Fuzzing the Out-of-Memory Killer on Embedded Linux: An Adaptive Random Approach

K. Y. Sim
School of Engineering, Computing and Science, Swinburne University of Technology (Sarawak Campus)
Jalan Simpang Tiga
93350 Kuching, Sarawak, Malaysia
ksim@swinburne.edu.my

F.-C. Kuo
Centre for Software Analysis and Testing, Swinburne University of Technology
John St. Hawthorn 3122
Victoria, Australia
dkuo@swin.edu.au

R. Merkel
Centre for Software Analysis and Testing, Swinburne University of Technology
John St. Hawthorn 3122
Victoria, Australia
rmerkel@swin.edu.au

ABSTRACT
Fuzzing is an automated black-box testing technique conducted with a destructive aim to crash (that is, to reveal failures in) the software under test. In this paper, we propose an adaptive random approach to fuzz the Out-Of-Memory (OOM) Killer on an embedded Linux distribution. The fuzzing process has revealed OOM Killer failures that cause the Linux kernel to remain in the OOM condition and become non-responsive. We have also found that the OOM Killer failures are more likely to occur when the Linux kernel has a higher over-commitment of memory requests. Finally, we have shown that the proposed adaptive random approach for fuzzing can reveal an OOM Killer failure with significantly fewer test inputs compared to the pure random approach.

Categories and Subject Descriptors
D.4.7: [OPERATING SYSTEMS]: Organization and Design - *Real-time systems and embedded systems.*

Keywords
Embedded Linux, Fuzzing, Adaptive Random Testing, Out-of-Memory Killer.

1. INTRODUCTION
An *out-of-memory (OOM) condition* occurs on Linux when the operating system is not able to allocate more memory to running processes. In order to recover from the OOM condition, an *OOM Killer* mechanism is usually deployed to select and kill (terminate) one of the memory consuming processes in order to regain memory space.

There are two primary reasons why the OOM Killer plays an important role in the Linux operating system, particularly one that runs on an embedded system. Firstly, the Linux kernel has the capability to accept and commit memory requests beyond the available memory space. Over-committing memory requests is done based on the assumption that programs are usually written to request more memory than actually used. However, adversity may occur and Linux will run into an OOM condition. Secondly, unlike workstations and servers, embedded systems usually have limited physical memory and no extra swap space to use. When an OOM condition occurs, the OOM Killer will use certain heuristics to select one of the memory consuming processes to be killed. If the OOM Killer fails to do so, the Linux kernel may remain in the OOM condition and become non-responsive.

Fuzzing is an automated black-box testing technique conducted with a destructive aim to crash (that is, to reveal failures in) the software under test [12]. In conventional fuzzing, pure random *test inputs* have been used to test the software under test until it crashes. Fuzzing has been widely used to reveal vulnerabilities in software security [13] and failures in software quality assurance [3][14]. Past studies have demonstrated that fuzzing could effectively reveal critical failures in UNIX utility programs [12].

Recently, *adaptive random testing* (ART) [6] has been proposed as a more effective alternative to random testing (RT) in revealing failure. Based on the observation that inputs that cause the system to fail (known as *failure-causing inputs*) are often contiguous, ART attempts to evenly spread test inputs across the input domain to improve the chance of hitting a failure. Numerous studies [4][5][7][8][10] have shown that ART requires significantly fewer test inputs to reveal the first failure compared to RT.

In this paper, we aim to reveal possible failures of the OOM Killer. We developed a test driver to simulate a situation where there exists a demanding process that endlessly requests memory of a certain *memory request size* (MRS). It is expected that such endless memory requests will continuously produce an OOM condition and cause the OOM killer to kill processes, until the test driver is eventually selected to be killed by the OOM killer. However, if the OOM killer does not behave as expected to kill the test driver eventually, the system will enter a non-responsive state and an OOM killer failure is said to be revealed.

