Multi-layered System Robustness Testing Strategy Based on Abnormal Parameter

Lin Xiang¹,², Zhan Zhang, Decheng Zuo, Xiaozong Yang
¹College of Information Engineering, China Jiliang University, Hangzhou, Zhejiang, China
²Department of Computer Science & Technology, Harbin Institute of Technology, Harbin, China
Email: xianglin.cjlu@gmail.com

Abstract—A multi-layered fault injection mode is explored and a multi-layered system robustness testing strategy based on abnormal parameter is put forward. Fault injection methods for three layers (API, DPI and system calls based on workloads) of Linux operation system are designed. And an integrated fault injection platform to multi-layered fault injection for testing robustness of operating system is implemented. Using the multi-layered system robustness testing strategy based on abnormal parameter under the platform, various layers of system robustness evaluation are achieved. Finally, the results under different layers of fault injection are compared and analyzed in order to evaluate the robustness of target system.

Index Terms — abnormal parameter, fault-injection, multi-layered robustness testing

I. INTRODUCTION

System robustness is a measure of the ability to maintain a normal condition in the environment of abnormal input or pressure system [1], reflecting the system’s detection and processing capabilities for failures, external events and system maintenance events. In critical industrial applications, system delays and failures may cause incalculable damage, so under the premise to ensure the system’s high-performance transaction processing capabilities, the system’s robustness is essential.

Robustness test is a more emerging area of research, the study of this topic focused on some foreign universities. Professor Miller of University of Wisconsin proposed by means of fault injection to evaluate the system robustness earlier. Professor Siewiorek of Carnegie Mellon University proposed the definition of robustness testing benchmark to test system handling error input or unexpected input in different levels, distinguish robustness testing and traditional performance testing for the first time; Professor Koopman described a portable testing benchmark to test the robustness of UNIX operating systems, and proposed an approach to classify the testing results and compare between different operating systems. Mukherjee and Siewiorek proposed a hierarchical approach to building robust testing benchmarks [2]. A workload-aware reliability evaluation model is introduced in [3].

In recent years, software testing technology has developed rapidly, a large number of software testing tools are developed, the tools for system robustness testing are more and more, and they can automatically detect software failures, and find a large number of software failures and security risks in some large commercial software and open source software testing. A failure detection scheme in order to improve the fault tolerance of the service is proposed in [4]. EXPLODE found a number of serious problems in some common file storage systems; MC found nearly 500 faults and more than 100 security vulnerabilities in Linux, OpenBSD and other software; SDV found 65 faults, including 12 serious software failure in the test of drivers of Windows operating system, etc.[5].

Although there have been a lot of study results of fault injection, and many fault-injection tools have been used, the research of fault injection techniques for the unity of its own formal model is still lacking [6]. Most researches have focused on two aspects:

1) Improvement and innovation to the concrete realization of fault injection, including the methods injecting faults to different abstraction levels, simulating the real fault impact as accurately as possible, improving the trigger accuracy of fault injection, reducing overhead and improving accuracy of testing results, etc.;
2) Fault injector designed for certain specific projects, which aims to get some interesting characteristics or indicators through fault injection to a specific system. The research limitations led to these specific fault injection techniques simple rather than system application.

All in all, the traditional fault injection tools are often designed to assess a target system, limited to a specific architecture, and fault injection method simple, such as early fault injection tools FIAT, FERRARI, etc.; recent years, the study of fault injection began to develop to the
multiplex direction, many new fault-injection tools integrate a variety of fault-injection techniques and can be transplanted between multiple operating systems, such as integrated fault injection tool GOOFI \[7\], which combines four fault injection means, including software fault injection, scan chain implemented fault injection, through the on-chip debug interface fault injection and standard interface fault injection.

Table 1 compared the characteristics of the classic robustness testing tools:

<table>
<thead>
<tr>
<th>Tool</th>
<th>Portability</th>
<th>Coverage</th>
<th>Scalability</th>
<th>Level of detail</th>
<th>Repeatability</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crashme</td>
<td>Good</td>
<td>Low</td>
<td>Bad</td>
<td>Low</td>
<td>Low</td>
</tr>
<tr>
<td>FIAT</td>
<td>Bad</td>
<td>High</td>
<td>Bad</td>
<td>High</td>
<td>Low</td>
</tr>
<tr>
<td>FUZZ</td>
<td>Good</td>
<td>Low</td>
<td>Good</td>
<td>Low</td>
<td>High</td>
</tr>
<tr>
<td>Ballista</td>
<td>Good</td>
<td>High</td>
<td>Good</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>FINE</td>
<td>Good</td>
<td>High</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

We can see Ballista has the advantages of portability, high test coverage, scalability, high level of detail, and good repeatability. And Ballista has high degree of automation and tests software without source code. Nevertheless, Ballista only calls API functions in user mode frequently, causing the user-level failures, can’t trigger kernel-level failures, and not consider the workload running on the system, the robustness testing of the system is not entire.

