Testing Loaded Programs Using Fault Injection Technique

S. Manaseer, F. A. Masooud, A. A. Sharieh

Abstract— Fault tolerance is critical in many of today’s large computer systems. This paper focuses on improving fault tolerance through testing. Moreover, it concentrates on the memory faults: how to access the editable part of a process memory space and how this part is affected. A special Software Fault Injection Technique (SFIT) is proposed for this purpose. This is done by sequentially scanning the memory of the target process, and trying to edit maximum number of bytes inside that memory. The technique was implemented and tested on a group of programs in software packages such as jet-audio, Notepad, Microsoft Word, Microsoft Excel, and Microsoft Outlook. The results from the test sample process indicate that the size of the scanned area depends on several factors. These factors are: process size, process type, and virtual memory size of the machine under test. The results show that increasing the process size will increase the scanned memory space. They also show that input-output processes have more scanned area size than other processes. Increasing the virtual memory size will also affect the size of the scanned area but to a certain limit.

Keywords— Complex software systems, Error detection, Fault tolerance, Injection and testing methodology, Memory faults, Process and virtual memory.

I. INTRODUCTION

There are three lines of defense against software faults: Fault avoidance, fault elimination and fault tolerance [9, 10].

The main objective of the Software Fault Injection Technique (SFIT) is to test the fault tolerance capability through injecting faults into the system and analyze if the system can detect and recover from faults as expected [5]. The SFIT is an important technique for at least two reasons: Failure acceleration, and systematic testing. There are several challenges in the pursuit of a methodology for testing fault tolerance [7]: Complexity of software, Dormancy of faults, and Constraint of resource availability.

In section 2, we present related work to fault injection. Section 3 presents the proposed technique. Section 4 explains the results. Section 5 concludes the advantages and drawbacks of the research.
low time-resolution may not be a problem. For short latency faults, such as bus and CPU faults, the approach may fail to capture certain error behavior, like propagation \[7\]. Engineers can solve this problem by taking a hybrid approach, which combines the versatility of software fault injection and the accuracy of hardware monitor. The hybrid approach is well suited for measuring extremely short latencies. However, the hardware monitoring involved can cost more and decrease flexibility by limiting observation points and data storage size, \[5\].

III. THE PROPOSED TECHNIQUE

When a process is running on a machine, and while it resides in memory, there are two parts of the memory space of the process; a read-only part and an editable part. The first part is locked by the system and cannot be altered. The second part is the part of concern for the injector \[4\]. The method used here is aimed to find out the size of the second part of the process and to study the factors that affect this part. The technical details of the method proposed in this paper are as follows:

The injector will go through the following stages.

Get command line.

Create new process and handle command line to it. Windows does this task with full transparency to the child process.

Wait until the process is fully initialized. This is necessary, because the memory of the child process isn't committed yet, which means that the contents of the memory space of the new process is garbage at this point. WaitForInputIdle is used to make sure the memory is committed before the injector starts to work on the memory space.

Faults are injected in the memory space of the child process. After the process memory is committed and the byte check is done, the injection process can be started.

Finally, terminate and leave the new process alone. This is a simple output process. The results are committed to a text file.

Close the process and thread handles.

As shown above, the process flow starts at the process of loading the target process into memory and then waits for the target to be idle and ready to start after the loading operation. This part of the job is controlled by the activator subsystem.

After the target process is loaded into memory and ready to start, the injector starts scanning the memory space of the target process from location 0x00000000 and tries editing each single byte of that process. At this stage, we have one of two options. First, the byte is a locked byte, which will be counted as BAD byte. The second option is that the editing operation succeeds and the byte is a GOOD byte.

The injector will keep memory editing task going on until the target process crashes at some certain point. At that point, the numbers of good, bad, and total bytes scanned are recorded. The percentages for good and bad bytes are calculated, and finally the results are stored into a text file for analysis.

The main contribution of this work is to presenting the technique explained for measuring the size of attackable memory space, and to study the factors that affect this memory space. The technique is not concerned in simulating the running environment in order to catch faults. It is only interested in knowing the ability of a program to be edited inside memory. It is an important point to know that the program needs to have the ability to be edited even if the number of faults that occur in memory is in millions.

IV. RESULTS AND ANALYSIS

The proposed methodology has shown that the number of edited memory locations varies from one target process to another as shown in Tables I and II. The tables demonstrate the results of running the fault injector on 9 different processes. The number represents the average number of bytes over 20 different runs of the program on these software samples.

Table I shows the total size of the area scanned before the process crashes. This is an indication of the size of interfere that the process can take before the injection process become harmful.