A test is said to be conducted each time an MRS is used as an input for the test driver to make endless memory requests. We propose an adaptive random approach to generate the MRSs as test inputs in this study. The effectiveness of these test inputs in revealing an OOM Killer failure is compared to a pure random approach through experiments.

SAC'11, March 21-25, 2011, TaiChung, Taiwan.

The contributions of this paper are twofold; First, we found through fuzzing that the OOM Killer indeed failed when certain memory request sizes (MRSs) are used to endlessly request memory. It was further found that the OOM Killer failures are more likely to occur when the Linux kernel has a higher over-commitment of memory requests. Second, we showed that the proposed adaptive random approach is more effective in fuzzing as it requires significantly fewer MRSs as test inputs to reveal an OOM Killer failure compared to a pure random approach.

The remainder of this paper is organized as follows: Section 2 presents the algorithm to generate adaptive random MRSs as test inputs to fuzz the OOM Killer on embedded Linux. The embedded system targeted for fuzzing and the fuzz procedures are described in Section 3. Section 4 presents the fuzzing results and compare the failure detection effectiveness of the adaptive random and pure random fuzzing approaches. Section 5 concludes the paper.

2. Generating Test Inputs for Fuzzing

Testing is a very costly but important activity to detect failures in software systems. ART improves the failure-detection effectiveness of RT by evenly spreading random test inputs. It was motivated by observations that failure-causing inputs are often contiguous (or clustered together) [1][2][9] rather than standing alone. Given a set of previously executed test inputs that have not revealed any failure, the next test input is more likely to hit a contiguous failure region (and therefore, reveal the failure) if it is located away from the previously executed test inputs. Hence, *evenly-spread* random test inputs generated by ART have better chances of revealing the first failure than the pure random test inputs [6].

Initialize ***E*** as an empty set.
Input the *maximumNumberOfTestInput*.
Generate an MRS randomly as a test input, *t*, where *1Byte≤t≤1MByte*.
Add *t* to ***E***.
Write *t* to the adaptive random test file.
Loop for *(maximumNumberOfTestInput-1)* times
  Generate *k* random candidates to form a candidate set $\boldsymbol{C}=\{c_i \mid i=1,2,3,\ldots, k$ and $1Byte\le c_i\le 1MByte\}$, where *k* is a positive constant integer. *k* is set to 10 in this algorithm.
  Find $c_j \in \boldsymbol{C}$ such that, among all the elements in ***C***, c_j has the longest distance to its nearest neighbor in ***E***.
  Add c_j to ***E***.
  Append c_j to the adaptive random test file.
End of Loop

Figure 1: Algorithm to generate adaptive random MRSs as test inputs for fuzzing.

Input the *maximumNumberOfTestInput*.
Loop for *maximumNumberOfTestInput* times
  Generate an MRS randomly as test input, *t* (*1Byte≤t≤1MByte*).
  Append *t* to the pure random test file.
End of Loop

Figure 2: Algorithm to generate pure random MRSs as test inputs for fuzzing.

Given the advantage of ART over RT in detecting a failure, we propose an adaptive random approach to replace the pure random approach in fuzzing. The Fixed Size Candidate Set ART (FSCS-ART) [6] algorithm is being used to generate MRSs as test inputs for the fuzzing process. The algorithm is outlined in Figure 1. The first test input is generated randomly. Subsequent test inputs are selected from a set of *k* randomly generated candidates. For each candidate, c_i, the minimum distance, d_i, from the previously generated test inputs in ***E*** is computed. The candidate with the largest d_i is selected as the next test input. The same process is repeated to generate a series of test inputs.

As the benchmark for evaluating the performance of the proposed adaptive random approach to generate test inputs for fuzzing of the OOM Killer, a pure random approach to generate test inputs is outlined in Figure 2. In this algorithm, each MRS is selected randomly as a test input.

3. Experiments

3.1 Fuzz target

Fuzzing is conducted on a TS-7260 ARM™ embedded system that is based upon the Cirrus EP9302 ARM9 200MHz CPU. This TS-7260 ARM embedded system comes with 32MB SDRAM and 32MB on-board Flash. It uses the SDRAM as the fast access volatile memory to run applications by the processor and the on-board Flash as non-volatile memory for storage purpose.

