EMULATING SOFTWARE AND HARDWARE FAULTS IN A MODIFIED LINUX REAL-TIME ENVIRONMENT

Arsim Susuri[1], Mentor Hamiti[2], Besnik Selimi[2], Artan Luma[2]

[1]University of Prizren, Prizren; [2]South East European University, Tetovo
e-mails: arsim.susuri@uni-prizren.com, m.hamiti@seeu.edu.mk, b.selimi@seeu.edu.mk, a.luma@seeu.edu.mk
[1] Kosovo, [2] Macedonia

Abstract: The purpose of this paper is to validate the real-time system in the Linux-RTAI/LXRT (Real Time Application Interface/Linux Real-time) environment, in the presence of hardware and software faults. Due to the simplicity of creating and injecting faults, the method of Software Implemented Fault Injection (SWIFI) is selected. With this mode of fault injection, a tested system can emulate real conditions, in which it is required to function. The Linux operating system was selected due to open source code and ease of manipulation of various parts. Verifications of some tests performed earlier by different authors, on various hardware and software platforms, have been made. The Linux operating system has been modified with an RTAI/LXRT patch in order to gain real-time system functionality. Subsequently, modification of the respective parts of the operating system was made to enable the injection of faults. After the injection of faults, several testing frames were selected to verify and validate the real-time system. Encouraging results have been obtained towards verifying the hard real-time capabilities of Linux RTAI/LXRT System.

Key words: Real-time, Fault injection, RTAI/LXRT, SWIFI.

1. INTRODUCTION

Emulating hardware and software faults is one way of testing fault-tolerant systems, applications and the power of non-fault-tolerant systems. Assessing the reliability of various computer systems involves the study of failures and errors. Because of the features of the faults and the duration of the errors, it is very difficult to identify the cause of computer system failures during their operating time. For this reason, the deliberate introduction of faults occurs in different computer systems. This deliberate introduction of faults is called fault injection. Injecting faults is important because through this method we do the testing of the reliability of computer systems. For our testing purposes, we use real-time systems.

The real-time system can be defined as a system capable of guaranteeing the time requirements of processes under its control [1]. This system should be predictable and with little delay. With a slight delay it means that the system must respond to external events, asynchronous, in a short time. Predictable means that the system has the ability to safely

determine the timing of the task. Usually, the real-time system is a computer control system that manages and coordinates the activities of the controlled system.

It is desirable that timely and non-critical time activities co-exist in real-time systems. Both of these are called tasks and a time-consuming task is called a real-time task.

Usually a real-time task has these requirements or limitations:

• Time constraints. The most common are periodic and non-periodic. A non-weekly task has a deadline to start or end. The periodic task must be repeated once within a period. Usually, sensory processing is periodic, while non-periodic demands arise from dynamic events.

• Resources Requirements. Real-time Tasks may require access to certain resources such as IO (Input / Output) devices, files, databases, etc.

• Request for communication. Tasks should be allowed to communicate with messages.

• Competition constraints. Tasks should be allowed to have competing access to shared resources under a condition that those resources are available at all times.

Assessing the reliability of various computer systems involves the study of failures and errors. Because of the features of the fault and the duration of the error, it is very difficult to identify the cause of computer system failures during their operating time. For this reason, the intentional introduction of errors is employed in different computer systems. This deliberate error is called error injection.

Injecting errors is important in assessing the reliability of computer systems. Through this method we try to:

- Identify the weaknesses of the systems in terms of reliability
- Analyze the system in the presence of errors
- Define the repair mechanisms and the extent of error detection
- Evaluate the effectiveness of fault tolerant mechanisms and performance losses.

2. RELATED WORK

Fault Injection-based Automated Testing (FIAT) [2] is an automated error-based testing that is capable of emulating different distributed system architectures and monitors the response of the tested system. FIAT is used to validate the dependency of the system. Experiments carried out through FIAT include the comprehensive injection of three types of errors into different system locations, including part of code and buffer memory. Types of errors are variations of memory bit errors. Through this testing system it is possible to predict the system level errors. This is very important during the design phase of the system.

Fault Injection and Monitoring Environment (FINE) [3] implements injecting software-related errors caused by the hardware, on UNIX, and following the execution flow and the core variables of the kernels. FINE consists of a bug injector, an experiment monitoring software, a load generator for the tested system, a controller, and some other auxiliary tools.

