Practical runtime security mechanisms for an aPaaS cloud

Mehmet Tahir Sandıkkaya
Computer Engineering Department
Istanbul Technical University
Istanbul, Turkey
Email: sandikkaya@itu.edu.tr

Bahadır Ödevci
Imona Technologies
Istanbul, Turkey
Email: bahadir.odevci@imona.com

Tolga Ovatman
Computer Engineering Department
Istanbul Technical University
Istanbul, Turkey
Email: ovatman@itu.edu.tr

***Abstract*—An emerging concept of today's cloud is aPaaS (application PaaS), which combines the ready-to-use software services of SaaS, application serving and development functionality of PaaS, and a convenient marketplace for the developed applications. The integrated development environment of an aPaaS usually provides drag-and-drop application creation and script embedding user interfaces to develop software that will be marketed and served within the same cloud. Yet, enabling application developers embed scripts or instantiate objects brings up security issues as deliberate or accidental actions may threat any cloud stakeholder during development or execution. The paper presents practical solutions to inspect tenants' software in the runtime in terms of object instantiation, method calls and CPU load generation. In the prototype implementation, object instantiation and method calls are managed to regulate access to critical file system or socket resources. Also, CPU load generated by each tenant is monitored to detect possible malicious or erroneous activity, which allows to free the CPU resources when necessary. According to the simulation results based on the prototype implementation, running the mentioned security mechanisms adds an overhead up to 20%, which is an acceptable absolute value around 2 ms, to the web applications served in the cloud in idle and normal load conditions. The mechanisms are scalable as the overhead relatively decreases with the increasing number of concurrent users.**

***Keywords*—PaaS, aPaaS, cloud, runtime, security**

I. Introduction

People tend to join the accelerating cloud adoption trend in the recent years. However, for small to medium scale businesses, this is not an easy task for several reasons. First, they mostly run legacy software at some point in their business which makes data migration difficult. Second, they do not employ experienced personnel to conduct the cloud migration and then take care of the following operational tasks. Third, developing a cloud-based software replacement for the legacy code is usually far from that such businesses can handle or invest. Finally, the required software logic may not exist and it might not be feasible for a global cloud provider to invest in such development.

A. Motivation

In the light of aforementioned facts, an easy-to-use cloud-based integrated development environment, together with a marketplace for the developed cloud applications, are expected. Such an integrated cloud service acts as Software-as-a-Service from the businesses' point of view where it is a Platform-as-a-Service from the developer's perspective. The service that covers both SaaS and PaaS is named as application PaaS, or aPaaS in short [1].

In an integrated cloud service where development tools, a marketplace and an application engine are featured, many cloud players can act together. Developers could design, implement and test cloud-based applications without any additional software rather than their browser. They can market these applications to the public through the marketplace. Businesses could select among the developed applications for a small fee. Even better, they could order the exact software they need. As the development is straightforward, the total cost of the cloud application will be less compared to the equivalent desktop application, both in units of time and money. Moreover, the users could use the cloud applications through their browser in any computer without the necessity of additional setup or configuration. Some other benefits of such a cloud service may be listed as follows. First, as integrated on-line development tools abstract many of the configuration details, developers could focus on algorithms and modeling. This feature ought to dramatically reduce development time and effort. Second, database connections and modeling are no longer among the responsibilities of the developer; so, the developer just needs to decide data representation. This approach even removes the barrier of database knowledge to develop enterprise applications. Third, versioning is able to be handled automatically. Whenever the developer updates the application, users receive the update next time they use the application automatically. Moreover, any change that is made in the data model or representation is able to be migrated to the new version transparently. Finally, as the data and even the application logic is abstracted, the application can be migrated together with the respective data of a specific user to a new infrastructure provider (IaaS) effortlessly; therefore this approach maintains provider independability.

B. Problem Definition

Unfortunately, there are also problems to be dealt with in an aPaaS. The service provider has the utmost responsibility to protect its customers' data, where the data in this context may imply either the program logic of a developer or the financial records of an enterprise that uses an accounting application in the cloud. Besides, protecting its own computational resources both during development and also during execution

against attackers is the natural concern of the service provider. Therefore, a stable and reliable protection mechanism must be deployed. As stated in a related work [2], isolation of resources from running applications has a visible benefit as it enables monitoring and controlling access to the resources.