The operating system for the TS-7260 ARM embedded system is TS-7 Embedded Linux (TS-Kernel 2.4.26), which is installed by default in on-board Flash memory. TS-7 Embedded Linux is a full-featured real-time capable Linux system. It is loaded by default with the Redboot boot-loader during boot-up. After all the processes have been loaded by the operating system after a typical boot-up, the embedded system has approximately 28,696KB of memory with no swap space, where approximately 5,668KB are used by running processes and 23,028KB are free (this information can be obtained by using the *free* command on Linux shell).

3.2 Fuzz Procedure

In this study, a test driver is developed to simulate a situation where there exists a demanding process that endlessly requests memory of a certain MRS. It is expected that such endless memory requests will continuously create an OOM condition. When an OOM condition occurs, the OOM Killer is supposed to select and kill one of the memory consuming processes running on Linux to regain memory space. The process selected to be killed could be the test driver or other processes running on

Linux. If the test driver is selected to be killed, then its endless memory requests of MRS will cease and Linux kernel will recover from the OOM condition. On the other hand, if other running process is selected to be killed to regain memory, the test driver will continue its endless memory requests and hence, create another OOM condition. Sooner or later, the test driver will be selected as the process to be killed because, eventually, there will be no other running processes that can be killed. An OOM Killer failure is said to occur if the OOM Killer failed to kill this test driver and caused the Linux kernel to remain in an OOM condition and become non-responsive.

Initialize *testInputCount* to *zero.*

Open a test file for reading.

Repeat

Increment *testInputCount* by *one.*

Record *testInputCount.*

Read the next MRS from the test file.

Invoke the test driver (outlined in Figure 4a and Figure 4b) and pass the MRS to it.

Until the OOM Killer fails to kill the test driver and the system becomes non-responsive (for five minutes or more).

Reset the hardware manually.

Retrieve the last recorded value of *testInputCount* and record it as *finalTestInputCount.*

Figure 3: Fuzz Procedure to reveal possible failure in the OOM Killer

TestDriverA(MRS)**:**

Loop endlessly

Request memory of MRS

if the memory request of MRS is successful, then

Fill out 100% of the requested memory with data.

End if

End of Loop

End of *TestDriverA*

Figure 4a: The algorithms for the *TestDriverA* used to continuously create OOM conditions.

TestDriverB(MRS)**:**

Loop endlessly

Request memory of MRS

if the memory request of MRS is successful, then

Fill out 5% of the requested memory with data.

End if

End of Loop

End of *TestDriverB*

Figure 4b: The algorithms for the *TestDriverB* used to continuously create OOM conditions.

Figure 3 outlines our fuzz procedure, which is implemented as a Linux shell script. In the fuzz procedure, a test driver (outlined in Figure 4a and Figure 4b) is invoked and an MRS is passed to the test driver. The test driver endlessly requests for memory of MRS to continuously create OOM conditions and forces the OOM Killer to kill it eventually. If the OOM Killer successfully kills the test driver (that is, no failure occurs), the fuzz procedure will invoke the test driver again with another MRS. This is repeated until the OOM Killer fails to kill the test driver and the system becomes non-responsive (that is, a failure has occurred).

It is worth noting that once an OOM Killer failure causes the Linux kernel to become non-responsive, a manual hardware reset for recovery is required and hence, fuzzing has to be terminated. The last recorded *testInputCount* (that is, the number of MRSs required to reveal an OOM Killer failure) is retrieved and recorded as *finalTestInputCount*. The *finalTestInputCount* will be used as the metric to compare the failure detection effectiveness of the proposed adaptive random approach to the pure random approach to generate test inputs for fuzzing. The smaller the *finalTestInputCount* (that is, the smaller the number of MRSs used to detect a failure), the more effective a testing approach will be.

