text Categorization*, volume 752, pages 41–48. AAAI, 1998.

[29] J.A. McDermid. *Software engineer's reference book*. Butterworth-Heinemann, 1991.

[30] S. Nadi, R. Holt, I. Davis, and S. Mankovskii. DRACA: decision support for root cause analysis and change impact analysis for CMDBs. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 1–11. ACM, 2009.

[31] J. Nickurak. *Slimming virtual machines based on filesystem profile data*. Master's thesis, University of Alberta, 2010.

[32] Office of Information Technology. Server traffic statistics. http://mrtg.net.princeton.edu/fcgi/mrtg-rrd/gigagate5-10-39.html.

[33] S. Pertet and P. Narasimhan. Causes of failure in web applications. *Parallel Data Laboratory, Carnegie Mellon University, CMU-PDL-05-109*, 2005.

[34] A. Razavi and K. Kontogiannis. Pattern and policy driven log analysis for software monitoring. In *Proceedings of the 32nd IEEE Computer Software and Applications Conference*, pages 108–111. IEEE, 2008.

[35] T. Reidemeister, M.A. Munawar, M. Jiang, and P.A.S. Ward. Diagnosis of recurrent faults using log files. In *Proceedings of the 2009 Conference of the Center for Advanced Studies on Collaborative Research*, pages 12–23. ACM, 2009.

[36] T. Reidemeister, M.A. Munawar, and P.A.S. Ward. Identifying symptoms of recurrent faults in log files of distributed information systems. In *Proceedings of the 12th IEEE/IFIP Network Operations and Management Symposium*, pages 187–194, 2010.

[37] F. Salfner and S. Tschirpke. Error log processing for accurate failure prediction. In *Proceedings of the 1st USENIX Conference on Analysis of System Logs*, page 4. USENIX Association, 2008.

[38] M. Szeredi. FUSE: Filesystem in Userspace. 2005. *http://fuse. sourceforge. net*.

[39] B. Topol, D. Ogle, D. Pierson, J. Thoensen, J. Sweitzer, M. Chow, M.A. Hoffmann, P. Durham, R. Telford, S. Sheth, and T. Studwell. Automating problem determination: A first step toward self-healing computing systems. *IBM white paper*, 2003.

[40] M. Wang, X. Shi, and K. Wong. Capturing Expert Knowledge for Automated Configuration Fault Diagnosis. In *Proceedings of the 19th IEEE International Conference on Program Comprehension*, pages 205–208. IEEE Computer Society, 2011.

[41] 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 *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems 2006*, pages 375–388. ACM, 2006.