Reference [3] shows experimentation with the SUN OS 4.1.2 operating system through the FINE system to investigate the propagation of faults and assess the impact of these faults of different types. Models of hardware and software fault propagation have been built. Loss of performance as a result of fault injection was analyzed through Markov's transient analysis. After these experiments, authors have noted that memory and

software failures usually have a very long delay, while failures in the processor cause an immediate crash of the system.

Xception [4, 5] presents a system of error injection which enables accurate and flexible verification and validation of critical computer systems. This Xception validation reaches through the injection of errors. This testing system has a special emphasis on injecting hardware and software faults. It is characterized by a high degree of automation that enables users to plan and implement easier fault injection experiments. It features a relatively advanced GUI (graphical user interface) through which errors can be injected, experimentation and analysis of results. Data to be placed on an SQL database is used to manage and automate experiments. Through Xception, errors can be injected into the kernel and user code. Within a distributed system it is possible to inject faults into different processors. Xception is used in different systems, ranging from Online Transaction Processing Systems (OLTP) [6] to Spatial Application Systems [7].

Propagation Analysis Environment (PROPANE) [8] is a tool for experimenting error injection into personal computer software (desktop). It was designed at the University of Gothenburg in Sweden. PROPANE allows software errors to be injected (making changes to the source code) and data errors (making manipulations with variables and memory contents). Supports different types of errors and enables injecting errors defined by the user of this test tool.

DEPEND [9] is a simulation system through which computer architectures are evaluated. This is accomplished by simulating a tolerant system based on triple modular redundancy (TMR) tandem based on UNIX operating system. Through this simulation, system response testing is performed on almost random errors caused by latent and correlative failures. In [10], a comparison was made between Linux kernels 2.4, 2.4 kernels with RTAI and RTAI / LXRT installed, and 2.6 modified kernels with precaution patch, using three real-time applications in two cases when the operating system is a bit charged and when it is too busy.

The results show that in the 100 Hz controller system, Linux 2.4 kernel with RTAI and RTAI / LXRT meets the requirements of delays in 100 percent of the time. If we apply this application to a 600 Hz control system, 2.6 modified patch kernel for prevention would supplement them. The experiment carried out in [11] has enabled the application of both patches (prevention and low latency) in the same kernel. It was noted that the obtained results were better than applying patches individually.

UMLinux [12] represents the User Mode of Linux that can be used for realistic experiments with regard to error injection. UMLinux simulates a Linux system. The simulated hardware can be forced to fail through the injection of errors and then become a reaction to the Linux operating system and the applications running on those machines.

To be as close to reality, UMLinux is implemented by protecting kernel memory and the entire virtual machine (operating system with all processes) is executed as a single process in the computer system. The whole system works so that first the virtual server system is activated and then the injecting of the errors is done through the fault injector which for this purpose is configured immediately after the virtual server system is upgraded. Fault injecting is done on virtual hardware. In this experiment, authors [12] have made bit flip in processor registers and in main memory.

VERIFY [13] presents a technique for evaluating digital systems based on the VERIFY (VHDL-based Evaluation of Reliability by Injecting Faults efficientlY) injector. It practically means a software tool designed to describe the response of hardware

components in case of faults. This is accomplished by expanding the VHDL (Very High Speed Integrated Circuits Hardware Description Language) with error injection signals along with the frequency of their use.

Verification of embedded real-time systems was implemented in [14, 15] with a framework for the verification of safety and timing properties of digital embedded real-time systems, based on models built with SystemC, using timed automata and the UPPAAL model checker.

3. EXPERIMENTS

As mentioned earlier, the main purpose of the testing was to validate the real-time operating system so as to confirm the assertion of its creators that indeed, regardless of the number and type of processes that can be executed at one and the same time Real-time system, this system will not be blocked and will continue to function normally.

To do this, preparations have been made to create a suitable test system. This system should include at least one computer (preferably PC architecture), a real-time Linux operating system. Validating the real-time system is implemented in the PC architecture (Dell Inspiron 8600) with these features: Intel Mobile Processor 1.4 GHz, 512 MB RAM, and 32 MB standard graphics card and 40 Gb hard drive. The operating system used was Red Hat Linux v9.0 and Suse Linux 10.0.