Due to the non-deterministic (random) nature of MRSs generated for fuzzing, the fuzzing experiment needs to be repeated to collect adequate samples of *finalTestInputCount* values. However, as a manual hardware reset is required upon revealing an OOM Killer failure, it is prohibitively laborious to collect a large number of samples for *finalTestInputCount*. Therefore, we only repeat the fuzz procedure 30 times to obtain 30 samples of *finalTestInputCount* for statistical analysis.

To evaluate the effect of over-commitment of memory requests on OOM Killer failures, two versions of test drivers are used in the fuzz procedure, namely, *TestDriverA* and *TestDriverB*. The algorithms for the test drivers are outlined in Figure 4a and Figure 4b. Both *TestDriverA* and *TestDriverB* are implemented as C programs.

In *TestDriverA*, the requested memory of MRS will be fully filled with data upon successful memory allocation by the Linux kernel (that is, 100% utilization). As the requested memory is fully

utilized, the Linux kernel has limited opportunity to over-commit memory requests. On the other hand, in *TestDriverB*, only 5% of the requested memory of MRS will be filled with data (that is, 5% utilization). With 5% utilization of the requested memory of MRS, the Linux kernel can have a higher over-commitment of memory requests, which is up to 20 times (100% / 5% = 20) of the available memory. As an example, suppose that 25MB of free memory is available to an embedded Linux. When *TestDriverB* is run, the Linux kernel can over-commit memory requests up to 20X25MB = 500MB before an OOM condition occurs because only 5% the memory requested by *TestDriverB* is actually utilized (that is, filled with data).

The *TestDriverA* and *TestDriverB* will receive an MRS as the test input from the fuzz procedure and then endlessly requests memory of the MRS to continuously create OOM conditions. The MRS passed to both test drivers can be test inputs read from the adaptive random test file (generated by algorithm in Figure 1) or test inputs read from the pure random test file (generated by algorithm in Figure 2).

In summary, to evaluate the effectiveness in revealing OOM Killer failures, the fuzz procedure is repeated to collect 30 samples of *finalTestInputCount* each with both pure random test inputs and adaptive random inputs, using *TestDriverA* and *TestDriverB*.

4. Results and Performance Evaluations

Applying the fuzz procedures to TS-7 Embedded Linux under the TS-7260 ARM embedded system has successfully revealed failures of the OOM Killer. Among all MRSs (ranged from 1Byte to 1MByte)_used in the fuzzy procedure, 30 MRSs that caused the OOM Killer to fail (known as *failure-causing MRSs*) were found. They are recorded and plotted into histograms in Figure 5 and Figure 6, respectively. Three interesting observations can be made from these histograms.

Firstly, for *TestDriverB* (5% utilization of requested memory), the failure-causing MRSs generated by both the adaptive random and the pure random approaches fall only into one histogram interval, that is, from 0.5Mbytes to 0.6Mbytes. This is a strong indication that the OOM Killer failures are indeed related to and caused by certain MRSs. Some bugs may possibly exist in the heuristics that the OOM Killer uses to select the process to be terminated.

Secondly, the failure patterns of the OOM Killer are dependent on the level of utilization of the requested memory, which directly affects the level of memory over-commitment by the Linux kernel. When 100% of the requested memory is utilized, the failure-causing MRSs are more wide spread. On the other hand, when only 5% of the requested memory is utilized, the failure-causing MRSs are spread narrowly within the range of 0.5Mbytes to 0.6Mbytes.

Thirdly, for *TestDriverA* (100% utilization of requested memory), it can be observed that failure-causing MRSs generated by the adaptive random fuzzing approach fall into seven histogram intervals. On the other hand, the failure-causing MRSs generated by the pure random fuzzing approach fall into only six histogram intervals. This suggests that the evenly-spread MRSs generated by the adaptive random approach are able to reveal failures from more diverse MRSs than the pure random approach. It concurs with the previous study [7] that evenly spreading of test inputs does explore more different failures, and consequently, improve the failure-detection effectiveness.