Utilizing thoughts of injecting abnormal parameters to the interface functions, integrating the classic abnormal parameter injection methods and a variety of fault injection strategies of dynamic and static, user mode and kernel mode, this paper proposed a multi-layered system robustness testing strategy based on abnormal parameters for the architecture of Linux operating system, using a consistent fault injection method to different levels of system, and tests system robustness of user mode and kernel mode programming interface and dynamic load of the system.

II. MULTI-LAYERED FAULT INJECTION STRATEGIES BASED ON ABNORMAL PARAMETERS

According to the process system provide service mode to user process, system can be divided into three levels: user programming interface layer, the system call layer and the core function layer. Thus, to provide services, the system robustness failures can be attributed to these three levels, a certain level of testing which can test system robustness from a certain point, but it is incomplete. The multi-layered system robustness testing strategy based on abnormal parameters integrates the classic method of abnormal parameters and the dynamic and static, user mode and kernel mode for a variety of fault injection strategies, uses the consistent fault injection method to different levels of system, and tests system robustness of user mode and kernel mode programming interface and dynamic load of the system.

A. Fault Injection Strategy of User Mode and Kernel Mode

1. Fault parameters generated

The fault injection objects to test API robustness are API function calls in line with the POSIX standard \[8\]. The fault injection objects of DPI robustness testing are the commonly used DPI function calls compatible with some target systems \[9\], including C library functions of the core API, memory management functions, etc.

Fault parameters generating process:

1) fault line = grep ($ function, list) //match the function name, get the function information of the list;
2) if (no matching information) {the function is not the injecting object};
3) @params = split(fault_line) //get the type of fault parameters of the function @ params;
4) ./paramGen @params //generate the fault parameters set of use cases corresponding to parameter type;
5) Sequentially select a set of fault parameters fault_params;
6) ./inject function fault_params //perform fault injection.

2. Fault injection method

Robustness testing of API layer through abnormal parameters fault injection to a single API function calls, the injection process is:

1) initialize use cases of fault parameter, perform the corresponding functions generating the fault parameters, and return the address of the abnormal parameters;
2) call the test functions after the first step generating the abnormal parameters, record call results when the function return value is not empty or structure type;
1. Fault parameters generated above of a certain type of system calls of the four categories value, the maximum integer value \[10\].

pointer), an empty cache pointer, the smallest integer file pointer, illegal buffer pointer (excluding empty cache pointer (not including the empty file pointer), an empty cache.

abnormal parameters caused the highest probability of ballista project, statistics show that six categories of statistic results of the system robustness failure causes of fault.

common type of fault in the actual operation: bit flip abnormal parameters, also takes into account the most fault parameters generation in addition to the method of programming interface (DPI) compatible in target functions, inject fault parameters to the driver

Fault injection is implemented through system calls written by ourselves. Specific process is:

1) match the kernel module file ModuleSyscall.c for the tested Kernel function (the module implements the system call SYS_kernelfunc to call the objective function);
2) Compile the kernel module to get ModuleSyscall.o;
3) Insert kernel module;
4) Hook system calls;
5) Initialize the fault parameters of the objective function;
6) Pass the initialized fault parameters to the kernel space by executing the system call, test the kernel function, and return the function returned values to the user state;
7) Finally, clean up the resources the initialization applied;
8) Unload kernel modules.

B. Fault Injection Strategy of Application Load Running

According to system call’s functions, the fault injections intercepted system calls are divided into four categories:

1) The system calls for file operations
2) The system calls for process control
3) The system calls for system signal
4) The system calls for disk read and write

Fault injection object on the application load is a set of a certain type of system calls of the four categories above.

1. Fault parameters generated

Because it is the runtime fault injection, this part of the fault parameters generation in addition to the method of abnormal parameters, also takes into account the most common type of fault in the actual operation: bit flip fault.