This operating system as such was not ready to apply the validation process. The main reason is that, as such, it is not a real-time operating system. For this reason, an operating system patch has been applied (RTAI-3.1r2 patch has been applied on Red Hat Linux v9.0 and RTAI-3.6 patch has been applied on Suse Linux 10.0). With this procedure, the operating systems have gained additional functionalities in terms of real time.

Kernel version 2.6.9 was used (kernel version that was used with operating system was 2.4.20-18). This kernel version was chosen as the most appropriate because of the RTAI version (RTAI-3.1r2) which was more stable and matched to the kernel version used. After installing the new kernel, the RTAI patch was installed and finally, after the system reset, the real-time operating system was tested.

To obtain a timing correctness behaviour, it is necessary to make some changes in the kernel sources, i.e. in the interrupt handling and scheduling policies. In this way, we can have a real time platform, with low latency and high predicatbility requirements, within full non real time Linux environment (access to TCP/IP, graphical display and windowing systems, file and data base systems, etc.).

The difference between the original Linux Operating System and the modified one are the added features of an industrial real time operating system. It consists basically of an interrupt dispatcher: RTAI mainly traps the peripherals interrupts and if necessary re-routes them to Linux. It is not an intrusive modification of the kernel; it uses the concept of HAL (hardware abstraction layer) to get information from Linux and to trap some fundamental functions.

Functions affected by the fault injection are:

malloc, *realloc* and *memalign.*

The period of disrupting *malloc* is done with the following variables:
FAILMALLOC_PROBABILITY, FAILMALLOC_INTERVAL, and FAILMALLOC_TIMES.

The probability of failure of memory allocation at the rate of 80% is achieved with the command:

Env LD_PRELOAD=libfailmalloc.so FAILMALLOC_PROBABILITY=0.8 1s

With *FAILMALLOC_SPACE* we have tweaked the size of empty space where memory can be reserved securely.

With the following commands we injected faults:

fail_make_request=100,10,-1,0

echo 1 > /sys/mlock/hda/hda8/make-it-fail

Number 100 in the command fail_make_request=100,10,-1,0 determines in percentage the frequency of the appearance of the failure.

Number 10 specifies the interval of the failures. Number -1 specifies how often the failures occur, at most. Number 0 specifies the size of the free space where disc I/O (Input Output) can be placed.

3.1. Performed Tests

The first three tests in this chapter are pre-prepared and the main purpose of using these three tests was to verify correct installation of the RTAI / LXRT patch on the Linux operating system.

Testing was performed using examples installed with RTAI *(/usr/realtime/testsuite/kern/* or */usr/realtime/testsuite/user/*). In the kernel or user folder we can find 3 examples of RTAI patch testing.

If we use the latency option, after executing the *./run* command in the latency subfolder, we will get some results (see sections B to F) to verify the operation of the real-time operating system. These tests are implemented in the one-shot timer mode rather than in the periodic timer mode to simplify testing and achieve greater flexibility because the task can be reset after the timer expires. The added flexibility of one-timer mode has a negative impact on performance [16]. On the other hand, if we analyze the testing of the worst case delays, then this mode is preferred. The fourth test is used to measure the delays that arise after executing tasks. This delay is known by the term jitter.

In the fifth test error injection was performed in one of the two tasks that were executed in parallel and then it was verified that the injected error did not affect the performance of the other task.

3.1.1. Test 1: Latency

The latency test implements the measurement of the real-time delay (the planned task time minus the activation time of the task) in nanoseconds.

The results, presented in Table 1, are obtained after 8 repeats of the test and include minimum, maximum and average values both in user - space and kernel - space as well as for two operating system states: not loaded and loaded.

Table 1. Delays in the two states of the operating system

State	*Not loaded*		*Loaded*	
User	*Minimum*	*-1204 ns*	*Minimum*	*166 ns*
	Maximum	*743079 ns*	*Maximum*	*866056 ns*
	Average	*9825 ns*	*Average*	*19797 ns*
Kernel	*Minimum*	*-3720 ns*	*Minimum*	*-1966 ns*
	Maximum	*1050981 ns*	*Maximum*	*771434 ns*
	Average	*6572*	*Average*	*8895 ns*

The state of the operating system immediately after the activation of the test is considered untargeted. The operating state of the operating system is reached by using the command cat /dev/hda1> / dev/null. By activating this command, all files in the hda1 partition are read and then a special file is created that rejects all the data listed therein.