Isolating resources and monitoring resource access of an application can guarantee unintended use of resources. Additionally, interaction of applications can be managed if the isolated resources span the ones which applications can interact through, such as sockets or files. Accordingly, isolation may help to solve many problems at once.

Another security mechanism is monitoring resource load. This helps to determine misbehaving tenants, even in their isolated environments, and finalize their activity.

The rest of the paper is organized as follows. The related work is introduced in the next section. Section III is kept for the proposed solutions to the discussed problems of aPaaS clouds. The prototype implementation is explained in Section IV. Section V presents the simulation and interprets the results. The paper is concluded with Section VI.

II. Related Work

Many examples of security developing architectures through virtual machine monitoring are proposed in the literature [3]. Related to our implementation, dynamic security contexts in multi-threaded environments is studied in [4]. Within the same years, the Java™ SE Platform Security Architecture is drafted [5]. This architecture is continuously developing since then. Finally, the proposed solutions in a different cloud environment is mentioned in one of our previous articles [2].

The generic approach for application isolation (or sandboxing) is deploying a trusted computing base (TCB) as a new layer in between the operating system and the running process. This approach has many examples [5] [6] and fits perfectly to be used in a single instance of an operating system. However, it cannot handle the dynamic structure of clouds. In the clouds there exist multiple different operating systems on top of heterogeneous platforms that are orchestrated to act as one service with practically infinite resources. Relying on separate operating systems' TCBs would not guarantee the overall security of the cloud service if they cannot be managed adequately. However, coordinating all those TCBs across the cloud is not practical.

The more problematic side of the generic TCB approach is the fact that a TCB is often designed as a layer where computational resources are abstracted for the upper layer of running processes. As a result, the running processes are forced to use the specific software objects or interfaces to access the resources. However, this is not easy to obtain in the current cloud offerings. Many workarounds could be possible to access the resources for different deployment scenarios – which is likely to occur in many different ways in clouds.

On the other hand, many cloud services are designed as web applications to work flawlessly out-of-the-box without additional setup or configuration. This approach directs cloud service providers to facilitate application servers and design their products with respect to the Internet's dominating REST [7] architecture. In turn, it leads them to implement a monolithic web application where each request leads to a thread that handles a single transaction from any user, classify the request and finally build the response. A monolithic web application may have several users at a given time who are communicating with several other parties as well as databases through this monolithic web application itself. From the TCB's point of view, every stakeholder is just another piece of the same monolithic application with the same access rights. Hence, such deployment does not work for current aPaaS services.

III. Proposed Solution

One of the ways to isolate cloud applications that reside in a single monolithic web application is dynamically isolating each transaction that belongs to one of the cloud applications. The web application works as the cloud service itself, the cloud development environment, the marketplace and finally as several instances of the developed cloud applications. This makes the single web application adapt and mimic itself dynamically to respect the different features of different applications according to the request. In that sense, cloud applications are thought as sub-applications of the monolithic web application.

The easiest part of the solution is deciding the protection domain as the web application already has adaptation capability based on incoming requests. Consequently, the isolation settings are deductible from the request. However, applying these settings may seem troublesome.

Apart from layered TCB approaches, the cloud applications, the marketplace, the development environment and even the service itself act as a single process to the underlying operating system – or virtual machine. As a result, each cloud application has the same access rights to the computational resources according to the underlying system. Furthermore, the cloud applications use the same objects and interfaces within the web application. A novel mechanism must be built into the web application logic to dynamically switch in between isolation strategies and enforce them based on incoming requests.

A. Isolating I/O resources

In the proposed approach, each transaction obeys a cycle of events. The first thing to do right after a request is received is deciding the type of the request and fetching the relative context from the database. At that point, deciding dynamic isolation settings is straightforward. Based on the isolation settings decided, the current transaction may have assigned a set of permissions. After the permissions are assigned, the program logic continues to run to send back the respective response to the cloud user.