The selection of abnormal parameters refer to the statistic results of the system robustness failure causes of ballista project, statistics show that six categories of abnormal parameters caused the highest probability of system robustness failure, they are as follows: illegal file pointer (not including the empty file pointer), an empty file pointer, illegal buffer pointer (excluding empty cache pointer), an empty cache pointer, the smallest integer value, the maximum integer value \([10]\).

Bit flip faults can be classified according to the bits damaged: single bit flip, two bits (compensation) flip and the whole word (32 bits) flip.

By flipping can be divided into: inversion, set 0 and set 1.

Fault parameters can choose the Cartesian product of the above types, also can randomly generate 32-bit mask, which and the original parameter values.

2. Runtime fault injection method

The fault injection to the system call called in the load is carried out in the load running, in Linux operating environment, to achieve fault injection to the running process, we need control the running process, read and write the process context. This paper uses system call ptrace to achieve the target process tracking, set traps, read the context, fault injection and result monitoring. Ptrace can intercept and modify the user level system calls, using ptrace system call, a process can dynamically read and write memory image and register values of another process, including the code segment, data segment and stack segment; and access and modify the general registers and special registers.

Runtime fault injection process is:

1) debugger forks a child process \(\text{pid} = \text{fork}()\);
2) The child process calls the system call PTRACE_TRACEME, the result is: the kernel set a trace bit in the child process table, requiring the parent process trace into debug state;
3) The child process executes the tracked program, i.e. the work load. Kernel executes the system call exec(workload), when execution to the last, due to the track bit is set, sends a soft interrupt signal SIGTRAP to the child process;
4) When the system call exec(workload) returns, check the soft interrupt signal. If the tracking bit is set in the process table, the child process wake up the parent process slept in the system call waitpid, into a tracking state similar to sleep state, and complete the context switch \([11]\).

Meanwhile, the parent process executes the system call waitpid(), wait for the wake-up by the child process. When the child process wake up the parent process, the parent process returns from waitpid(), begin to execute the fault injection inject_faults(), and then monitor the results of fault injection.

Fault injection to workload is achieved through the ptrace system call, the format as follows:

\(\text{int ptrace (int request, int pid, int addr, int data);}\);

The parameter request is an enumeration variable, its value determines the function of system calls, as shown in Table 2:

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Feature</th>
</tr>
</thead>
<tbody>
<tr>
<td>PTRACE_TRACEME</td>
<td>Process requires the parent process to track, so that the current process into debug state</td>
</tr>
<tr>
<td>PTRACE_ATTACH</td>
<td>Track user process</td>
</tr>
<tr>
<td>PTRACE_DETACH</td>
<td>Stop debugging the process</td>
</tr>
<tr>
<td>PTRACE_PEEKTEXT</td>
<td>Read a word of the user instruction space</td>
</tr>
<tr>
<td>PTRACE_PEEKDATA</td>
<td>Read a word of the user data space</td>
</tr>
<tr>
<td>PTRACE_PEEKUSR</td>
<td>Read a word of the user process data area</td>
</tr>
<tr>
<td>PTRACE_POKETEXT</td>
<td>Write a word of the user instruction space</td>
</tr>
<tr>
<td>PTRACE_POKEDATA</td>
<td>Write a word of the user data space</td>
</tr>
</tbody>
</table>

© 2013 ACADEMY PUBLISHER
process. system's handling of SIGALRM, interrupt the child SIGALRM, then sig_alrm() will take over the operating (PTRACE_TRACEME).

transfers data between two processes, and lock the track tracked, then track it using a global data structure, to prevent other tracking process rewriting.

3. Runtime fault injection trigger mechanism

Fault injection on the application load is different from fault injection on single function call, fault injection time need to be chosen, because the system call injected is not always the first system call of the application load, also can choose to skip the initialization phase of the process, injecting faults when the load provides services. Therefore, under certain circumstances, trigger the fault injection through the mode of fault trigger. This strategy offers two fault trigger modes: single-step trigger and time trigger.

Single-step trigger that is trigger fault injection after the process executes the specified number of steps is implemented by the function run_step(). For the specified number of instructions, each instruction is executed in the function and keep the signal lastsig before the step to continue processing.