From the above we notice that the average time lag in user-space was twice as big as that in kernel-space.

3.1.2. Test 2: Pre-emption

This test measures the preemption of the fastest task to the slower task. The minimum, maximum, and average are the jitter values (delay of the task execution reaction) expressed in nanoseconds. The "fast jitter" and "slow jitter" values show delay values for the fastest and the slowest tasks. In Table 2 we notice such delays are more expressed in user-space. The number of tests carried out in this experiment was 65.

Table 2. Values of the preemption test

State	*Not loaded*		*Loaded*	
User	*Minimum*	*1 µs*	*Minimum*	*0 µs*
	Maximum	*577 µs*	*Maximum*	*654 µs*
	Average	*11 µs*	*Average*	*8 µs*
	Fast jitter	*257 µs*	*Fast jitter*	*269 µs*
	Slow jitter	*262 µs*	*Slow jitter*	*256 µs*
Kernel	*Minimum*	*-99 µs*	*Minimum*	*-99 µs*
	Maximum	*261 µs*	*Maximum*	*791 µs*
	Average	*0 µs*	*Average*	*0 µs*
	Fast jitter	*199 µs*	*Fast jitter*	*199 µs*
	Slow jitter	*459 µs*	*Slow jitter*	*1199 µs*

3.1.3. Test 3: Switches

Measures the switching time between multiple signal/wait and suspend/resume calls. Similar to the first 2 tests, user-space takes more time but is relatively close to real-time requirements. The number of tests carried out in this experiment was 80. The results of the tests are shown in Table III.

This is the first part of the practical work (preparation of the test system).

In the next phase, the selection of the test mode was done. So, the ways in which the system can be "attacked" and the stability of the system are checked. One possibility is to execute a task within which there is an "endless loop" and parallel to this execution, to execute another common task and time measurement (usually at this time expressed in nanoseconds).

Table 3. Values of the Switches Test

State	***20 tasks***	
User	*Time*	*154 ms*
	Switches	*40.000*
	Switching time	*3853 ns*
Kernel	*Time*	*113 ms*
	Switches	*40.000*
	Switching time	*1420 ns*

3.1.4. Test 4: Task Scheduling

During this test, the testing of the minimum scheduling time that could be achieved in the operating system was done. In RTAI/ LXRT environment, this time reached 455 nanoseconds. Afterwards, jitter testing (unpredictable delay) of the scheduling was done by charging the processor from 0 to 97% of the processor cycles. This is noted in Fig. 1 and Fig. 2. The number of tests performed in this experiment was 25.

From the figures we notice that the change of the jitter is very small, which also shows a confirmation of the stability of the system. Standard deviation band during execution of the same test is less than 1 microsecond.

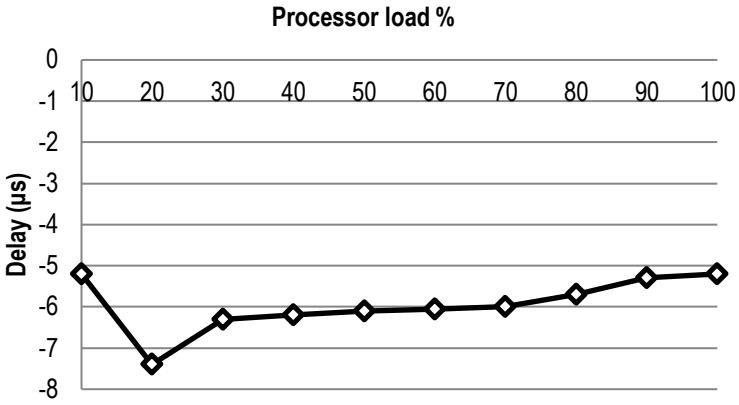

Fig. 1. Average values of the jitter in task scheduling test

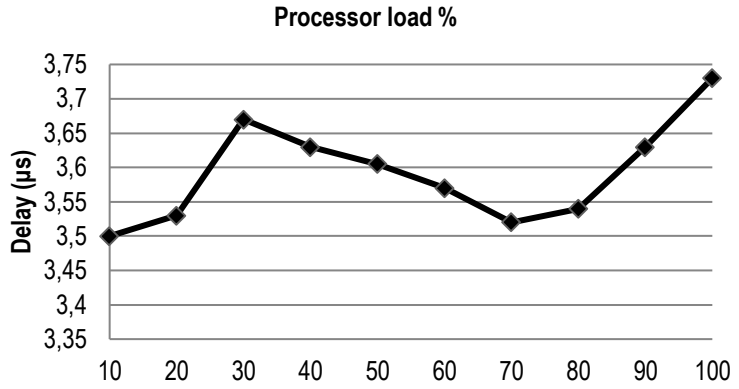

Fig. 2. Standard deviation of the jitter in task scheduling test

3.1.5. Test 5: Parallel Execution of Two Tasks

In this test, two tasks are executed in parallel, one with higher priority and the other with lowest priority.

