Secure Web Server System Resources Utilization

Liberios Vokorokos, Anton Baláž, Norbert Ádám

Technical University of Košice, Letná 9, 042 00 Košice, Slovakia,
liberios.vokorokos@tuke.sk, anton.balaz@tuke.sk, norbert.adam@tuke.sk

Abstract: One of the main goals of an operating system is the efficient usage of system resources and their management. Modern systems offer complex mechanisms for the management of these resources. Their system-wide use may be separated into multiple layers – from the lowest (the operating system kernel) to the highest (various virtualization techniques). Somewhere in the middle are the applications consuming these system resources. An example of such an application is the Apache web server. It implements some of the resource management mechanisms at application level. However, the system contains elements, the management of which is not possible at application level or its implementation has a significant overhead and may negatively affect the performance and secure usage of the server. The aim of the study is to create a more efficient resource management mechanism at the level of the Apache web server. The proposed architecture allows the treatment of possible security threats, such as DoS attacks, makes the use of system resources more efficient, thus – indirectly – it also addresses the environmental impacts of data centers. The reason for selecting the Apache web server as the referential implementation of the resource usage improvement are its market share and its open source nature, which renders making amendments to the server simple.

Keywords: security; system resources; management; Apache; cgroups; CGI

1 Introduction

System resource management is a topic often related to operating system mechanisms. Solving system resource management problems is therefore left over to the authors of operating systems or architects. Even though it is required – to a certain extent – that the operating system manages most of the system resources, there are cases, when the task of system resource management has to be left over to the application, or to ensure that the application increases the level of use of the allocated resources.

With today's web applications at hand, web servers often have to deal with enormous loads – millions of web content generation requests coming from their clients. Generating content has a price in terms of memory, computation time,

power, CPU time, etc., especially if one has to deal with extreme amount of requests. By improving resource management we may improve security, make the use of the resources more efficient and – in the end – decrease the power costs by improving system usage efficiency.

We know of experiments to improve the resource usage of web servers used in in-house solutions, producing certain results [1], nevertheless there is still room for improvement – both in the field of implementation or the results themselves. With the advent of new technologies, new approaches are being developed to improve and make the current resource management solutions more efficient.

This work describes the approaches in use, presents the new technologies available in the field of resource management, and – most importantly – implements an improvement in this field and proposes the implementation of such an extension of the Apache web server.

2 Efficient Use of System Resources

In modern operating systems, the use of system resources is considered to be a more or less solved problem. Resource management is being done efficiently at the lowest OS level by the kernel, which may be configured to a certain extent, depending on the system in use.

Recently, great effort has been put into higher level system resource management, mainly to separate programs executed at application level and the operating system itself. This field is represented by various virtualization technologies – for their application in modern systems, seen in the work G. Banga et al.

2.1 Virtualization and Separation

Virtualization is a method of executing various virtual environments (virtual machines) on a single physical machine.

The virtual environment is where the programs are executed; it behaves as a separate entity and represents a real computer system. The individual environments are mutually isolated and thus have no knowledge of the existence of similar environments, if multiple virtual environments are being executed simultaneously on a single physical machine. However, this kind of isolation requires full resource management, therefore in the text below we have described the approaches used in virtualization and resource management.

Virtualization is based on the principle of having virtual machine monitor software managing the allocation of computer resources to the respective virtual environments. This monitor is the abstraction layer between the hardware – the physical machine – and the given virtual environment. There are multiple types of

virtualization, depending on the executed virtual environment and the implementation of this monitor.

Kolyshkin described three basic virtualization principles, differing in the methods and conditions of execution of the virtual environments [1]. These are:

- Full virtualization or emulation
- Para-virtualization
- OS-level virtualization

Each of these approaches has its advantages and disadvantages, related to the execution performance of the given virtual environment. The respective virtualization approaches are thus compromises between performance and modifications made to the executed system.

2.2 OS-level Virtualization

Virtual environments using this approach are called containers – they represent the execution environment of the operating system [5]. Operating systems running on a single kernel are being executed in the individual containers. The resources are allocated to such environments upon creation and they are often altered dynamically during the execution.

