VM Leakage and Orphan Control in Open-Source Clouds

C. Dabrowski and K. Mills
Information Technology Laboratory
NIST
Gaithersburg, MD USA
{cdabrowski, kmills}@nist.gov

***Abstract*—Computer systems often exhibit degraded performance due to resource leakage caused by erroneous programming or malicious attacks, and computers can even crash in extreme cases of resource exhaustion. The advent of cloud computing provides increased opportunities to amplify such vulnerabilities, thus affecting a significant number of computer users. Using simulation, we demonstrate that cloud computing systems based on open-source code could be subjected to a simple malicious attack capable of degrading availability of virtual machines (VMs). We describe how the attack leads to VM leakage, causing orphaned VMs to accumulate over time, reducing the pool of resources available to users. We identify a set of orphan control processes needed in multiple cloud components, and we illustrate how such processes detect and eliminate orphaned VMs. We show that adding orphan control allows an open-source cloud to sustain a higher level of VM availability during malicious attacks. We also report on the overhead of implementing orphan control.**

***Keywords-** availability; cloud computing; modeling; reliability; scalable fault resilience techniques*

I. INTRODUCTION

The impact of resource leakage on computer performance is a well-known problem [1-8]. A number of studies have shown how programming errors [1-3, 7], data corruption [5], and events such as external malicious attacks [6, 9] can cause resource losses, which degrade system performance. Ultimately, if needed resources are depleted, or exhausted, a system can fail [4, 8]. We extend the general concept of resource leakage to encompass virtual machine (VM) leakage in clouds.

The advent of cloud computing has resulted in many innovative applications, which promise to transform the practice of information technology. Much of this innovative work has centered on open-source cloud software [10-14], which has gained widespread distribution. Such open-source software may be used to establish cloud systems for experimentation, for private use and for public use. Unfortunately, development and distribution sites are susceptible to attacks that can place Trojan code into software packages [15]. Such attacks have occurred on both proprietary software [16] and open-source software [17-21]. This paper considers a scenario where Trojan code is inserted into an open-source Web server, used as one component in an open-source cloud software distribution. The Trojan code randomly discards Web messages, a simple malicious attack requiring no knowledge of the internal operation of the cloud software.

Using simulation, we demonstrate how a cloud system, based on the infected software, can exhibit degraded availability of computing resources in the form of virtual machines (VMs). We describe how the simple message-discard attack leads to VM leakage, causing orphaned VMs to accumulate over time, exhausting the pool of resources available to users, and leading to a collapse in system performance. We identify two kinds of VM orphans that could exist in clouds and the circumstances under which they are created. We then suggest a set of orphan control processes and provide examples of their use to detect and eliminate orphaned VMs. We show that adding orphan control allows a cloud system to sustain a higher level of VM availability during message-discard attacks. In addition, we show that more than one orphan control method is needed to prevent performance collapse. We also report on the overhead of implementing orphan control. In doing this, we hope to provide awareness of the potential for resource leakage in clouds, and to further research on cloud reliability.

The paper consists of six sections. Section II describes previous work on resource leakage in computer systems. Section III overviews the cloud model used in this study. Section IV defines the concept of VM leakage in cloud systems, identifies potential causes, and proposes remedial VM orphan control methods. Section V describes the experiment scenario used here, in which a malicious attack on an open-source cloud leads to significant VM leakage. Section VI provides experiment results, and details both the potential impacts of VM leakage, and the remedial effects of orphan control. Section VII concludes.

II. PREVIOUS WORK

The problem of resource leakage in computer systems has received significant attention, most particularly with respect to memory leaks in executing programs coded in languages such as C [1] and Java [2], or in garbage collectors [3]. The effect of memory leaks has also been considered in the study of software aging in Web servers [4]. Other studies use the more general term resource leakage [5-8], and some use the term resource exhaustion to denote total depletion of

978-0-7695-4622-3 2011
U.S. Government Work Not Protected by U.S. Copyright
DOI 10.1109/CloudCom.2011.84

needed system resources, including [6, 8, 9]. The term orphan has been used [5] to refer to leaked database records. Hence, the general concepts associated with resource leakage in computer systems are established. However, to date, the resource leakage problem has not been studied for VMs as resources in computational clouds.