According to our proposal, if no permissions are assigned during a transaction, this makes the transaction restricted to response only within the program logic, without access to any additional resources. The permissions to access a resource may be of two types; a transaction is permitted either to instantiate an object that abstracts a computational resource or call a method to an interface that abstracts a computational resource. The first type of permissions are used to limit unintended access to the resource defining objects where the second type of permissions are used to verify which transaction a resource

is used from. During a transaction, one must have at least two of these permissions together to access a resource.

A final note is that these permissions are equally effective both during development and cloud application execution. At the end, the provided development environment is no different than another application except that it is maintained by the cloud provider and has a capability to dynamically add more applications to the cloud.

B. Monitoring CPU load

Permissions regulate access to the named resources within the programming languages such as files, databases or network connections conveniently. On the contrary, central processing unit (CPU) allocations and computation time remain hidden within the context of the programming languages. Still, these are very valuable computational resources that cannot be left uncontrolled. Unfortunately, as an aPaaS is an open platform for conscientious developers, it is also open to adversaries. As a result, protecting the cloud service and also its users from CPU misuse, another innovative mechanism is deployed within the platform.

Fortunate part of existing aPaaS implementations is the fact that they work based on transactions and each transaction strictly takes place in a separate thread. As mentioned earlier, deciding the protection domain is an easy process when it is made according to the incoming requests. As each transaction is related to a cloud application (or some specific part of the cloud service), monitoring the CPU cycle and time span of transacting threads make use of valuable data. Apart from being able to set static time bounds or CPU usage rate bounds, it is possible to reason misbehaving applications based on these monitored data and its collective interpretation.

Consequently, any thread – thereby transaction, thereby cloud application, thereby respective developer – that passes beyond the predefined time limit, CPU usage rate limit or seems like suspiciously misbehaving, raises an exception. Even if the developer tries to by-pass it, such an exception is always caught by the aPaaS when the developer tries to access any other I/O resource. Frankly, even an adversarial developer does not have any other choice rather than accessing one of the provided I/O resources or services to do a substantial evil task within the platform. Therefore, it is not likely for an adversary to by-pass such a protection mechanism unnoticed.

Finally, it is worth to obviate any misunderstandings. A developer must not be able to access the whole feature set of the relative programming language in the cloud development environment. It is very likely that most of the cloud development environments limit the cloud developers to use some of the error-prone or security-related keywords of a programming (or scripting) language. Therefore, this elementary prevention mechanism neither discussed in the paper nor developed in the prototype implementation.

IV. Implementation details

The proposed solution is generic and applicable to any platform independent from underlying operating system, application server or programming language. Still, it is believed that explaining the implementation and presenting the usage with popular examples may be helpful for better understanding of the concept.

A. Implementing I/O isolation

The aPaaS instance where the proposed solution is deployed is a common web application environment where the underlying application server is an Apache Tomcat with the capability of running Java™ bytecode on the server side. The implemented permissions are designed in comply with the regular permissions [5] of the Java™ language. Even though the underlying mechanisms to enforce the permissions are different, instantiation permissions and method calling permissions are merged into one permission class for ease of use during programming. Two sample instances of the joint permission are shown below to express the aim.

```
JointPermission("java.lang.*", "*");

JointPermission("java.io.File",
    "TestClass.permittedMethod", "java.*");
```

In the first permission above, all classes of “java.lang” package is permitted to be instantiated and then any method call from any method of any class to the classes within the “java.lang” package is permitted.

In the second permission, instantiation of “java.io.File” class is permitted. Then, calling its methods from “permittedMethod” method of “TestClass” class as well as any method of any class within “java” package and its sub-packages is permitted.

Java™ enables runtime class loading decisions. According to the Java™ language specification [8], each reference to an object is managed by the class loader of the context even if another instance of the same class is loaded before. One must implement a class loader inherited from “java.lang.ClassLoader” and override one or more of this class’ methods to handcraft a particular class loader. Most third-party class loaders loads a class from another source if the parent class loader fails. On the other hand, the proposed approach requires strict controlling of loaded objects within the context. Instead of waiting for the parent class loader to fail, hooking the very beginning of the class loading logic is suitable to achieve this. In this manner, any reference to any object in the running code can be inspected even before loading any instance of an object. The access to the non-critical software objects can be delegated to the parent class loader afterwards. Note that, it is not possible for an unpermitted method to instantiate a class through permitted methods as each separate call in the call stack must have the required permission.