Applications, as well as the individual environments behave as individual units, which may be isolated (seen from the other containers) [10]. Such isolation is not performed by virtualization of the whole operating system, just the mechanisms offered by the operating system itself (the kernel), which executes all containers. These mechanisms include process identifiers (PIDs), user identifiers (UIDs), shared memory, etc. Contexts and filters are created to perform the separation of the respective containers. Contexts represent the separation of the identifiers used in the said mechanisms: each container has its own context of identifiers, relevant only for it. On the other side, filters control the access of the respective containers to the kernel objects by controlling the privileges of the individual containers.

There are multiple container implementations available in operating systems running a Linux kernel. The most widely known is OpenVZ Linux VServer, while the newest addition to these is a solution called LXC Linux containers. They differ is the way they implement the container and resource management levels. To prevent attacks between the environments, the system must distribute the resources among the respective virtual environments securely.

These containers often implement processor time management in two levels [11]. One level is the CPU scheduler of the operating system. The second level is the scheduler of the respective implementation, which is either part of the amendments applied to the system or a separate entity, similar to the virtual machine monitor. CPU time scheduling is then done using known algorithms.

The remaining resources, such as physical memory, disk space, etc. may be managed in the respective containers separately by using the appropriate memory management mechanisms [16] or a resource driver may be introduced, such as a memory controller [17], which can limit the amount of memory available to certain application groups (e.g. to an application in the context of a specific container).

Interesting is the implementation of system resource limitations in LXC Linux containers. It uses various novel attributes of the Linux kernel, even a mechanism called Control Groups [9]. This allows setting constraints using control groups. The running processes (i.e. tasks) are then allocated to the respective groups as needed. The system kernel will then allocate the system resources in accordance with the rules specified in the respective groups.

From this point, the respective groups (cgroups) are treated as standard Linux processes. Thus, they form a certain hierarchy and the descendant groups in the hierarchy inherit the attributes, in this case the constraints of the parent groups. These group constraints are implemented by means of so-called subsystems. These are linked to the directory structure of the system. The whole directory structure of the link represents a hierarchy of constraints – it contains files of the constraint rule applicable to the individual subsystems. The basic subsystems, thus also the system resources, influenced by the cgroups are the following:

- CPU – controlling processor access of the tasks by using the CPU scheduler,
- CPUACCT – generates messages concerning the usage of CPU resources,
- CPUSET – assigns the individual processors and memory nodes to tasks,
- MEMORY – manages task memory constraints,
- BLKIO – manages I/O device data transfer constraints,
- NET_CLS – marks packets with identifiers for the network traffic controller,
- NET_PRIO – sets the network traffic priority in case of the network interfaces.

The newly created group is a directory in each of the subsystems in the directory structure. It also contains the files of the respective subsystems required to define new constraints, as well as the file with the identifiers of running processes belonging to the particular group.

Cgroups may be used even beyond the scope of virtualization tools to separate applications and thus this technique is an appropriate candidate for the implementation of the solution – our improvement to web server resource management.

The advantage of virtualization at the level of the operating system is the performance, comparable to that of native code execution. By removing the virtualization layer – the virtual machine monitor – the execution costs decrease.

The separation of the respective virtual environments is not as significant as with emulation or paravirtualization, but it is sufficient enough. The security of the system as such is ensured by the separation of the important applications into separate environments. Dynamic resource management allows runtime application migration.

A disadvantage may be the requirement of using the same kernel with the virtualized operating systems. Thus, in case of Linux only Linux systems may be virtualized.

3 The Solution Proposal

The proposed security improvement in server resource usage lies in ensuring a more strict allocation of the individual system resources to the processes executing the programs generating dynamic web server content. These are especially the programs using the Common Gateway Interface (CGI) protocol. Stricter system resource allocation allows more efficient system operation by defining system resource consumption constraints and thus achieves the required service quality. Strict allocation of processor time and memory to the respective processes helps solve possible Denial of Service (DoS) attacks, limit the impact of badly written web applications by using well-defined constraints to prevent any request from consuming all available system resources.

3.1 Design of Architecture

The goal is to limit the respective system resources as required, during the assignment of CGI programs to the respective processes or immediately after it. The reason of preferring such a direct assignment of resource constraints to processes before allocating the resources during process creation is the fact that the same process may be used for the sequential execution of multiple independent programs, using a process pool. In such a case the resource constraint of a single program would remain valid for other programs, if they were executed in the same process.