III. Model of an Open-Source Cloud

We based our study on Koala [22], a discrete-event simulator inspired by the Amazon Elastic Compute Cloud (EC2)[1] [23] and by the Eucalyptus open-source software [11]. Using published information describing the EC2 application programming interface (API) [24] and available virtual machine (VM) types [25], Koala models essential features of the interface between users and EC2. Koala models four EC2 commands, three of which we use here: *RunInstances DescribeInstances*, and *TerminateInstances*. The internal structure of Koala is based on three Eucalyptus (v1.6) open-source cloud software components: *cloud controller*, *cluster controller* and *node controller*. As in Eucalyptus, these Koala components communicate using simulated Web Services [26]. In this study, Koala simulates 20 clusters and 200 nodes overall, where each node can be of one of four possible platform configurations (see [27] for details). Koala is organized as five layers: (1) demand layer, (2) supply layer, (3) resource allocation layer, (4) Internet/Intranet layer and (5) VM behavior layer. Elsewhere [28], we provide details of each layer. Here, we summarize and identify selected parameter values used in the experiments described in Sec. V.

In a Koala simulation, a variable number of users (500) execute in a cycle. During each cycle, a user issues a *RunInstances* request to the cloud controller to request a minimum and maximum number of instances of one or more VM types. Each VM type is defined to include an integrated set of virtual cores (processors), memory and disk space. The VM types and quantities a user selects depend upon the user's type, which is also randomly determined (see [27]).The cloud controller may respond to a *RunInstances* request with an allocation of instances between the minimum and maximum for each requested VM type or with a NERA (not enough resources available) fault. A *full grant* denotes that a user was allocated the maximum requested instances of each VM type. A *partial grant* denotes that allocated VMs were below the maximum requested. If VM instances are allocated, a holding time is then determined (mean 4 hours). Upon receiving a grant response, the user issues a *DescribeInstances* request to determine when granted VMs have booted. At the end of the holding period, the user issues a *TerminateInstances* request to stop any remaining running instances, which is called the *final termination* request. If this request fails, the user retries (0 to 3) times before giving up. After termination, or when retries are exhausted, the user pauses for a time (mean 7.5 minutes) and then starts a new request cycle.

During the holding period following a grant, users may randomly terminate subsets of running instances, which we call *intermediate termination* requests. Upon failure of an intermediate termination, a user retries 0 to 3 times for individual instances. If the cloud controller responds to a *RunInstances* request with a NERA, then the user waits for a mean time of 7.5 minutes before retrying the request. The user retries for a random period (mean 2 hours) before resting for a random period (mean 8 hours), until a random number of retry/rest periods (mean 4) occur. Then, the user abandons the request and starts a new cycle.

Koala patterns resource allocation after Eucalyptus procedures, which involve two decisions: (1) on which cluster should requested VMs be allocated and (2) on which nodes within the cluster should VMs be allocated. Allocating all VMs in a single request to the same cluster ensures that inter-VM communications remain local to one cluster. At the cluster level, Koala simulates the Eucalyptus *first-fit* algorithm to choose nodes for VMs. First-fit simply searches nodes by identifier from first to last until a node is found that can accommodate a given VM type. In making an accommodation decision, the cluster controller compares resources required by a VM type against a node's availability of virtual cores, disk space and memory. If no nodes can create the VM, the cloud controller receives a NERA fault.

At the cloud level, Koala simulates the Eucalyptus *least-full-first* algorithm, which carries out an initial estimation in which it polls the clusters to find out which can accommodate the VMs requested and then orders the list from the least to most full (we ordered ties by increasing time at which clusters responded). Then the cloud controller selects the first cluster from the list and asks that the VMs be created. If the VMs are created successfully, then the cloud controller returns the positive result to the user; otherwise, the cloud controller *reassigns* the VMs to the next cluster on the list. This process continues until VMs are created or until all clusters have been exhausted. If no clusters can create the VMs, then the user receives a NERA fault.

IV. VM Leakage and Orphan Control

We use the term *VM leakage* to refer to VMs that exist on node controllers but that are unknown to any user and that are not in the process of being terminated by a cloud or cluster controller. Such VMs are considered *orphans* because they can persist indefinitely. Orphaned VMs constitute a type of resource leakage, because they retain assigned computing resources, including virtual cores, memory, disk space, and network channels. These resources cannot then be allocated for any other purpose, and so are effectively lost (or leaked).