Time Trigger that is after the program execution time exceeds the specified time (unit: microseconds) trigger fault injection. Implement by the function run_time() to complete the following tasks:

1) Register a signal SIGALRM, the signal processing function is sig_alrm(), sigalarm() will send a signal SIGTRAP to the child process wordload to interrupt the child process.

2) Get the current machine time, resume operation of the sub-process interruption caused by ptrace (PTRACE_TRACEME).

3) Generate timer interrupt, the interrupt signal is SIGALRM, then sig_alrm() will take over the operating system's handling of SIGALRM, interrupt the child process.

III. INTEGRATED FAULT INJECTION PLATFORM BASED ON MULTI-LAYERED FAULT INJECTION

For Fault injection to API function calls, DPI calls and the application process, a single fault injection measure is not enough, we need to combine multi-level fault injection and multi-angle robustness testing. Based on multi-level fault injection strategies, the paper designs and implements an integrated fault-injection platform to complete evaluation of different levels of system robustness.

It can be seen from Figure 2, at different levels of fault injection and results monitoring collection requires different fault injector and results monitor, fault set, fault trigger and injection strategies are different, but robust testing tools at home and abroad are basically to achieve a particular aspect of test, their architectures are not compatible, cannot construct an integrated test environment. The integrated fault-injection platform designed in this paper, whose advantage is most of the classical fault injection tools and robust testing tools can achieve compatibility in this platform through small changes, and jointly build an integrated robust test environment. In the platform, the development of robust testing tools implements the incremental development mode.

A. Control Module

Fault injection control module determines the fault injection object, generates fault parameters (fault load), and determines the application-oriented workloads.

Fault injection object: i.e. the target need fault injection. For example, fault injection to API function calls, an API function call is a fault injection object; in memory fault injection, memset1 is a fault injection object. The fault injection objects in this paper have three categories: API functions, DPI functions and the system calls in the application process.

Fault parameter generation is to generate the required parameters for the fault injection object. For example, for fault injection to the API function access, the fault parameters are a set of parameters {S1, CHAR, EMPTY, R_OK}; for the fault injection to memory fault memset1, the parameters are the memory address injected necessarily and parameter 1.

Load library is the load needed to use in fault injection, the control module designates the fault injection to the target system in which load.

Results collection module recycles the results of fault injection collected by the target system monitoring module and analyzes.

B. Fault Injection Module

Fault injection module selects fault injection method according to the fault injection object, generates the workload the control module selected, injects faults according to the fault parameters, and uses different strategies to inject faults to different levels respectively. Injection method selection module: select the
appropriate injection method for the fault injection object. For example, select the ballista tools to inject abnormal parameters for all of the API functions. The relationship between objects and methods is many to one, that is, many objects may use the same injection method.

Injection method module: corresponding to different fault injection tools, injection method module selected by the injection method selection module injects faults corresponding to the parameter information in the interface information.

Load generation module: run the workload according to the type of load the control module specified.

C. Results Monitoring Module

Results monitoring module is actually the parent process executing fault injection monitors function calls or work load injected faults by the method of monitoring the child process, record the results of fault injection, and feedback to the control module for analysis. The fault injection results are divided into three categories:

- Restart: process timeout restarts after fault injection;
- Abort \[^{[12]}\]: process throws an exception signal after fault-injection, causing the process to abort;
- Pass: after fault injection, the process ends normally or returns the corresponding error code.

Fork child process first, implement fault injection, monitor the function call result is restart, abort or pass through monitoring the termination status of the child process \[^{[13]}\].

Statistical analyzing the results collected by result monitoring module, results analysis module performs the following functions:

1. Read the result documents generated by result monitoring module, count the number of fault injection for each test function;
2. Read the results files, count the number of failures in fault injection and the number of the various results (pass/abort/restart) occurrences after fault injection for each function;
3. Calculate the ratio of three outcomes (pass/abort/restart) in case of fault injection success for each function, and proportion of total failure;
4. Calculate the average failure rate of all measured functions, determine an overall robustness score.