The respective CGI programs may be identified by the UID of the executed program at the time of system resource constraint allocation. Normally, the UID of the executed program (web) is identical to the one of the executed web server. The UID is altered using the suExec module. Using UIDs to identify the programs allows resource constraints to be assigned according to the user (web page), the program was executed by.

The proposed improvement is implemented as a server module. Due to the modular architecture of the web server, this module is connected to the web server architecture and the other modules, required for the correct function of this

module. This provides the following: a connection between the module and server functionalities; a mechanism to define, process and store the module configuration; the possibility to initialize the module upon server start-up. The architecture of the resulting solution is shown in Figure 1. The mod_cgrp module is the outcome of this work.

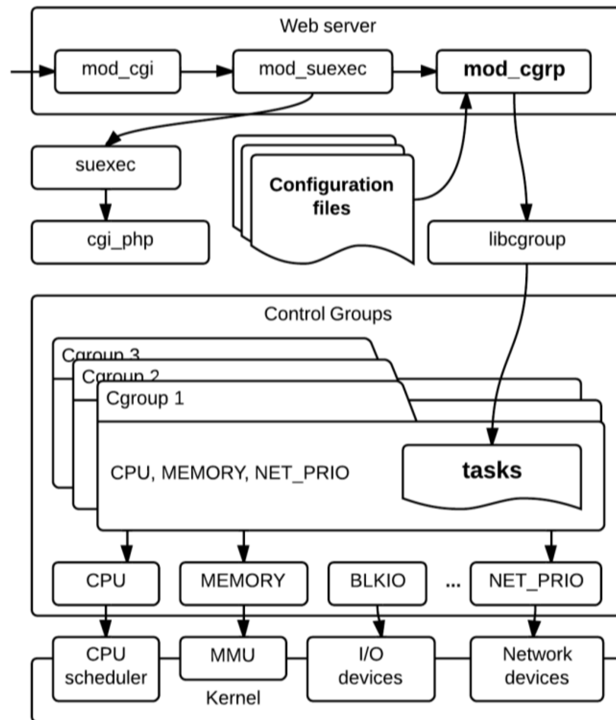

Figure 1
System architecture

Constraining system resources is provided by a system mechanism called Control Groups. This mechanism is supported by the kernel itself. Resource management is implemented by the control groups organized in the file system. The kernel allocates resources to the respective groups according to the group attribute settings.

The processes are identified by means of the configuration files. These contain the data required for the identification of the respective processes, as well as the processes, which have to be assigned to specific cgroups. These data are stored in data pair records. The first member of the pair is the PID of the process assigned to the constraint group. The other member of the pair is the identifier of the cgroup. Only a single constraint group may be assigned to a process having a specific PID.

3.2 Principles of Resource Constraint

Designed module assigns newly created processes to groups with limited privileges to consume specific system resources, as shown in Figure 2.

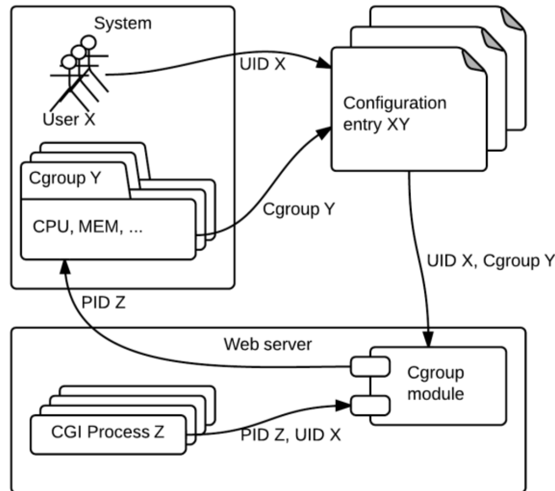

Figure 2

System resource assignment

The execution of the module starts with loading the configuration. Configuration processing is executed automatically by the web server upon its initialization, along with the other modules. Next, the UID of the executed process is retrieved. The server checks, if there is an entry in the configuration for the specified identifier. If the identifier in question is not present in any of the entries, the execution terminates. Otherwise, the ID of the process running the CGI program is retrieved. The module checks the existence of the constraint group stated in the configuration and as in case of the configuration itself, if the existence is not confirmed, the execution terminates and an error message is written to the log file. If the group exists, the retrieved process identifier is assigned to the particular group and the execution terminates with a message stating successful assignment of the process to the group with the given constraints.