A. Causes of VM Leakage and Orphan Creation

Orphaned VMs are created under two circumstances. In the first, which gives rise to what we will call *creation orphans*, VMs are successfully created in response to a user request, but confirmation messages, reporting VM creation, are lost when transiting among elements within the demand and supply layers. In our model, there are three such

[1] Any mention of commercial products within this paper is for information only; it does not imply recommendation or endorsement by NIST.

opportunities: (1) a lost message from node to cluster controller that indicated successful creation of a VM; (2) a lost message from cluster to cloud controller that indicated successful (full or partial) allocation; or (3) a lost message from cloud controller to user that indicated a successful result. In (1), the result is a single orphaned VM. However, in (2) and (3), all VMs allocated for a request become orphans, and the amount of leakage can thus be quite large. In all three cases, the user will resubmit the request, according to the retry regimen described above. Each re-request is treated as a new request by the cloud.

The second circumstance, leading to what we will call *termination orphans*, occurs after VMs are created by the cloud and the user is notified successfully. Subsequently, the user issues a *TerminateInstances* request for one or more VMs. If the user receives confirmation of successful termination, the user considers the operation to be finished. However, if the user receives no reply, the user retries the terminate operation as described above, until either success is obtained or the number of retries is exhausted. Within the cloud, terminate operations may fail due to lost messages when relaying the request from cloud to cluster controller, or from cluster to node controller, or because the terminate operation fails on the node. Eucalyptus makes no provision for retrying failed termination requests by either the cloud or cluster controllers; instead such failures are merely logged. Thus, the related VMs will remain un-terminated unless a user termination request eventually reaches the related node controllers. If a user abandons termination retries, the affected VMs will persist on nodes until an administrator scans the log and manually terminates them.

If termination orphans arise due to lost termination requests sent from user to cloud controller or from cloud to cluster controller, then all VMs in the request may become termination orphans. In this case the number of orphans and the resulting resource leakage can be quite large. This is particularly true for final terminations, which encompass all VMs held by a user. When termination-related messages are lost between cluster and node controller, only individual VMs become termination orphans.

B. Orphan Control Methods

Neither creation orphans nor termination orphans are detected and removed automatically by Eucalyptus. We therefore devised two orphan control methods for this purpose. First, to eliminate creation orphans, we instituted a node controller process, which monitors receipt of *DescribeInstances* requests for VMs. VM requesters use replies to *DescribeInstances* requests to determine when allocated VMs are ready for logon. Since these requests originate from users, they indicate a user's awareness of the VM. In the node controller, a *creation orphan monitor* relates arriving *DescribeInstances* requests to recently created VMs. If a *DescribeInstances* request is not received for a VM by a specified time (2 h) after boot up, the monitor declares the VM to be an orphan, terminates it, and releases its resources for use by the supervising cluster controller.

Second, to mitigate termination orphans, we extended the Eucalyptus protocol to provide a *persistent termination* capability to both the cloud and cluster controllers. Persistent termination simply means resending termination requests until the receiver responds that either (1) the termination request has been received and normal termination commences, or (2) the termination operation was completed earlier and no further action is needed. A persistent terminator is activated by the cloud or cluster controller when no response is received to a normal termination request within a timeout (90 s). Once activated, the *cloud persistent terminator* resends termination requests to a cluster controller at specified intervals (90 s) until it receives one of the two desired responses, or until a 2 h termination period ends. After the first three retries, the cloud persistent terminator lengthens the retry period (to 150 s) and then doubles it on each retry. If the termination period ends, the cloud persistent terminator ceases and notifies an administrator that manual intervention is needed to terminate the orphaned VMs and free their resources. When activated, the *cluster persistent terminator* also attempts three retries to the node controller (every 90 s), before increasing the retry interval in the manner described for the cloud persistent terminator. This process continues for a shorter period (900 s), since the retry encompasses only a single orphaned VM. Since persistent termination adds complexity to the cloud and cluster controllers, we elected to limit persistent termination to final termination requests. Because intermediate termination requests are excluded, failed intermediate terminations can result in the affected VMs persisting until a final termination request succeeds. We call such affected VMs *temporary orphans*.