IV. EXPERIMENTAL RESULTS AND ANALYSIS

A. Robustness Testing Results of Application Programming Interface

1. Experimental Environment

Test environment I:

- VMware OS: Redhat6.0 \(i386\); kernel: Linux2.2.14;
- Memory: 128M;
- Host OS: Windows XP;
- CPU: AMD2800 +

Test environment II:

- VMware OS: Redhat9.0 \(i386\); kernel: Linux2.4.20;
- Memory: 128M;
- Host OS: Windows XP;
- CPU: AMD2800 +

Test environment III:

- OS: RHEL5.1 \(i386\); kernel: 2.6.18;
- Inspur server NX7140D;
- CPU: Intel Xeon E5405

API package is different for different versions of the Linux kernel, so API robustness is different too for different versions of the Linux operating system. The operating systems in the three experimental environments above are based on three versions of the kernel 2.2, 2.4 and 2.6 of Linux respectively.

2. Experimental results and analysis

POSIX standard API test of Redhat 6 system (Linux 2.2.14 kernel) started from 8:00 on March 28, 2009 and ended at 3:30 p.m. on March 28, 2009 (to obtain the test results). Test 213 API functions, generate 345974 test cases, of which the effective implementation of test cases 281267, each API function implemented 1320.5 effective test cases averagely. The test results are 1241 times to restart, accounting for effective test cases of 0.4412%; 32519 times to abort, accounting for effective test cases of 11.5606%; 247507 times to normally pass, accounting for effective test cases of 87.9971%.

System API functions are divided into 15 categories, including file management, directory management, process control, process scheduling, signals, semaphores, terminal I/O, advanced I/O, standard library functions, string handling, character handling, mathematical functions, time functions, memory management, and other library functions. The function failure rate according to the classification above is as follows:

Test 211 POSIX standard APIs of Linux 2.4.20 kernel, generate 365735 test cases, of which the effective implementation of test cases 305201, each API function implemented 1446.5 effective test cases averagely. The test results are 3255 times to restart, accounting for
effective test cases of 1.0663%; 46055 times to abort, accounting for effective test cases of 15.0898%; 255891 times to normally pass, accounting for effective test cases of 83.8434%.

POSIX standard API test of Redhat 5EL system (kernel version 2.6.18) started from 18:00 on March 12, 2009 and ended at 15:32 on March 30, 2009 (to obtain the test results). Test 208 API functions, generate 370665 test cases, including the effective implementation of test cases 326301, each API function implemented 1568.75 effective test cases averagely. The test results are 5253 times to restart, accounting for effective test cases of 1.61%; 72434 times to abort, accounting for effective test cases of 22.1986%; 248614 times to normally pass, accounting for effective test cases of 76.1916%.

By comparing failure rate of different types of functions (file management, directory management, process control, process scheduling, signals, semaphores, terminal I/O, advanced I/O, string handling, character handling, math functions, time functions, memory management and other library function), as shown in Table 3, further analyze the reason of robustness failure rate increasing for different kernel versions.

<table>
<thead>
<tr>
<th>Function</th>
<th>linux2.2.14</th>
<th>linux2.4.20</th>
<th>linux2.6.18</th>
</tr>
</thead>
<tbody>
<tr>
<td>file management</td>
<td>2.3895</td>
<td>2.4949</td>
<td>5.0101</td>
</tr>
<tr>
<td>directory management</td>
<td>46.8822</td>
<td>51.3514</td>
<td>66.1193</td>
</tr>
<tr>
<td>process control</td>
<td>8.7339</td>
<td>4.7188</td>
<td>16.4803</td>
</tr>
<tr>
<td>process scheduling</td>
<td>1.2361</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>signals</td>
<td>7.1246</td>
<td>4.7901</td>
<td>7.545</td>
</tr>
<tr>
<td>semaphores</td>
<td>32.6615</td>
<td>27.1955</td>
<td>28.7383</td>
</tr>
<tr>
<td>terminal I/O</td>
<td>0.0063</td>
<td>0.0063</td>
<td>0</td>
</tr>
<tr>
<td>advanced I/O</td>
<td>9.5281</td>
<td>19.3929</td>
<td>33.2777</td>
</tr>
<tr>
<td>stllib</td>
<td>13.0155</td>
<td>9.9452</td>
<td>13.8521</td>
</tr>
<tr>
<td>string handling</td>
<td>38.9493</td>
<td>40.22</td>
<td>64.3342</td>
</tr>
<tr>
<td>character handling</td>
<td>6.7692</td>
<td>74.4615</td>
<td>81.2308</td>
</tr>
<tr>
<td>math functions</td>
<td>2.3522</td>
<td>2.3522</td>
<td>3.2777</td>
</tr>
<tr>
<td>time functions</td>
<td>21.1428</td>
<td>21.8549</td>
<td>20.5587</td>
</tr>
<tr>
<td>memory management</td>
<td>0.0063</td>
<td>0.0063</td>
<td>0</td>
</tr>
<tr>
<td>others</td>
<td>4.9091</td>
<td>4.4641</td>
<td>5.1048</td>
</tr>
</tbody>
</table>