The number of tests carried out in this experiment was 2.

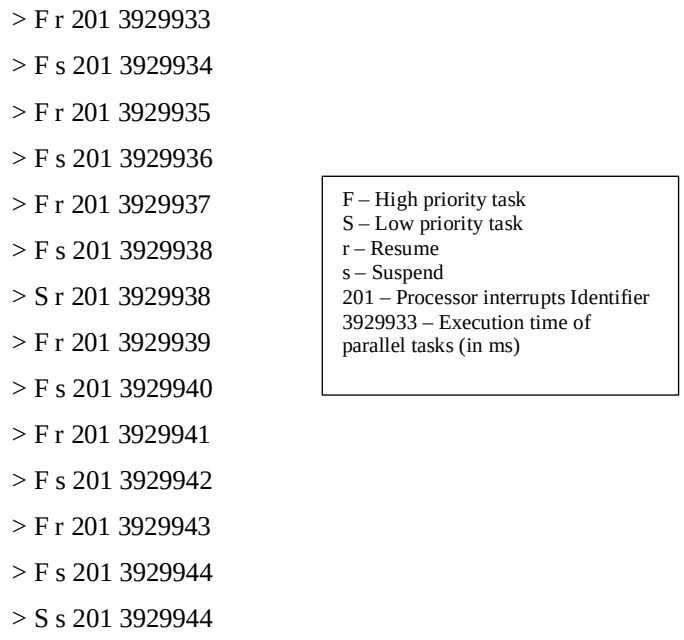

Fig. 3. Results of the execution of two tasks in parallel

As in the first case, when neither of them had a fault injection, when in the lowest priority task, a fault was injected (endless loop), the result of their execution was normal, as seen in Fig. 3.

The result shows that the highest priority task is executed six times faster than the lowest priority task (as expected).

4. CONCLUSION

With RTAI/LXRT, it becomes feasible to develop hard real-time applications similar to normal Linux applications. Easier and more development options are benefits of developing in user-space with minimal performance penalties.

The test cases provided with RTAI were run and modified to substantiate the claim that hard real-time applications can be successfully executed without being interfered by other applications in user-space. In general, the RTAI-Linux operating system patched with RTAI (RATI-3.1r2 and RTAI-3.6) produced consistent and expected results. RTAI was stable even at high CPU loads.

Installing and patching the operating system was relatively easy, once the process of building and installing RTAI was understood. While choice of an operating system depends on many factors, RTAI would be adequate for soft-real-time systems.

For hard-real-time requirements, a thorough evaluation of the operating system and the operational environment should be undertaken to assure that the operating system can meet the requirements. The tests that were made in this work indicate that RTAI is likely to meet hard-real-time requirements, depending on the nature of the timing requirements are.

In this work a new way of validating the RTAI/LXRT system was created by previously inserting faults into the system and then the testing was conducted through the modification of the operating system. With this, a validation of RTAI/LXRT was achieved. In conclusion to this, we can say that the RTAI/LXRT system accomplishes the goals set out for a hard- real-time system.

Based on the tests that were implemented in the RTAI/LXRT system, performance was not degraded during the duration of these tests.

Every run generated a high success rate (over 98%). The very small number of failures that occurred was as a result of concurrent executions of tests. These concurrent tests are designed to overload system resources.

Overall, the Linux kernel, with RTAI/LXRT patches, and other core components (device drivers, file systems, IPC, memory management, etc.) operated consistently and competed all the expected durations of runs with zero critical failures.

For a better validation of real-time systems a continuous development of software fault injection (hardware or software faults), especially in situations where the CPU load exceeds 95%, is recommended.

This way of using the testing framework with these fault injectors can be a start for developing other fault injectors and to continuously validate real-time systems.