V. Experiment Design

In designing our experiment, we sought to address the following questions. How does VM leakage affect system performance when lost messages interfere with resource allocation (*runInstances*) and termination operations? Can orphan control methods mitigate performance degradation caused by VM leakage? What are the costs of orphan control and how do such costs increase as the rate of VM leakage increases? We modeled an attack scenario in which Trojan code is introduced into an open-source distribution for Web server software. The Trojan code modifies the Web server so that arriving and departing messages are discarded randomly with some probability. We assume that the maliciously modified Web server is deployed by all users, cloud controllers, cluster controllers, and node controllers.

To understand effects from increasingly frequent message discards, we simulated our model under six, order-of-magnitude, increases in message discard probability from a lowest probability (10^{-6}) in which one in 10^{6} messages is lost to a highest probability (10^{-1}) in which one in 10 messages is lost. All messages, regardless of type or component of origin, are subject to possible loss.

To assess the benefits of orphan control, we modeled the operation of the system at each of the six message loss rates (10^{-6} to 10^{-1}), both with and without each of the two orphan control methods, creation orphan control and persistent termination, identified in Sec. IV. Holding the configuration parameters described in Sec. III constant, we executed Koala during 1000 simulated hours for each of 24 combinations: on/off for two orphan control processes × six message loss rates. During the simulation, we measured system performance at 1 h intervals, as described below.

VI. Results and Discussion

We counted the number of VMs held by both users and node controllers and the number of orphans created at the end of the 24 1000–hour simulations. With no orphan control, Fig. 1 shows that as message loss rate increases, a large gap opens between the number of VMs held by node controllers (over 11 000 at the highest loss rate) and the number held by users. When the message loss rate reaches 10^{-2}, the number held by users falls to nearly zero. In contrast, with creation orphan control and persistent termination operating, the gap stays relatively small until the highest message loss rate, where node controllers hold about 14 000 VMs, while users hold only 6 000. Figure 2 shows that without orphan control nearly all VMs held by node controllers become orphans at the two highest message loss rates. This means that the Trojan attack has led to creation of orphans that consume most of the simulated cloud's computing resources, leaving none to allocate to incoming requests. Hence, due to leaked VMs, nearly total resource exhaustion occurs. Fig. 2 also shows that almost all of the leaked VMs arise from creation orphans. We say more about this below.

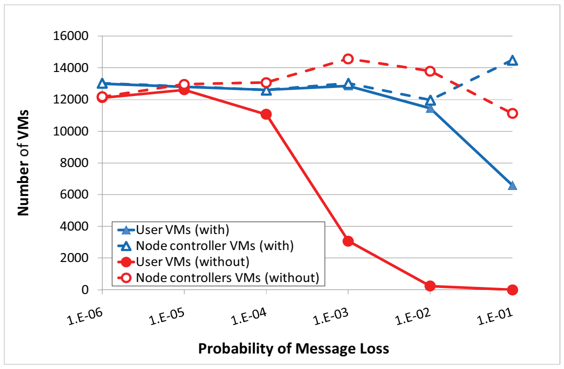


Figure 1. Number of VMs held by users and node controllers with (blue) and without (red) creation orphan control and persistent termination at the end of the1000-hour simulated period as the message loss rate increases.

To measure the influence of VM leakage on cloud performance, we tracked the number of user requests submitted to the cloud and recorded the total proportion of users granted some VMs, along with the proportion that were full and partial grants. We also recorded the proportion of users not granted VMs, users who subsequently abandoned the request process.