From figure 9 can be found the increase of robustness failure probability was mainly due to the rapid increase of advanced I/O function, string handling function and character handling function. Typically in linux2.2 kernel, the robustness failure probability of fault injection to character handling function is only 6.7692%, while the failure probability of 2.4 kernel and 2.6 kernel is 74.4615% and 81.2308% respectively. The reason is mainly due to robustness failure of the high version function implementation in the wrong argument type.
memmove, memset, strcp, kmem_cache_destroy will lead to the kernel stack fault, resulting in system crash.

From the experimental results can be seen, robustness failure of DPI level is mainly due to the operation of string and memory. The mainly failure reason is caused by illegal string pointer and buffer pointer.

C. System Robustness Testing Results of Fault Load
1. Experimental Environment
OS: RHEL5.1 i386; Kernel 2.6.18; Inspur server NX7140; CPU: Intel Xeon E5405
2. Experimental results and analysis

Workload: DBT-5 (simulate securities company’s online transaction processing)
Injection timing: Skip 1000 steps, skip 1000-8000 steps randomly
Fault Injection times: 100 times
Fault Response Time: 8000 microsecond
Injection object: system calls of file operations
Figures 11 and 12 are the fault injection results for the parameters of one bit flip and two bits flip.

Result of fault flip_1_bit injection:
Total: 100 times.
Success: 100 times 100%.
Exit: 13 times 13%.
Signal: 81 times 81%.
Sigalarm: 6 times 6%.
Others: 0 times 0%.
Fail: 0 times 0%.

Figure 11 Fault injection results of parameters of one bit flip

Result of fault flip_2_bit injection:
Total: 100 times.
Success: 100 times 100%.
Exit: 7 times 7%.
Signal: 93 times 93%.
Sigalarm: 0 times 0%.
Others: 0 times 0%.
Fail: 0 times 0%.

Figure 12 Fault injection results of parameters of two bits flip

Fault injection results on all types of bit flip of four types of system calls are shown in Table 4 to Table 7:

---

**Table 4:** Fault Injection Results of File Operation System Calls

<table>
<thead>
<tr>
<th>1 bit inversion</th>
<th>2 bits inversion</th>
<th>32 bits inversion</th>
<th>1 bit set zero</th>
<th>2 bits set zero</th>
<th>32 bits set zero</th>
<th>1 bit set one</th>
<th>2 bits set one</th>
<th>32 bits set one</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>7</td>
<td>13</td>
<td>36</td>
<td>34</td>
<td>31</td>
<td>42</td>
<td>18</td>
<td>17</td>
<td>17</td>
</tr>
<tr>
<td>81</td>
<td>93</td>
<td>87</td>
<td>64</td>
<td>64</td>
<td>68</td>
<td>57</td>
<td>87</td>
<td>82</td>
<td>82</td>
</tr>
<tr>
<td>6</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

**Table 5:** Fault Injection Results of Disk Read and Write System Calls

<table>
<thead>
<tr>
<th>1 bit inversion</th>
<th>2 bits inversion</th>
<th>32 bits inversion</th>
<th>1 bit set zero</th>
<th>2 bits set zero</th>
<th>32 bits set zero</th>
<th>1 bit set one</th>
<th>2 bits set one</th>
<th>32 bits set one</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>70</td>
<td>73</td>
<td>67</td>
<td>81</td>
<td>67</td>
<td>62</td>
<td>77</td>
<td>10</td>
<td>57</td>
</tr>
<tr>
<td>34</td>
<td>37</td>
<td>25</td>
<td>52</td>
<td>18</td>
<td>32</td>
<td>37</td>
<td>22</td>
<td>30</td>
<td>41</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
</tbody>
</table>