The fact that Table III shows is the size and percentage of the edited area of the total area scanned, that is:

\[
\text{Percentage of Edited Bytes} = \frac{(\text{number of successfully edited bytes})}{(\text{Edit bytes} + \text{blocked bytes})}
\]

| SAMPLE ID | NUMBER OF SCANNEDBYTES | NUMBER OFEDITEDBYTES | PERCENTAGE 
of EDITEDBYTES |
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>551546</td>
<td>82182</td>
<td>15%</td>
</tr>
<tr>
<td>1</td>
<td>1380251</td>
<td>77115</td>
<td>6%</td>
</tr>
<tr>
<td>2</td>
<td>1359806</td>
<td>113002</td>
<td>8%</td>
</tr>
<tr>
<td>3</td>
<td>1312855</td>
<td>40645</td>
<td>3%</td>
</tr>
<tr>
<td>4</td>
<td>474341</td>
<td>73223</td>
<td>18%</td>
</tr>
<tr>
<td>5</td>
<td>894649</td>
<td>95192</td>
<td>13%</td>
</tr>
<tr>
<td>6</td>
<td>582838</td>
<td>79367</td>
<td>17%</td>
</tr>
<tr>
<td>7</td>
<td>899303</td>
<td>107514</td>
<td>19%</td>
</tr>
<tr>
<td>8</td>
<td>472738</td>
<td>36226</td>
<td>10%</td>
</tr>
</tbody>
</table>

Table 1: the size of maximum scanned and edited areas for 9 samples.
The number of injected memory locations is affected by more than one factor, such as process type and process size.

Process size: The number of injected memory locations is proportional to the size of the loaded process. The larger the target process is, the higher the number of memory locations that are edited. Figure 1 shows the relation between the sample size in megabytes and the percentage of the injected bytes for five processes out of the nine samples mentioned in Table II. These five processes are non-interactive processes.

The process type: as a general method for classifying programs, in this paper the software samples are of two types, [6]: Input / Output processes and Executable processes.

The injection process itself is affected by some factors such as operating system security policy and virtual memory size. Tables II, III, IV, and V demonstrate the results of the injection process under various virtual memory sizes. The sizes used for this part are four categories, 300, 500, 700, and 1000 Megabytes.

Table II: Injection results for 300 MB virtual memory.

<table>
<thead>
<tr>
<th>Process ID</th>
<th>Total Bytes Scanned</th>
<th>Total Bytes Edited</th>
<th>Percentage of Edited Bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>413438</td>
<td>73582.4</td>
<td>18%</td>
</tr>
<tr>
<td>P1</td>
<td>544041</td>
<td>77824</td>
<td>14%</td>
</tr>
<tr>
<td>P2</td>
<td>1375089</td>
<td>119236.6</td>
<td>9%</td>
</tr>
<tr>
<td>P3</td>
<td>1407933</td>
<td>108368.6</td>
<td>8%</td>
</tr>
<tr>
<td>P4</td>
<td>338722</td>
<td>34331.2</td>
<td>11%</td>
</tr>
<tr>
<td>P5</td>
<td>769390</td>
<td>92196.4</td>
<td>16%</td>
</tr>
<tr>
<td>P6</td>
<td>956496</td>
<td>106364.2</td>
<td>14%</td>
</tr>
<tr>
<td>P7</td>
<td>1270044</td>
<td>30431.2</td>
<td>2%</td>
</tr>
<tr>
<td>P8</td>
<td>317366</td>
<td>43212.6</td>
<td>13%</td>
</tr>
</tbody>
</table>

Analyzing the results in Tables 2, 3, 4 and 5 can lead to the fact shown in Figure 2. The fact indicated by this figure is that increasing the virtual memory size will lead to a decrement in the average number of memory locations scanned. One fact to notice is that the factor of page size will stop affecting the result of the injection process at some point. The virtual memory size will not affect the scanned area size when the process is not using the added virtual memory space.
V. CONCLUSIONS

The main contribution of this paper is presenting a technique for measuring the size of attackable memory space, and to study the factors that affect this memory space. The technique is not concerned in simulating the running environment in order to catch faults. It is only interested in knowing the ability of a program to be edited inside memory to provide high-quality services to users during abnormal conditions and failures.

The results of the method proposed in this paper lead to the following conclusions:

Measuring the maximum scanned area of the memory will save the time that is needed for a fault injection process that uses simulation aspects.

All the samples used in the paper did reach a crash state at some point. This fact means that the factor measured, i.e. the maximum scanned area, is a valid aspect to use for estimating the stability of a running process in the memory.

Depending on the size of the process in memory, the size of the scanned area will differ. The results showed that the larger the process is, the larger the scanned area.

The size of the virtual memory has a reverse effect on the scanned area size. Increasing the size of the virtual memory will decrease the area size directly.

At some point, the virtual memory size becomes a non deterministic factor. This is the case when the page size in the virtual memory is larger than the size of the target process itself.

The security policy of the operating system also has a direct effect on the size of the scanned area. The access to a memory location must have the permission of the operating system's rules before it is ready for update.

The main achievement of this research is an innovation of a technique that estimates software stability. This technique is not concerned in simulating the running environment in order to catch faults; it is only interested in knowing the ability of a program to be edited inside memory. However, the proposed method has two drawbacks:

The termination process of the injection operation depends on the crash of the target process, i.e. no results until the target process crashes. This operation deals with memory content directly. So, if it is not used carefully, it can corrupt other process in memory. Next steps might go towards: Developing a mechanism to investigate all states of the injected process, and not to wait for a crash in the process.

A better classification of process types will make the injection process give much better results.

REFERENCES


Fig 2: Average Area Scanned (MB) vs. Virtual memory size