4 Experimental Validation

To evaluate the reliability and functionality of the proposed and implemented solution, we have performed some measurements. These measurements focused on the performance and the correct server functionality during the use of the proposed module. The measurements were performed with the tools provided by the operating system and the web server itself.

The computer system used to perform the measurements was a HW system with the following parameters:

- Intel i3 processor working at 2.20 GHz,
- 4096 MB RAM system memory,
- 500GB hard drive spinning at 5400 RPM.

As to the software, the system was running a GNU/Linux Fedora 20 operating system with a 3.13.7 Linux kernel. The web server version number was 2.4.6.

The results of all measurements were recorded by redirecting the standard output of the *top* and Apache Benchmark (*ab*) tools to separate files. The stress testing of the respective resources was performed using a series of php scripts allocating small or large chunks of data (depending on the types of the measured resources) in cycles.

4.1 Module Performance

The following measurements focused on comparing the web server performance with various amounts of requests executed in parallel. Test performed with the *ab* program. The output of the program was – in addition to other things – information on the transfer time and the average time of execution of a single request. We have executed multiple measurements, with varying amounts of parallel requests. Each of these measurements was performed twice, with the module active and inactive. This way we could test the effect of the module on the performance of the web server.

Figure 3 and Figure 4 show that the presence of the module does not significantly impact the speed of request processing. Differences in server performance having the module inactive or active are negligible with low amounts of requests and increase along with the increasing the amount of the requests. The overall evaluation of the presence of the module when responding to the requests aimed at simple HTML or more complex CGI programs is positive. The module has no significant impact on the web server itself.

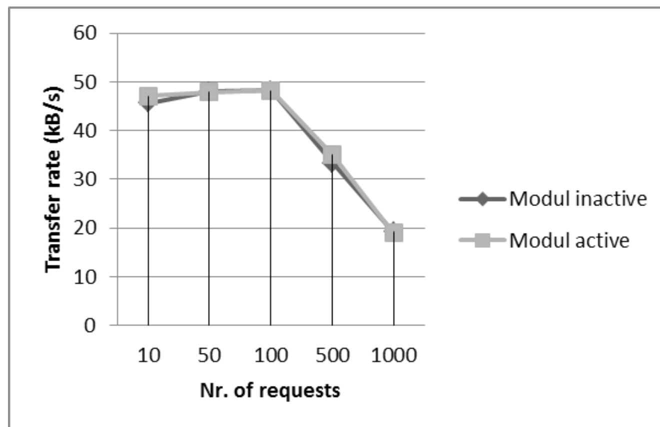

Figure 3
Transfer speed

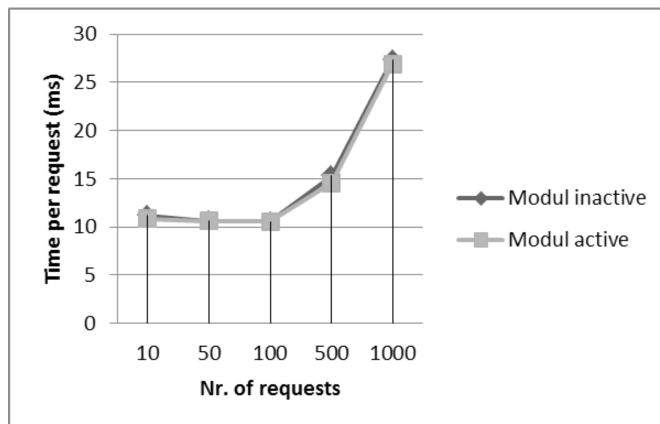

Figure 4
The average time spent by the execution of a single request

4.2 High System Utilization

In this section, we have focused the measurement efforts on comparing the activities of the web server with the module inactive and active, using requests of programs stressing selected system resources. Each measurement was performed individually, with the same program, specifically written for the given system resource. To gather the results we have used the *ab* toolset; we used the *top* program monitoring various system resources to create a specific number of requests.

The subsequent two measurements are aimed at CPU load. The created cgroup has been crafted to prevent the CPU load values from exceeding 50%. The executed program is a cycle, in which the number PI is allocated 3 million times as part of the Foo class. The chart in Figure 5 (with the module being inactive) shows the CPU load – three of the five processes cause CPU loads surpassing 50% during the execution.