© 2013 ACADEMY PUBLISHER
TABLE 6
FAULT INJECTION RESULTS OF PROCESS CONTROL SYSTEM CALLS

<table>
<thead>
<tr>
<th></th>
<th>1 bit inversion</th>
<th>2 bits inversion</th>
<th>32 bits inversion</th>
<th>1 bit set zero</th>
<th>2 bits set zero</th>
<th>32 bits set zero</th>
<th>1 bit set one</th>
<th>2 bits set one</th>
<th>32 bits set one</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>34</td>
<td>6</td>
<td>0</td>
<td>100</td>
<td>100</td>
<td>20</td>
<td>2</td>
<td>0</td>
<td>100</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>13</td>
<td>34</td>
<td>25</td>
<td>0</td>
<td>0</td>
<td>28</td>
<td>35</td>
<td>29</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Alarm</td>
<td>53</td>
<td>60</td>
<td>75</td>
<td>0</td>
<td>0</td>
<td>52</td>
<td>63</td>
<td>71</td>
<td>0</td>
<td></td>
</tr>
</tbody>
</table>

TABLE 7
FAULT INJECTION RESULTS OF SYSTEM SIGNAL SYSTEM CALLS

<table>
<thead>
<tr>
<th></th>
<th>1 bit inversion</th>
<th>2 bits inversion</th>
<th>32 bits inversion</th>
<th>1 bit set zero</th>
<th>2 bits set zero</th>
<th>32 bits set zero</th>
<th>1 bit set one</th>
<th>2 bits set one</th>
<th>32 bits set one</th>
<th>Mask</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>Abort</td>
<td>51</td>
<td>52</td>
<td>44</td>
<td>40</td>
<td>46</td>
<td>52</td>
<td>44</td>
<td>40</td>
<td>53</td>
<td>46</td>
</tr>
<tr>
<td>Alarm</td>
<td>49</td>
<td>48</td>
<td>56</td>
<td>60</td>
<td>54</td>
<td>48</td>
<td>56</td>
<td>60</td>
<td>47</td>
<td>54</td>
</tr>
</tbody>
</table>

The traditional fault injection method is to use ptrace to inject faults into registers and memory to simulate hardware failures. Table 8 is the fault injection results of the traditional fault injection method:

TABLE 8
FAULT INJECTION RESULTS OF THE TRADITIONAL METHOD

<table>
<thead>
<tr>
<th></th>
<th>IP 1 bit flip</th>
<th>Status register 1 bit flip</th>
<th>General register 1 bit flip</th>
<th>Memory/code/data segment flip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exit</td>
<td>15</td>
<td>12</td>
<td>74</td>
<td>100</td>
</tr>
<tr>
<td>Abort</td>
<td>81</td>
<td>0</td>
<td>26</td>
<td>100</td>
</tr>
<tr>
<td>Alarm</td>
<td>4</td>
<td>88</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

From the above results, we can see, the traditional fault injection tool for evaluating the robustness of the system has significant limitations, such as memory fault injection, judging from the robustness of the system are all normal exits and returns an error code. Using the method of abnormal parameters fault injection to the interface of system calls proposed in this paper compared with the traditional fault injection, can effectively lead to failure of the system robustness, and different types of system functionality can be classified to evaluate, which help to assess robustness.

Calculate the system failure rate of this workload:

1) Calculate the average system failure rate after fault injection to the file operation system calls, disk read and write system calls, the process-related system calls and signal-related system calls under this workload: were 76.5%, 29.8%, 16.4% and 46.8% respectively.

2) Under normal circumstances of the workload, count various types of system calls used, results are as follows: system calls related to file operations are 320 times; system calls related to disk read and write are 410 times; system calls related to process control are 155 times; and system calls related to system signals are 747 times.

3) According to the proportion of various types of system calls, calculate the weights respectively: 0.20, 0.25, 0.10 and 0.45.

4) Calculate the system failure rate under this load:

\[
F = \sum_{i=1}^{N} w_i \cdot f_i
\]

\[
= 0.20 \cdot 0.765 + 0.25 \cdot 0.298 + 0.10 \cdot 0.164 + 0.45 \cdot 0.468
\]

\[
= 0.153 + 0.0745 + 0.0164 + 0.2106
\]

\[
= 0.4545
\]

That is the system failure rate under this load is 0.4545.

D. Comparison and Analysis of Different Levels of System Robustness Test Results

1. Robustness comparison and analysis of API layer and DPI layer

Shown in Figure 13, the failure rate caused by DPI fault injection is greater than API, because the protection for functions of system kernel state decreases compared to the application functions [14,15], and therefore the robustness problems detected increase.