To put it simply, the designated class loader is loaded with a set of permissions linked with its protection domain. These permissions define the object instantiation and method call boundaries of different contexts within the cloud. Whenever an object initialization is executed in the system, the class loader generates a temporary permission object. This temporary permission object actually represents the permission that must be defined within the class loader’s protection domain to successfully instantiate an object. Then, the temporary permission is compared with the permission set defined within the class loader’s protection domain. If the permission set do

not imply the temporary permission, an exception is thrown. Otherwise, the algorithm continues to check the method calls. In this second phase, temporary permissions are generated for each stack trace element akin to the previous phase. For each method in the stack, the temporary permission is compared with the assigned permission set of the class loader. If the permission set does not imply one of the temporary permissions at any point, an exception is thrown. The algorithm continues to instantiate the object from the class file unless an exception is not thrown up to the point. If class loading cannot succeed, the designated class loader delegates class loading to the parent class loader. The pseudo-code of the depicted algorithm is shown in Algorithm 1.

Algorithm 1 I/O isolation algorithm

```
permSet ← define permission set of contexts
cLoader ← instantiate with permSet
while ongoing object initialization do
  iPerm ← permission required to initialize the object
  if not permSet imply iPerm then
    throw exception
  else
    for all methods in stack do
      mcPerm ← permission required to use the object
      if not permSet imply mcPerm then
        throw exception
      end if
    end for
  end if
end while
object ← instantiate object
if not object is initialized then
  delegate instantiation to parent
end if
```

B. Implementing CPU load monitor

The virtual machine management features of Java™ are included in the language starting from version 1.5 [9]. The package is capable of querying the CPU time of a thread running within the same process on the JVM (Java™ Virtual Machine). This is not very useful in the context of the proposed approach, as the CPU time of a thread hardly tells anything about the actual load of a thread. Nonetheless, if relative CPU rate among all other threads of the same process and the ratio of CPU share for each thread are measured, these metrics can be used to reason about the load of a thread to the underlying system. Furthermore, an upper limit to finalize the assigned task, namely a timeout, can be determined.

The presentation of the proposed CPU load monitoring mechanism mentions separate threads for each transaction. However, this is hardly correct and useful in reality. Most of the time, threads are not trashed when they finish their task; instead they are added to a thread pool for a specified period of time. The cost of instantiating a separate thread for each transaction is decreased in that manner. The environment where the prototype implementation deployed is not any different from that. So, the implementation has slight differences with respect to the stated solution to overcome the additional issues raised by the reuse of the threads.

In the prototype implementation, each thread reports the current time and its current load at the beginning and at the end of each transaction handling. Also, when a thread uses a service of the cloud, for instance database access, the thread's statistics are updated. Additionally, a separate daemon thread collects information about the running threads in regular intervals to deal with the threads that may run for relatively longer than usual. A central data structure keeps track of the currently instantiated threads' statistics. The statistics include three crucial information: the average CPU rate among HTTP threads while handling the last transaction, average CPU share of the thread while handling the ongoing transaction, and the elapsed time since the arrival of the request. A simple misuse detection policy that features a set of thresholds with respect to the measured values has been defined during the development of the prototype. The pseudo-code of the described algorithm is given in Algorithm 2.

Note that, there is room to introduce adaptive anomaly detection mechanisms. The reason not to deploy any is introducing such a mechanism requires real-world system's knowledge and statistics. An adaptive mechanism may cause more harm than good without reasonable initial values. Therefore, this task is left out of the scope of the prototype and considered as a future step after the initial mechanism collects enough usage data that some knowledge can be deducible from.