To compare the effectiveness of the adaptive random approach and pure random approach in revealing the OOM failure, the number of test inputs required to reveal the first failure (*finalTestInputCount*) is used as the metric for performance evaluation. 30 samples of *finalTestInputCount* have been collected for both the adaptive random and pure random approach using *TestDriverA* and *TestDriverB*.

The statistics of the number of test inputs required to reveal the first failure (*finalTestInputCount*) when *TestDriverA* is used are presented in Table 1. From Table 1, it can be observed that, on average, 83.7 pure random test inputs are required to reveal the first OOM Killer failure. On the other hand, only an average of 51.3 adaptive random test inputs are required to reveal the first OOM Killer failure. In other words, the adaptive random approach requires significantly fewer test inputs (smaller *finalTestInputCount*) to reveal the first OOM Killer failure compared to the pure random approach. The mean, 25-percentile, median and 75-percentile of *finalTestInputCount* values for adaptive random test inputs are approximately 40% lower than those of pure random inputs. In addition, the *finalTestInputCount* values for adaptive random test inputs also have smaller standard deviation and confidence interval of mean, which suggest higher reliability [11] for the *finalTestInputCount* values collected by the adaptive random approach.

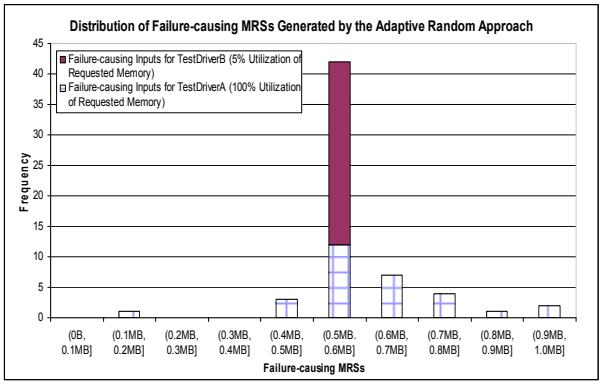

Figure 5: Histogram of failure-causing MRSs that were generated by the adaptive random approach.

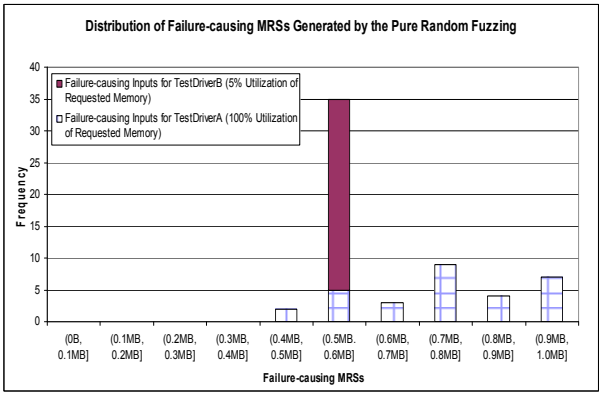

Figure 6: Histogram of failure-causing MRSs that were generated by the pure random approach.

Similar observations can be made from Table 2, which presents the experiment results from running *TestDriverB* where only 5% of the requested memory is utilized. On average, 18.3 pure random test inputs are required to reveal the first OOM Killer failure. On the other hand, only an average of 12.7 adaptive random test inputs are required to reveal the first OOM Killer failure. This represents a 30% improvement in mean. However, the minimum, 25-percentile and median of *finalTestInputCount* values are similar for both pure random and adaptive random approaches. While the improvement of the adaptive random approach over the pure random approach is less obvious in Table 2, the lower 75-percentile and maximum do suggest that the adaptive random approach requires fewer test inputs to reveal the first OOM Killer failure in the worse cases. Overall, the adaptive random approach still performed better than the pure random approach in revealing the first OOM Killer failure when *TestDriverB* is used.