![Figure 13 Comparison of DPI-API failure rate](image)

Function Types

In the API fault injection experiments, the case of system crash does not occur, and in the DPI fault injection experiments, five functions lead to system crash, because the kernel stack fault. Because fault injection to the kernel function can trigger a kernel state of failure, the seriousness of the failure sometimes is greater than the failure caused by the user state.

2. Robustness test results comparison and analysis of static function call and dynamic load

The robustness failure generated by fault injection to four different types of system call functions and the corresponding API function calls are shown in Figure 14:
Linux systems, compare and analyze the relationship of tools. Using the integrated fault injection platform, test and the incremental development of robustness testing multi-level fault injection based on abnormal parameters integrated fault injection platform implemented mode and the system robustness of dynamic load. The programming interface of user mode and kernel mode, to test the system robustness injection method to inject faults to dynamic and static, Linux operating system, using the consistent fault based on abnormal parameters for the architecture of paper proposed the multi-level robustness testing strategy improvement, for a fuller evaluation of the system's robustness.

Figure 14 System failure rate comparison for run-time system call and static API functions after fault injection

Visibly, the probability of system failure caused by dynamic fault load is mostly greater than system failure caused by fault injection to the single interface. Especially for fault injection to the system calls associated with system signals in the load, the system failure probability is much larger than the corresponding single API call. There are two reasons:

1) System call interface is the kernel mode function, system failures injecting abnormal parameters in kernel mode are more than in user mode.
2) The spread of the fault caused by abnormal parameters of the function interface in the application will further lead to system failure, and which is closely related to the robustness of the application itself.

V. CONCLUSION

By comparing the classic robustness testing tools, this paper proposed the multi-level robustness testing strategy based on abnormal parameters for the architecture of Linux operating system, using the consistent fault injection method to inject faults to dynamic and static, user mode and kernel mode, to test the system robustness of the programming interface of user mode and kernel mode and the system robustness of dynamic load. The integrated fault injection platform implemented multi-level fault injection based on abnormal parameters and the incremental development of robustness testing tools. Using the integrated fault injection platform, test robustness of different levels for different versions of Linux systems, compare and analyze the relationship of the system failure rate between the robustness of static interface and dynamic load running. As robustness test results are generated from the consistent fault injection strategy, the test results of different levels have good comparability.

Due to the protection mechanism for system calls of high version of the kernel different, fault injection on DPI layer lacks portability, currently only implemented fault injection in the kernel of linux2.4. The fault injection strategy on the application load also requires further improvement, for a fuller evaluation of the system's robustness.

ACKNOWLEDGEMENTS

This work is supported by Key Projects of National High Technology Research and Development Program 863 (No.2008AA01A204) and the National Natural Science Foundation of China (61003047), partly supported by the Scientific Research Project of Zhejiang Education Department (Y201226267). Thanks for the help.

REFERENCES

dependability benchmarking of partitioning operating
systems. 2011 IEEE/IFIP 41st International Conference
on Dependable Systems & Networks (DSN), pp.422-429,
2011.

Tolerance with Exception Handling: Verification and
Validation[J] Journal of Computer Science and

Lin Xiang, born in 1975, is a lecturer in
China Jiliang University, and Ph.d
candidate in Harbin Institute of
Technology. She received her master
degree from Harbin Institute of
Technology, and her major is computer
architecture. Her research interests are
in the areas of computer architecture,
fault tolerant computing and mobile
computing.

Zhan Zhang, born in 1978, is an
associate professor in Harbin Institute of
Technology. He received his Ph.d degree
from Harbin Institute of Technology,
and his major is computer architecture.
His research interests are in the areas of
computer architecture, fault tolerant
computing and mobile computing.

Decheng Zuo was born in 1972 in
China. He received the Ph.D. degree in
School of Computer and Science from
Harbin Institute of Technology in 2001.
He is a professor of the School of
Computer Science and Technology at
the Harbin Institute of Technology. His
main research interests include
fault-tolerant computing, mobile
computing and system reliability
estimation.

Xiaozong Yang was born in 1939 in
China. He received his B.S.degree in
School of Computer and Science from
Harbin Institute of Technology, in 1964.
He is currently a professor and Ph.D.
supervisor in School of Computer
Science and Technology at Harbin
Institute of Technology (CS HIT). His
main researches interests include
computing architecture, fault tolerant
computing.