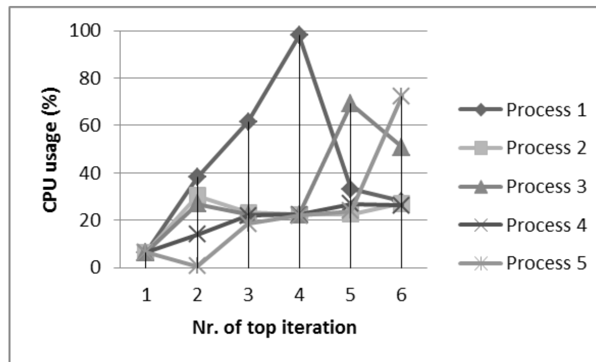

Figure 5
CPU time usage, with the module being inactive

Figure 6 (with the module being active) shows that none of the processes surpassed the 50% CPU load constraint. Each process utilizes approximately the same CPU time during execution and the remaining parts of the system do not degrade due to eventually malicious code.

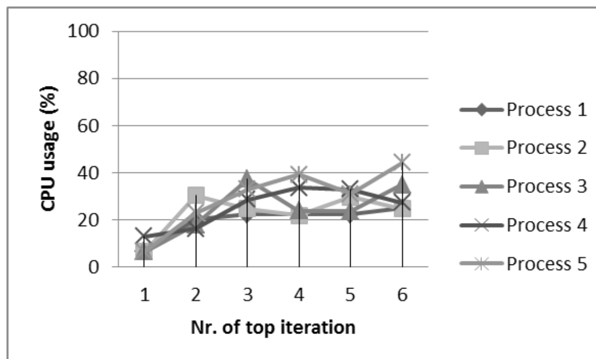

Figure 6
CPU time usage, with the module being active

The next measurement focused again on the CPU as a system resource. However, unlike the previous case focused on CPU time, this time we focused our measurements on one of the CPUs of a multiprocessor system. Again, we executed the same PHP program aimed at stress testing the system CPU. The test system contained multiple processors.

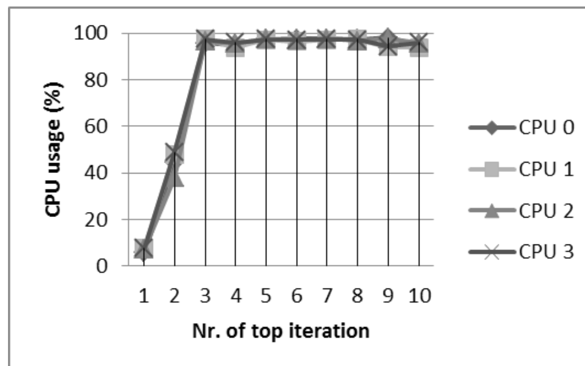

Figure 7

Multi-core CPU load, with the module being inactive

Figure 7 (with the module being inactive) shows the load of all processors in the system from the start of the execution of the respective CGI servers on the server. In this measurement, the server did not have the module functionality activated, so all four processors were loaded to almost 100%.

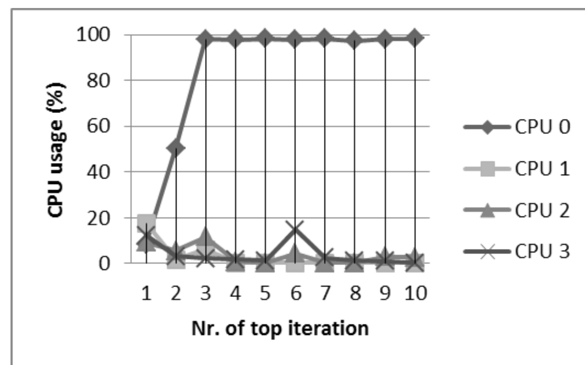

Figure 8

Multi-core CPU load, with the module being active

In the next measurement, the module was active and the cgroup was set up to allow the respective CGI processes of the given user (web) only to the CPU0 processor. As it is shown in the chart in Figure 8 (with the module being active), only the CPU0 processor was running. The other processors did not run the above malware; therefore their load did not exceed 20%.

The last server functionality measurement focused on memory as a system resource. Again, a special PHP script, representing malware was written, but this time it stressed the system memory. The program works again in cycles, but instead of a small number- PI, it allocates a number equal to the iteration multiplied by 100 and inserts it into a dynamic array.

