International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:1, No:3, 2007

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.

S. Manseer is a research assistant at theuniversity of Jordan. Email:saher@dcs.gla.ac.uk

F. A. Massoud is with The University of Jordan, Amman-Jordan, Vice Dean of KASIT (e-mail:) fawaz@ju.edu.jo.

A. A. Sharieh is with the University of Jordan, Dean of King Abdullah II School for Information Technology. (e-mail: sharieh@ju.edu.jo)

II. LITERATURE REVIEW

A huge amount of time and money is often spent to verify that a system is fault free. But some faults can not be found due to either lack of time or simply overlooked faults, [1, 2]. In a big system that is used by thousands or maybe even millions of users' everyday, some faults may occur that the designers never even thought were possible, [3]. Faults in a software program are often called bugs. Every programmer sees his/her program bug-free until the next bug is discovered. Many bugs can often be hard to locate, these are called Heisenbugs (whereas the Bohrbugs causes predictable failures) [3].

In order to make sure that a fault will not crash the system or even stay hidden in the system without being discovered Fault Injection is used, [8]. The principle is to inject a fault into the system, a fault as close to a "real" fault that can occur in the field, and check how the system reacts. A simple fault-tolerance device could be a watchdog that "watches" over applications and restarts the application if it should hang or crash. The fault can result in an error when executed. Errors should not go undetected, so some sort of error detection mechanism is necessary. This could be as simple as parity checking. Once the error is detected, it should be localized (fault-diagnostic) [7].

The Fault Injection can be achieved on different levels. Low-level fault injection in which one tries to simulate hardware faults to check how the software reacts. This can include changing memory and register contents etc. High-level fault injection that means changing code, corrupt data or change program states.

The Fault injection can be done by injecting faults manually, using commercial tools or make a program for automatic injection of faults according to statistical models and algorithms. The commercial tools developed for fault injection are often called Software Implemented Fault Injector (SWFI). These are often relatively inexpensive compared to other tools such as hardware injectors. Software Fault Injection is a flexible approach of injecting faults compared to hardware injection, but it has shortcomings, [5]:

1. It cannot inject faults into locations that are inaccessible to software.
2. The software instrumentation may disturb the workload running on the target system and even change the structure of original software. Careful design of the injection environment can minimize perturbation to the workload.

The poor time-resolution of the approach may cause fidelity problems. For long latency faults, such as memory faults, the

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 commandline.

Create new process and handle commandline 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 out put 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:

Percentage of Edited Bytes = (number of successfully edited bytes) / (Edit bytes + blocked bytes)

TABLE 1: the size of maximum scanned and edited areas for 9 samples.

SAMPLE ID	NUMBER OF SCANNED BYTES	GOOD BYTES	
		NUMBER OF EDITED BYTES	PERCENTAGE OF EDITED BYTES
0	551546	82182	15%
1	1380251	77115	6%
2	1359806	113002	8%
3	1312855	40645	3%
4	474341	73223	18%
5	894649	95192	13%
6	582838	79367	17%
7	899303	107514	19%
8	472738	36226	10%

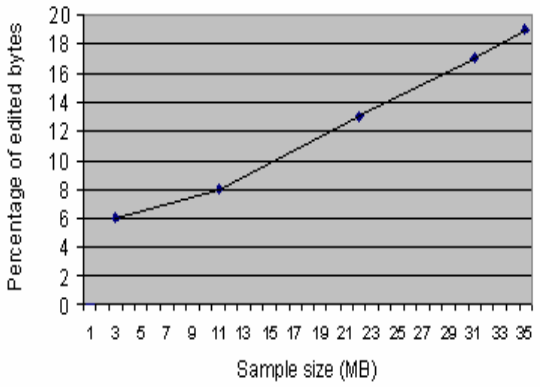

Fig 1 : Sample size vs. Percentage of bytes edited

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.

Process ID	Total Bytes Scanned	Number of Edited Bytes	Percentage of Edited Bytes
P0	413438	73582.4	18%
P1	544041	77824	14%
P2	1375089	119236.6	9%
P3	1407933	108368.6	8%
P4	532545	38503.4	9%
P5	552612	70583.8	17%
P6	1073726	109633	11%
P7	1297792	39696.8	3%
P8	306105	53342.4	17%

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.

Table III : Injection results for 500 MB virtual memory.