REFERENCES

[1] H. Kopetz - "Real-Time Systems: Design Principles For Distributed Embedded Applications". *Kluwer Academic Publishers*, 1997.

[2] J. Barton, E.Czek, Z. Segall, and D. Siewoirek, "Fault Injection Experiments using FIAT", *IEEE Transactions on Computers*, 39(4), 1990.

[3] W. I. Kao, R.K. Iyer, and D. Tang, "FINE: A fault injection and monitoring Environment for tracing the UNIX system behavior under faults". *IEEE Transactions on Software Engineering*, 19(11):1105-1118, November 1993.

[4] J. Carreira, H. Madeira, and J. G. Silva, "Xception: A Technique for the Experimental Evaluation of Dependability in Modern Computers", *IEEE Transactions on Software Engineering*, 24(2):125-136, February 1998.

[5] R. Maia, L. Enriques, D. Costa, and H. Madeira, "Exception – Enhanced Automated Fault-Injection Environment", *Fast Abstract Session*, June 2002.

[6] D. Costa, T. Rilho, and H. Madeira, "Joint Evaluation of Performance and Robustness of a COTS DBMS Through Fault-Injection", *IEEE/IFIP Dependable Systems and Networks Conference* – DSN, New York, USA, June 2000.

[7] H. Medeira, R. R. Some, F. Moreira, D. Costa, and D. Rennels, "Evaluation of a COTS System for Space Applications", *International Conference on Dependable Systems and Networks*, 2002 (DSN), pages 325-330, Washington D.C., USA, June 2002.

[8] M. Hiller, A. Jhumka, and N. Suri, "PROPANE: An Environment for Examinig the Propagation of Errors in Software". *International Symposium on Software Testing and Analysis*, Rome, Italy, 2002.

[9] K.K. Goswami, "DEPEND: a simulation-based environment for system level dependability analysis", Computers, IEEE Transactions on. Computer Volume 46, Issue 1, (60-74), January 1997.

[10] P. Laurich, "A Comparison of hard real-time Linux alternatives". *Linux Journal*, December 2004.

[11] C. Williams, "Linux Scheduler Latency". *Linux Devices*, March 2002.

[12] H. Höxer, K. Buchacker, and V. Sieh, "UMLinux - A Tool for Testing a Linux System's Fault Tolerance". *Institut für Informatik* 3, Erlangen, Germany, 2002.

[13] V. Sieh, O. Tschäche, and F. Balbach "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions". *Proceedings of the 27th International Symposium on Fault-Tolerant Computing*, Washington, USA, 1997.

[14] P. Herber and S. Glesner, "Verification of Embedded Real-time Systems", Formal Modeling and Verification of Cyber-Physical Systems pp 1-25. *1st International Summer School on Methods and Tools for the Design of Digital Systems*, Bremen, Germany, September 2015.

[15] A. Cimatti, I. Narasamdya, M. Roveri, Software model checking systemC. *IEEE Transactions on CAD of Integrated Circuits and Systems* 32(5), 774–787, 2013.

[16] P. Mantegazza, E. Bianchi, and L. Dozio, "DIAPM RTAI Programming Guide 1.0". *Dipartimento di Ingeneria Aerospaziale*, Politecnico di Milano, 2000.

Information about the authors:

Arsim Susuri, PhD, received a PhD degree in Computer Sciences from South East European University, Tetovo, Macedonia in 2017. He is currently lecturer at the University of Prizren, Kosovo, Faculty of Computer Sciences. His research focuses on Real-time Systems and Machine Learning.

Mentor Hamiti, PhD, received a PhD degree in Computer Sciences from South East European University, Tetovo, Macedonia in 2010. He is currently associate professor in Computer Sciences at the South East European University, Faculty of Contemporary Sciences and Technologies. His research focuses on Programming Languages and Technologies.

Besnik Selimi, PhD, received a PhD degree in Computer Science from Joseph Fourier University, Grenoble, France, in 2009. He is currently associate professor in Computer Sciences at the South East European University. His research interests evolve around Software Engineering, Software Testing, Service Oriented Architectures and Security. **Artan Luma, PhD**, received a PhD degree in Computer Sciences from South East European University, Tetovo, Macedonia, in 2010. He is currently associate professor in Computer Science and Engineering at the East European University. His current research interests are in cryptography, security, semantic web, etc.

Manuscript received on 1 October 2017