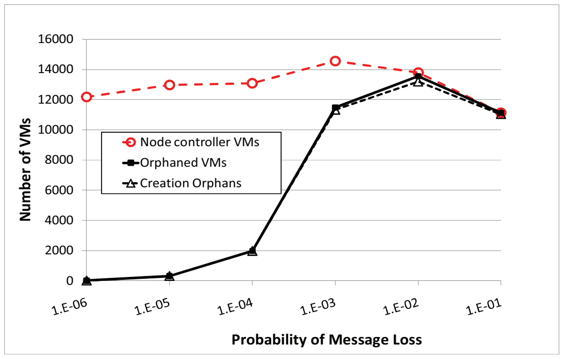
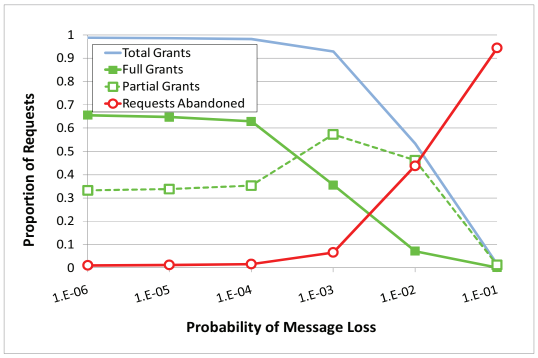


Figure 2. Number of VMs held by node controllers and number of orphans at the end of the 1000-hour simulated interval as the rate of message loss increases. Counts are plotted for the case without persistent termination and creation orphan control.

Figure 3(a) shows that without orphan control, the number of total grants (full and partial) drops sharply as the message loss rate passes 10^{-3}. At the same time, the number of un-served users increases. At the highest message loss rate, 94.4 % of users are not served, while only 1.5 % of users receive grants. For the remaining 4.1 % of requests (not graphed), users are still engaged in the request cycle. On the other hand, with both orphan control processes operating, Fig. 3(b) shows the rate of total grants decreases only slightly until the highest message loss rate is reached, at which point a noticeable drop appears, as 5.7 % of users are not served, while 0.4 % are still actively requesting VMs. We conclude that, without orphan control, the collapse of system performance at higher message loss rates, as illustrated in Fig. 3(a), is attributable to resource exhaustion due to orphan VMs. This conclusion is supported by more detailed analysis below.

These results do not mean that our orphan control procedures free a cloud of all the effects of VM leakage. Figure 3(b) also shows that even with orphan control, the proportion of full grants decreases and the proportion of partial grants increases at the highest message loss rate, to the point that partial grants are more likely. This change can be related to Fig. 1, which shows that node controllers hold more VMs than users at the highest loss rate, even with orphan control. This gap occurs because we chose to limit persistent termination to only final termination requests. Thus, when earlier intermediate termination requests from users fail, the related VMs continue to occupy cloud resources as temporary orphans until a final termination is issued and succeeds. Though these temporary orphans do not exhaust resources, Fig. 1 shows that, at the highest loss rate, temporary orphans still occupy a significant portion of VM resources. Thus, the cloud is less able to fully satisfy requests and must issue more partial grants.

(a)



(b)

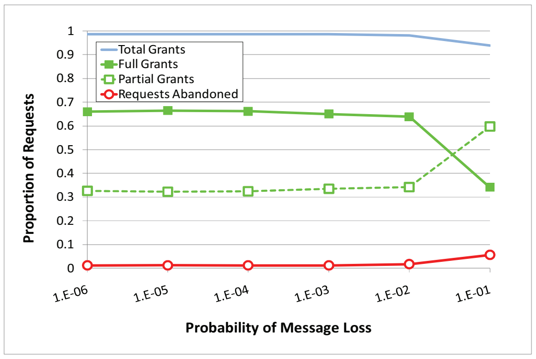


Figure 3. Disposition of user requests (a) without orphan control and (b) with creation orphan control and persistent termination.

Recall that in Fig. 2 nearly all VM leakage is due to creation orphans (only four are termination orphans). Without orphan control, the dominance of creation orphans occurs for two reasons. First, *RunInstances* requests occur before *TerminateInstances* requests. Second, Eucalyptus treats each *RunInstances* request (including each retry) as a new and separate allocation request rather than as a retry of a previous request. Hence each user re-request is an added opportunity for creation orphans. Thus, at high loss rates, creation orphans quickly (in the first 100 hours) exhaust nearly all of the cloud's resources, leaving few opportunities for termination orphans to occur.

To design appropriate orphan control strategies, it is important to determine the extent to which both creation orphan control and persistent termination are needed. To answer this question, we conducted trials in which only one of the two orphan control methods was active (graphs omitted). With only persistent termination active, a total system performance collapse occurs that is similar to what appears in Fig. 3(a). When only creation orphan control is active, the performance decline is partial, but still crippling (48.1 % of all users are not served at the highest loss rate). In this latter case, over time, accumulation of termination orphans leads to significant VM leakage.