Algorithm 2 CPU load monitoring algorithm

```
if thread starts a transaction or thread finalizes a transaction
or thread accesses a cloud service then
  update statistics of thread
  if cpuRate > cpuRateThreshold or cpuShare >
  cpuShareThreshold or elapsedTime > timeout then
    throw exception
  end if
end if
In parallel:
loop
  wait for the interval
  for all thread in threadPool do
    update statistics of thread
    if cpuRate > cpuRateThreshold or cpuShare >
    cpuShareThreshold or elapsedTime > timeout then
      throw exception
    end if
  end for
end loop
```

V. Simulation

The proposed solution is implemented to work flawlessly within an existing aPaaS cloud service, then deployed in the test setup. The serving host is a regular personal computer with 4 GBs of RAM and Intel® Core™ i5-4200U 1.6 GHz CPU running Microsoft® Windows® 8.1 64-bit operating system where version 1.7.60 of Java™ runtime environment and Apache™ Tomcat™ 7.0.42.A is installed. Apache™ JMeter™ 2.11 is used as the traffic generator in another personal computer with 8 GBs of RAM and Intel® Core™ i7-4700EC 2 GHz CPU running 64-bit Linux® Mint® 16 operating system. Two computers are networked through a Wi-Fi™ channel.

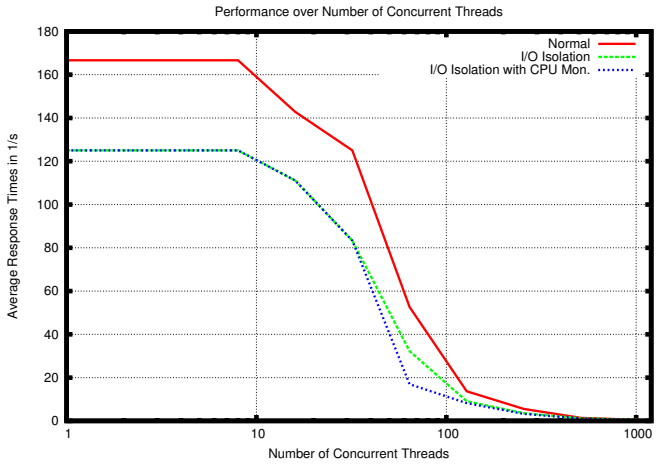

Fig. 1. Performance over concurrent thread number.

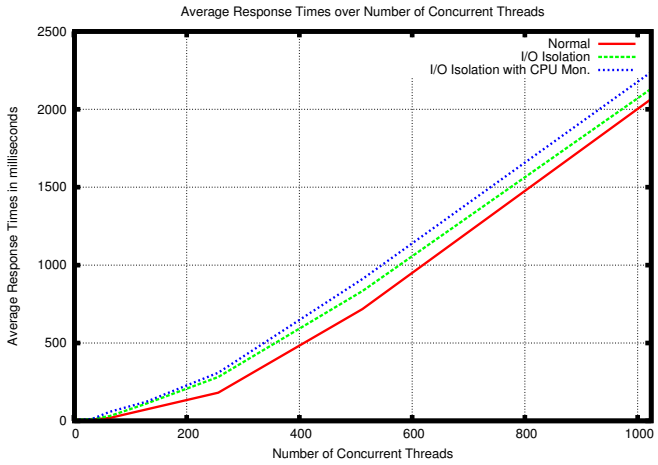

Fig. 2. Average response time over concurrent thread number.

Elementary tests are run to cancel out the odds of memory leaks that may occur during the implementation and network-related delay. For the first case, a single thread continuously generated sequential random requests for nearly five minutes. Measured values at the server side, as well as at the client side stay stable during this period. This trend is interpreted as that there is no critical memory leaks in the implementation. For the second case, elapsed time to handle each request, where the content of the requests and the number of concurrent requests are random, is measured both at the server and the client side. Afterwards, the difference calculated based on the two measurements are aggregated. The distribution clearly leans close to 0 ms time difference; therefore, the delay caused by the underlying network is neglected in the rest of the tests.

The first simulation is run to determine the scalability of the proposed solutions. The number of concurrent requests are increased by a factor of two at each step to monitor server's response to incremental load. According to the results, the server handles concurrent requests up to some extent quite reasonably in terms of scalability. The server is tested under three different usage scenarios. First, "normal" case where none of the security mechanisms are enabled. Second, "I/O isolation" case where Algorithm 1 is enabled but not Algorithm 2. Finally "I/O isolation with CPU load monitoring" case where two of the security mechanisms are concurrently in use.