The PHP interpreter contains mechanisms to limit the amount of memory allocated to the respective processes. This is 128 MB by default. To demonstrate the functionality and flexibility of the module configuration, we have altered this setting to 2 GB. There are plenty of languages – in addition to PHP – which may be used to run CGI programs and have no such limitation implemented.

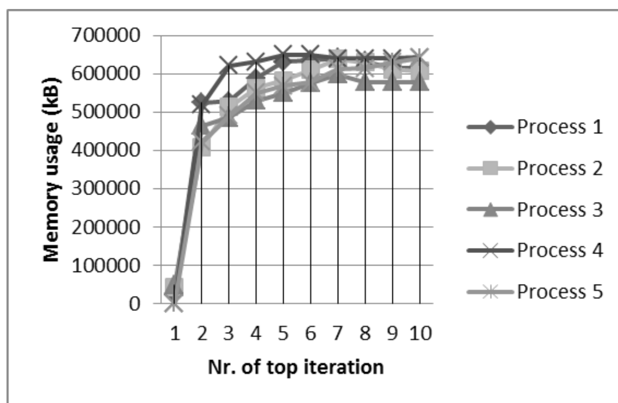

Figure 9
Memory usage, with the module being inactive

Figure 9 (with the module being inactive) shows the memory usage after loading the malicious program. The chart contains the first 10 iterations of the log of the *top* program. The chart shows that after the first iteration more than 600 MB of the memory of all processes was allocated. All 5 processes used all available system memory. If there are many client-side requests, the memory resources run out fast and the service is going to be unavailable.

In the second measurement focused on memory this module was active. A cgroup was set up to limit the maximum memory allocated to the process in this group. The constraint set up was 10 MB. The chart in Figure 10 showed the results of this measurement. As it is evident, none of the processes has produced a load greater than 10 MB to the physical memory of the system. The load was approx. 5 MB/process. The constraint was working well and we may state that this module setting is more flexible, since it may be used even for programs written in other languages than PHP. Moreover, we may define different constraints for different users (web sites).

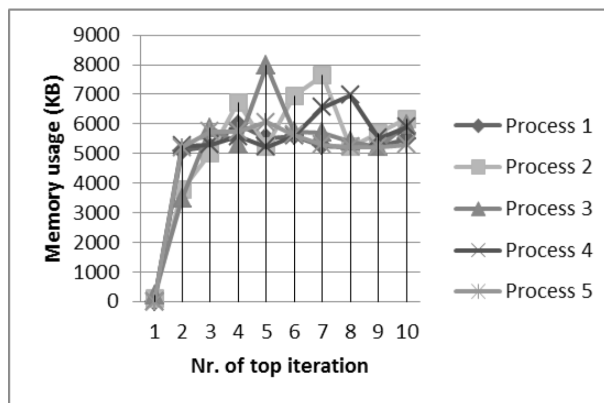

Figure 10
Memory usage, with the module being inactive

Conclusions

The output of this article is an efficient solution of using system resources. The solution designed and implemented at the level of a web server, allows limiting key system resources, such as CPU time and memory.

The system was subject of performance and functionality tests of the module with various configurations. The module as such does not degrade the performance too much to have a significant impact on the execution of the web server. Our tests showed that the module does not have any significant performance impact and all results achieved without the module stopped were approximately the same as those achieved with the module running.

As far as functionality tests are concerned, some attributes in the implementation have already been implemented but are not flexible enough or they do not cover all cases, in which insufficient resource management could affect the performance and stability of the whole system.

In addition to the tested resource constraints, cgroups provide significantly broader system resource configuration possibilities – other than CPU time and memory – therefore they provide a far better way of system resource management and a more stable environment for applications running on such a system, which provides further development possibilities in the proposed solution.

We may also state that the limitations, achievable by means of this resource management facility, are a more coherent and transparent configuration system applicable to various system resources, managed at a single place. It prevents configuration errors, eventual stability problems or various security threats to the system.

The work presented here, allows data centers to use their system resources more efficiently and securely, taking the costs related to power consumption into

account – being an important attribute of data centers, monitored not only to decrease operating costs but also the environmental impact.