Hence, we conclude that both creation orphan control and persistent termination are needed. Otherwise, unless an administrator finds and removes orphans, the cloud moves toward a frozen state, where all VMs are orphans, and so incoming user requests cannot be satisfied. We leave it to the reader to speculate the difficulties involved in perusing system logs throughout thousands of nodes in a cloud and manually finding and removing termination orphans. Further, in the absence of usage billing, there appears to be no obvious manual process to discover creation orphans. Even with usage billing, creation orphans cannot be identified until users raise objections after receiving their bill for VMs of which they were unaware.

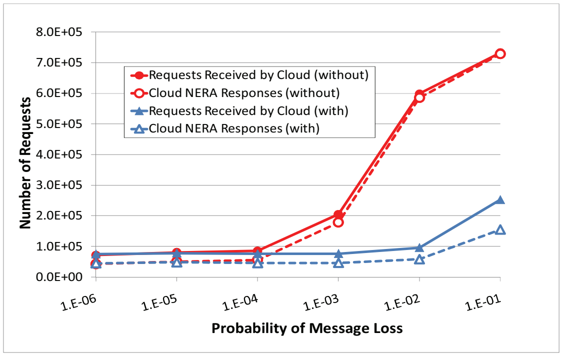


Figure 4. User requests received by cloud controller and NERA responses as the message loss rate increases, with (blue) and without (red) creation orphan control and persistent termination.

To further understand the impact on system performance, we observed the number of user *RunInstances* requests received by the cloud and the number of NERA responses, as message loss rate increased. Our analysis supports the conclusions reached above. Figure 4 shows that without orphan control, at the highest loss rate, a threefold increase occurs in the number of requests, nearly all of which result in NERAs. This reflects the cloud controller's inability to find a cluster to accommodate incoming requests, as cluster resources are almost fully exhausted by orphans. The rise in the number of requests reflects the resultant thrashing caused by user retries. With creation orphan control and persistent termination active, Fig. 4 shows that increasing loss rate leads to only a modest rise in requests, most of which are granted. Elsewhere [27], we provide additional analysis at the cluster level, which supports these findings.

Finally, we counted the total number of messages sent across all layers in the cloud system as the message loss rate increases. Without orphan control, the overall number of messages increased from about 2.6E+07 at the lowest loss rate to about 6.4E+07 at the highest rate. This reflects increased effort expended as users retry requests, causing the cloud to make failed allocation attempts, as VM leakage drains needed resources. With orphan control, the overall number of messages increased only to 4.5E+07 at the highest loss rate, with only about 0.44 % of these messages related to orphan control.

VII. CONCLUSIONS

We addressed the potential problem of resource leakage in cloud systems and introduced the concepts of VM leakage and orphan VMs. We demonstrated that VM leakage is a potentially serious vulnerability that can lead to resource exhaustion in clouds. Using a scenario, in which a Trojan attack introduces malicious code modifications into one part of an open-source cloud implementation, we showed how this vulnerability can be exploited to cause serious performance degradations in a simulated cloud system. To remedy this problem, we also provided examples of orphan control processes that could be used to detect and eliminate orphaned VMs. Our experiment results show that adding orphan control methods allows an open-source cloud to sustain a higher level of resource availability during malicious attacks. Our work has illustrated that VM leakage is a potential problem that must be considered in the design of cloud systems, if these systems are to be reliable. The results of our experiments indicate that the scale of the problem precludes manual discovery and removal of VM orphans by system administrators—and that automated means are needed. In the future, it will be necessary to investigate other orphan control methods and to evaluate their performance and accuracy. It will also be desirable to extend this work to obtain a more general understanding the potential for VM leakage in different kinds of cloud systems operating under a wide range of conditions.

REFERENCES

[1] D. L. Heine and M. S. Lam, "A practical flow-sensitive and context-sensitive C and C++ memory leak detector", SIGPLAN Not., Vol. 38, No. 5, May 2003, pp. 168-181.