Figure 1 shows the performance of the server in seconds^{-1} in the y-axis under increasing concurrent thread numbers, which is shown in the x-axis in logarithmic scale. The performance difference for small number of threads is clearly seen when the security mechanisms are in use or not. Still, the critical points of the curves lie in the same vertical positions in the graph; therefore indicate that the capability of handling concurrent threads is mostly independent from the security mechanism usage but dependent on the concurrent thread count, which implies that the proposed security mechanisms are scalable. The performance values of three measurements converge asymptotically. Hence, as the concurrent number of threads increase, the negative effect of security mechanisms relatively decrease. The situation is better visualized in Figure 2, where the y-axis is the average response time. It can be seen that the overhead stays nearly constant – therefore relatively less effective – right to the 250 concurrent threads.

The values shown on the graphs are arithmetic averages of the measured response times during the simulations. Each data point on the graph consists of measurements from at least one hundred thousand separate transactions.

A final note is that Algorithm 2 is not used in this simulation as previously noted. The reason for this decision is, especially for large number of concurrent threads, getting stuck (blocked) on a thread for a period of time is more likely to happen. After the determined timeout these threads are interrupted, therefore, relieve the server load and increase the performance. As the aim of the simulation is to show the worst case scenario, the algorithm is modified to not to throw any exceptions during the simulation, but stay blocked and worsen the performance.

According to the recently presented results of the first simulation, the server's linear response to the number of concurrent threads are somewhere around 32 to 64. The smaller number of concurrent threads do not really affect the server's performance; while in contrast, the larger number of concurrent threads affect server performance tremendously. Based on this observation, the number of concurrent threads are kept constant at 50 for the second simulation.

The aim of the second simulation is to determine the effects of different security mechanisms to the response time under different usage scenarios. Some methods are prepared to access the file system and read a file or to access a socket and control network traffic. These methods are called with or without required permissions. Therefore, in the "normal" case unauthorized file or socket access is possible where it is not possible in the other two cases. More than twenty thousand requests that uniformly span over all kinds of methods with randomly chosen valid or invalid permissions are uniformly generated and fed to the server simultaneously over 50 threads. The distribution of response times are shown below. Figure 3 presents the effect of security mechanisms to socket access. Figure 4 focuses on the file system activity. Finally, Figure **??** shows the overall results. In these figures, the x-axis is the

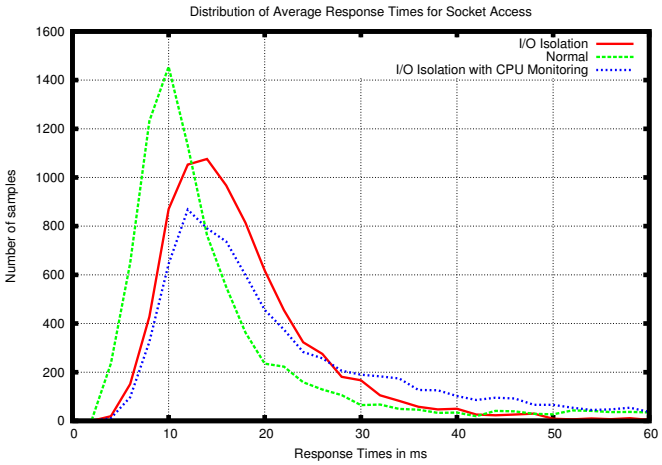

Fig. 3. Distribution of response times under different cases for network activity.

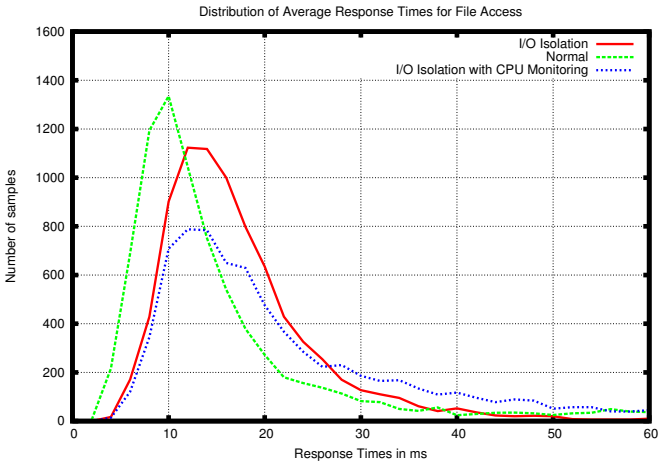

Fig. 4. Distribution of response times under different cases for file system activity.

response time in milliseconds where the y-axis indicates the number of samples.