By comparing the results in Table 1 and Table 2, it can be concluded that significantly fewer test inputs (smaller *testInputCount*) are required to reveal an OOM Killer failure when only 5% of the requested memory is utilized compared to 100% utilization. In other words, the OOM Killer has a significantly higher failure rate (that is, more likely to fail) when TS-7 Embedded Linux has a higher over-commitment of memory requests. This is very detrimental for TS-7 Embedded Linux as the risk of running into an OOM condition increases as the operating system over-commits more memory requests.

Table 1. The performance of the adaptive random approach vs. pure random approach for the fuzz procedure running *TestDriverA* (100% utilization of requested memory).

Statistics	Number of test inputs required to reveal the first failure, *finalTestInputCount*	
	Pure Random	Adaptive Random
Number of Samples	30	30
Mean	83.7	51.3
Standard Deviation	57.9	44.0
Confidence Interval of mean (at 95% Confidence Level)	±20.7	±15.8
25 Percentile	38.8	21.8
Median (50 Percentile)	70.0	39.0
75 Percentile	128.8	78.8
Minimum	3	2
Maximum	198	158

Table 2. The performance of the adaptive random approach vs. pure random approach for the fuzz procedure running *TestDriverB* (5% utilization of requested memory).

Statistics	Number of test inputs required to reveal the first failure, *finalTestInputCount*	
	Pure Random	Adaptive Random
Number of Samples	30	30
Mean	18.3	12.7
Standard Deviation	17.7	8.5
Confidence Interval of mean (at 95% Confidence Level)	±6.3	±3.0
25 Percentile	7.0	7.0
Median (50 Percentile)	13.0	12.0
75 Percentile	22.8	16.0
Minimum	4	3
Maximum	87	38

From the experiment results, we could observe that the OOM failures are related to the MRS and the utilization level of requested memory. Such failures may be caused by the OOM Killer itself or its interaction with other system configurations. An extensive and systematic analysis is required to diagnose the root cause of the OOM Killer failures reported here. However, it is beyond the scope of this paper.

5. Conclusion

When an OOM condition occurs, the OOM Killer must select one of the memory consuming processes to be terminated. If the OOM Killer fails, the Linux kernel will be unable to recover from the OOM condition and may enter a non-responsive state. Given (1.) the capability of the Linux kernel to over-commits memory requests, (2.) the limited physical memory and the absence of extra swap space in an embedded system, the OOM condition is likely to occur on embedded Linux. Therefore, the OOM Killer plays an important role in embedded Linux systems.

Fuzzing is an automated black-box testing technique conducted with a destructive aim to crash (that is, to reveal failures in) the software under test [12]. In this paper, we propose an adaptive random approach to generate test inputs to fuzz the Out-of-memory (OOM) Killer on an embedded Linux distribution. Our fuzzing procedure has successfully revealed failures in the OOM Killer for various memory request sizes (MRSs) on an embedded Linux distribution. We have also discovered that the OOM killer is more likely to fail when the Linux kernel has a higher over-commitment of memory requests. Such failures are very detrimental to the operating system because the risk of running

into an OOM condition increases as the operating system over-commits more memory requests.

Our experiment results show that the proposed adaptive random approach requires significantly fewer test inputs to reveal an OOM Killer failure compared to the conventional pure random approach to fuzzing. The improvement is more significant (approximately 40% improvement) when *TestDriverA* (that utilizes 100% of requested memory) is used in the fuzz procedure. When *TestDriverB* (that utilizes only 5% of requested memory) is used in the fuzz procedure, the improvement is less obvious but the adaptive random approach still performs better. These findings reconfirm results from previous studies that adaptive random testing is a more effective alternative for random testing in revealing failures.

The histograms of the failure-causing MRSs (Figure 5 and Figure 6) have provided interesting insights to the failures of OOM Killer and the proposed adaptive random approach to fuzzing. The distribution of failure-causing MRSs for *TestDriverB* (5% utilization of requested memory) provides strong indication that the OOM Killer failures are indeed related to and caused by certain MRSs. On the other hand, it can also be observed that the failure patterns of the OOM Killer are dependent on the level of utilization of the requested memory, which directly affects the level of memory over-commitment by the Linux kernel. Finally, the distribution of failure-causing MRSs for *TestDriverA* suggests that the evenly-spread MRSs generated by the adaptive random approach are able to reveal failures from more diverse MRSs than the pure random approach.