Acknowledgements

This work was supported by the Slovak Research and Development Agency under the contract No. APVV-0008-10 and project KEGA 008TUKE-4/2013: Microlearning environment for education of information security specialists.

References

[1] Kolyshkin Kirill: Virtualization in Linux, OpenVZ Technical Report, Kirkland 2006

[2] Banga Gaurav, Druschel Peter, Mogul Jeffrey: Resource Containers: A New Facility for Resource Management in Server Systems. Proceedings of the Third Symposium on Operating Systems Design and Implementation (OSDI), New Orleans 1999

[3] Soltesz Stephen et al.: Container-based Operating System Virtualization: A Scalable, High-Performance Alternative to Hypervisors, ACM SIGOPS Operating Systems Review (EuroSys), Vol. 41, No. 3, New York 2007, pp. 275-287

[4] Singh Balbir, Srinivasan Vaidyanathan: Containers: Challenges with the Memory Resource Controller and its Performance, Proceedings of the Ottawa Linux Symposium, Vol. 2, Ottawa 2007, pp. 209-222

[5] Tomášek Martin: Computational Environment of Software Agents, Acta Polytechnica Hungarica, Vol. 5, No. 2, Budapest 2008, pp. 31-41

[6] Totok Alexander, Karamcheti Vijay: Optimizing Utilization of Resource Pools in Web Application Servers, Concurrency and Computation: Practice and Experience, Vol. 22, Danvers 2010, pp. 2424-2444

[7] Diao Yixin, Hellerstein Joseph, Parekh Sujay: Using Fuzzy Control to Maximize Profits in Service Level Management, IBM Systems Journal, Vol. 41, No. 3, New York, 2002, pp. 403-420

[8] Prpič Martin et al.: RedHat Enterprise Linux 6.5 GA Resource Management Guide [online], New York 2013, Available on: https://access.redhat.com/site/documentation/en-US/Red_Hat_Enterprise_Linux/6/pdf/Resource_Management_Guide/Red_Hat_Enterprise_Linux-6-Resource_Management_Guide-en-US.pdf [quoted: February 2, 2014]

[9] Mihályi Daniel, Novitzká Valerie: Towards the Knowledge in Coalgebraic Model of IDS, Computing and Informatics, Vol. 33, No. 1, Bratislava 2014, pp. 61-78

[10] Mihályi Daniel, Novitzká Valerie: What about Linear Logic in Computer Science?, Acta Polytechnica Hungarica, Vol. 10, No. 4, Budapest 2013, pp. 147-160

[11] Sitko Maroš, Madoš Branislav: Specialized Information System for Web Content Management, Computer Science and Technology Research Survey, Vol. 6, Košice 2012, pp. 364-370

[12] Barham Paul et al.: Xen and the Art of Virtualization, Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles (SOSP), Bolton Landing 2003, pp. 164-177

[13] Sugerman Jeremy, Venkitachalam Ganesh, Lim Beng-Hong: Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor, USENIX Annual Technical Conference, General Track, Boston 2001, pp. 1-14

[14] Gulati Ajay et al.: Cloud-Scale Resource Management: Challenges and Techniques, Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing, Portland 2011, p. 3

[15] Engler Dawson et al.: Exokernel: An Operating System Architecture for Application-Level Resource Management, ACM SIGOPS Operating Systems Review, Vol. 29, No. 5, New York 1995, pp. 251-266

[16] Seawright Love, Mackinnon Richard: VM/370—a Study of Multiplicity and Usefulness, IBM Systems Journal, Vol. 18, No. 1, Riverton 1979, pp. 4-17

[17] Vokorokos Liberios, Pekár Adrián, Ádám Norbert: Data Preprocessing for Efficient Evaluation of Network Traffic Parameters, Proceedings of 16th IEEE International Conference on Intelligent Engineering Systems (INES), Lisbon 2012, pp. 363-367

[18] Vokorokos Liberios, Baláž Anton, Trelová Jana: Distributed Intrusion Detection System using Self Organizing Map, Proceedings of 16th IEEE International Conference on Intelligent Engineering Systems (INES), Lisbon 2012, pp. 131-134

[19] Madoš, Branislav: Architecture of Multi-Core System-on-the-Chip with Data Flow Computation Control, International Journal of Computer and Information Technology (IJCIT) Vol. 3, No. 5 (2014), ISSN 2279-0764, pp. 958-965