Process ID	Total Bytes Scanned	Edited Bytes	Percentage of Edited Bytes
P0	494586	72196.4	17%
P1	524519	77824.2	15%
P2	1393035	127435.8	9%
P3	1432674	103365.2	7%
P4	338722	34331.2	11%
P5	769390	92196.4	16%
P6	956496	106364.2	14%
P7	1270044	30431.2	2%
P8	317366	43212.6	13%

Table IV:

Process ID	Total Bytes Scanned	Edited Bytes	Percentage of Edited Bytes
P0	494586	72196.4	17%
P1	524519	77824.2	15%
P2	1393035	127435.8	9%
P3	1432674	103365.2	7%
P4	338722	34331.2	11%
P5	769390	92196.4	16%
P6	956496	106364.2	14%
P7	1270044	30431.2	2%
P8	317366	43212.6	13%

Table IV : Injection results for 700 MB virtual memory.

process ID	Total Bytes Scanned	Edited Bytes	Percentage of Edited Bytes
P0	421035	62370.8	16%
P1	486671	89293	15%
P2	1254795	105411	8%
P3	1371124	80882.4	6%
P4	394304	38503	10%
P5	501901	67822.4	19%
P6	956758	99357.6	13%
P7	1414483	40141	3%
P8	515391	75905.4	20%

Table V: Injection results for 1000 MB virtual memory.

Process ID	Total Bytes Scanned	Edited Bytes	Percentage of Edited Bytes
P0	369270	63351.4	17%
P1	507973	90931.4	14%
P2	1247716	97672.8	7%
P3	1475545	106831.4	7%
P4	294713	33162.4	11%
P5	355284	73728.4	21%
P6	1032919	106309.4	12%
P7	1290734	35531	3%
P8	346871	73659.6	22%

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

[1] Broadwell P., Sastry N., and Traupma J., (2001). **"FIG: A Prototype Tool for Online Verification of Recovery Mechanisms "**. ICS SHAMAN Workshop '02 New York, New York USA Copyright 2001 ACM.

[2] Broadwell, P. and Ong E. (2002)."**A Comparison of Static Analysis and Fault Injection Techniques for Developing Robust System Services",** a project paper, retrieved (January2004), from (http://www.cs.berkeley.edu/~pbwell/papers/saswifi.pdf).

[3] Lai M. and Wang S., (1995). **"Software Fault Tolerance"**, Wiley & Sons LTd, New York.

[4] Manaseer S. (2004). **"Software Testing Using Software Fault Injection"**. A Master Thesis in Computer Science , KASIT, University of Jordan, Jordan.

[5] Sanders W., (2003)**." Fault Injection Methods and Mechanisms".** Department of Electrical and Computer Engineering and Coordinated Science Laboratory, University of Illinois at Urbana-Champaign.

[6] Silberschatz A. Galvin P., and Gagne G., (2003). **Operating System Concepts**. Wiley & Sons, New York.

[7] Thorhuns R., (2000). **"Software Fault Injection Testing".** A Master thesis in Electronic System Design. KTH, Stockholm.

[8] Torres-Pomales W., (2000)."**Software Fault Tolerance: A Tutorial".** NASA Center for Aerospace Information (CASI) National Technical Information Service (NTIS).

[9] Voas J., Charron F., and McGraw G., (1997). **"Predicting How Badly 'Good' Software Can Behave"**, IEEE Software, 14(4):73-83.

[10] Voas M., McGraw G., (1998). **Software Fault Injection, Inoculating Programs against Errors**. Wiley & Sons, New York.

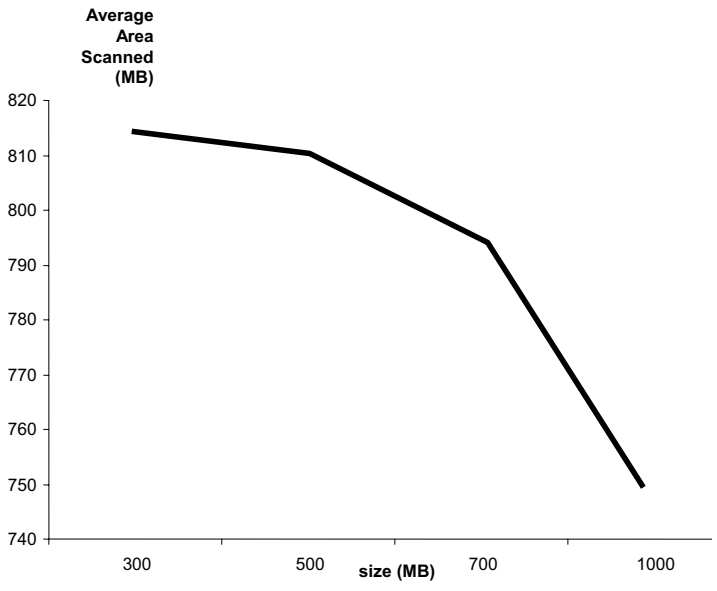

Fig 2: Average Area Scanned (MB) vs. Virtual memory size