This paper is a pilot study on using the adaptive random approach for embedded system testing. As for future work, we plan to analyze the OOM Killer implementation to diagnose the root cause of its failures reported here. We also plan to further explore the applications of the adaptive random approach on other aspects of embedded systems - those beyond memory management.

6. ACKNOWLEDGMENTS

This work was supported by a discovery grant of the Australian Research Council (project no. ARC DP DP0984760).

7. REFERENCES

[1] Ammann, P.E. and Knight, J.C., Data diversity: an approach to software fault tolerance, IEEE Transactions on Computers, 1988, 37(4):418-425.

[2] Bishop, P.G., The variation of Software Survival Times for different operation input profiles, Proceedings of the 23rd International Symposium on Fault-Tolerant Computing, IEEE Computer Society Press, 1993, pp. 98-107.

[3] Brummayer, R., Lonsing, F., Biere, A. Automated Testing and Debugging of SAT and QBF Solvers. In Proc. 13th Intl. Conf. on Theory and Applications of Satisfiability Testing (SAT'10), Lecture Notes in Computer Science (LNCS) vol. 6175, pages 44-57, Springer 2010.

[4] Chan, K. P., Chen, T. Y., and Towey, D. Restricted random testing: Adaptive random testing by exclusion. Accepted to appear in International Journal of Software Engineering and Knowledge Engineering, 2006.

[5] Chen, T.Y., Eddy, G., Merkel, R.G. and Wong, P.K., Adaptive random testing through dynamic partitioning. In: Proceedings of the 4th International Conference on Quality Software (QSIC 2004). IEEE Computer Society Press, Los Alamitos, CA, pp. 79–86, 2004.

[6] Chen, T.Y., Leung, H. and Mak, I.K., Adaptive random testing. In: Proceedings of the Ninth Asian Computing Science Conference (ASIAN'04), Lecture Notes in Computer Science, vol. 3321, pp. 320–329, 2004.

[7] Chen, T. Y., Kuo, F.-C., Merkel, R. G., and Tse, T. H., Adaptive random testing: the ART of test case diversity. Journal of Systems and Software, 83 (1): 60–66, 2010.

[8] Ciupa, I., Leitner, A., Oriol, M. and Meyer, B., ARTOO: adaptive random testing for object-oriented software. In: Proceedings of the 30th International Conference on Software Engineering (ICSE 2008). ACM Press, New York, NY, pp. 71–80, 2008.

[9] Finelli, G.B. NASA Software Failure Characterization Experiments, Reliability Engineering and System Safety, IEEE Computer Society Press, 1993, 32(1-2): 155-169.

[10] Johannes, M. and Christoph, S., An Empirical Analysis and Comparison of Random Testing Techniques, Proceedings of the 2006 ACM/IEEE international symposium on Empirical software engineering, 105-114, 2006.

[11] Liu, Y. and Zhu, H. An Experimental Evaluation of the Reliability of Adaptive Random Testing Methods. In Proceedings of the 2008 Second international Conference on Secure System integration and Reliability Improvement (July 14 - 17, 2008). SSIRI. IEEE Computer Society, Washington, 2008.

[12] Miller, B., Koski, D., Lee, C., Maganty, V., Murthy, R., Natarajan, A. and Steidl, J.: Fuzz Revisited: A Re-examination of the Reliability of UNIX Utilities and Services. Technical Report CS-TR-1995-1268, University of Wisconsin, Madison, 1995.

[13] Sutton, M., Greene, A. and Amini, P.: Fuzzing – Brute Force Vulnerability Discovery. Pearson Ed., London, 2007.

[14] Takanen, A., Demott, J. and Miller, C.: Fuzzing for Software Security Testing and Quality Assurance. Artech House, 2008.