An inspection of Figure 3, and Figure 4 exposes the fact that, apart from the y-axis scale, the figures resemble each other. Hereby, one may say that the effects of the proposed mechanisms are independent from the underlying activity. In both figures, the best performing case is the "normal" case as it leans to the left most. In "I/O isolation" case the peak is less pointed and leaned relatively to the right. Even though the third case looks like the worst performing case, the position of the peak is still satisfactory. The relative difference of the peak points in both figures is around 20%, yet the absolute difference is less than 2 ms and it can practically be neglected for idle to normal server load. Please note that, as stated before, the effect of the security mechanisms is a nearly constant time delay for high load conditions.

VI. Conclusion

Two practical security mechanisms, targeting some specific problems of aPaaS clouds, are introduced in the paper. The proposed I/O isolation mechanism protects the virtual sub-applications that reside within a single monolithic web application. This protection is achieved by isolating access to critical I/O resources based on the dynamically specified contexts. The second mechanism monitors the CPU load of separate threads, therefore separate users, of the web application and takes required measures to free the cloud resources in case of suspicious activity.

The practical applicability and feasibility of the mechanisms are shown as a prototype implementation is deployed in a running example of an aPaaS cloud instance to run the simulations. As stated, the simulation results are promising.

Furthermore, dynamic decisions of security contexts within the same web application and dynamic enforcement of these security contexts simultaneously in different virtual sub-applications of the single monolithic web application is a novel approach which may lead to further benefits and may refine new security mechanism designs for the multi-threaded multi-user cloud environments.

Acknowledgment

This work is performed as a part of project number 01677.STZ.2012-2 "Operational Optimization of a Cloud System Serving Software Development Platform" supported by SAN-TEZ program of T.C. Ministry of Science, Industry and Technology and Imona Technologies Ltd.

References

[1] Gartner Inc., "Gartner IT glossary," http://www.gartner.com/it-glossary/application-platform-as-a-service-apaas/, accessed: 2014-07-15.

[2] M. T. Sandikkaya and A. E. Harmanci, "Security problems of Platform-as-a-Service (PaaS) clouds and practical solutions to the problems," in *Proceedings of the 2012 IEEE 31st Symposium on Reliable Distributed Systems*. IEEE Computer Society, 2012, pp. 463–468.

[3] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A VMM security kernel for the VAX architecture," in *Research in Security and Privacy, 1990. Proceedings., 1990 IEEE Computer Society Symposium on*. IEEE, 1990, pp. 2–19.

[4] L. Koved, "Multiple resource or security contexts in a multithreaded application," Jun. 22 1999, US Patent 5,915,085.

[5] L. Gong, "Java SE platform security architecture," http://docs.oracle.com/javase/8/docs/technotes/guides/security/spec/security-spec.doc.html, accessed: 2014-07-15.

[6] Y. Kosuge and IBM Corp. International Technical Support Organization, *AIX 4.3 Elements of Security: Effective and Efficient Implementation*. IBM, International Technical Support Organization, 2000. [Online]. Available: http://books.google.com.tr/books?id=GhmnGwAACAAJ

[7] R. T. Fielding and R. N. Taylor, "Principled design of the modern web architecture," *ACM Transactions on Internet Technology (TOIT)*, vol. 2, no. 2, pp. 115–150, 2002.

[8] K. Arnold, D. Holmes, T. Lindholm, F. Yellin *et al.*, "Java language specification," 2000.

[9] Oracle Corporation, "Monitoring and management for the Java platform," http://docs.oracle.com/javase/8/docs/technotes/guides/management/index.html, accessed: 2014-07-15.