[2] G. Xu, and A. Rountev, "Precise memory leak detection for java software using container profiling," *Proceedings of the 30th international conference on Software engineering* (ICSE '08). New York, NY, USA, 2008, pp. 151-160.

[3] M Jump, and K. S. McKinley, "Cork: dynamic memory leak detection for garbage-collected languages," *Proceedings of the 34th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages* (POPL '07), 2007, New York, NY, USA, pp. 31-38.

[4] K. Vaidyanathan, and K. S. Trivedi, "An Approach for Estimation of Software Aging in a Web Server", Proceedings of the 2002 International Symposium on Empirical Software Engineering (ISESE'02), 2002.

[5] S. Bagchi, Y. Liu, K. Whisnant, Z. Kalbarczyk, R. Iyer, Y. Levendel, and L. Votta, A Framework for database audit and control flow checking for a wireless telephone network controller. International Conference on Dependable Systems and Networks, Goteborg Sweden, July 2001, pp. 225 – 234.

[6] J. Antunes, F. Neves, and P.Verissimo, "Detection and Prediction of Resource-Exhaustion Vulnerabilities", Proceedings of the 19th International Symposium on Software Reliability Engineering, 2008, pp. 87-96.

[7] M. Arnold, M. Vechev, and E. Yahav, "QVM: an efficient runtime for detecting defects in deployed systems", *SIGPLAN Not.* Vol. 43, No. 10, October 2008, pp. 143-162.

[8] S. Pertet and P. Narasimhan Causes of Failure in Web Applications, CMU-PDL-05-109, Carnegie-Mellon University, December 2005.

[9] J. Lemon, "Resisting SYN flood DoS attacks with a SYN cache", Proceedings of the BSDCon '02 Conference on File and Storage Technologies, February 11-14, 2002, San Francisco, California, USA.

[10] D. Nurmi, et al., "The Eucalyptus Open-Source Cloud-Computing System", Proceedings of the 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, May 18-21, 2009, pp. 124-131.

[11] Rupley, S, "11 Top Resources for Open-source Cloud Computing", GIGACOM, November 6, 2009, http://gigaom.com/2009/11/06/10-top-open-source-resources-for-cloud-computing/

[12] Hinkle, M., "Eleven Open-Source Cloud Computing Projects to Watch", SocializedSoftare.com, January 10, 2010, http://socializedsoftware.com/2010/01/20/eleven-open-source-cloud-computing-projects-to-watch/

[13] OpenStack Cloud Software, http://www.openstack.org/, Accessed August 1, 2011.

[14] Higgenbotham, S, "VMware Launches Open-Source Cloud", GIGACOM, April 12, 2011, http://gigaom.com/cloud/vmware-open-source-cloud/

[15] E. Levy, "Poisoning the software supply chain", *IEEE Security & Privacy*, 1(3), 2003, 70-73.

[16] D. A. Wheeler, Secure Programming for Linux and Unix HOWTO, http://www.dwheeler.com/secure-programs/Secure-Programs-HOWTO/open-source-security.html, accessed on Aug. 18, 2011.

[17] IT World Canada Staff, Trojan Horse Attacks GNU Project, PC World, Aug. 18, 2003. http://www.pcworld.com/article/112071/trojan_horse_attacks_gnu_project.html

[18] Staff, Attacker attempts to plant Trojan in Linux, ZDNet UK, Nov. 7, 2003. http://www.zdnet.co.uk/news/application-development/2003/11/07/attacker-attempts-to-plant-trojan-in-linux-39117696/

[19] R. Singel, Firefox Infects Vietnamese Users With Trojan Code, WIRED, May 7, 2008. http://www.wired.com/threatlevel/2008/05/firefox-infects/

[20] T. Forenski, Open-source hacks - sneaky Skype Trojan code released, ZDNet, August 27, 2009. http://www.zdnet.com/blog/foremski/open-source-hacks-sneaky-skype-trojan-code-released/736

[21] K. J. Higgins, Open-Source Project Server Hacked, Software Rigged With Backdoor Trojan, Dark Reading, Dec. 2, 2010. http://www.darkreading.com/authentication/167901072/security/application-security/228500217/open-source-project-server-hacked-software-rigged-with-backdoor-trojan.html

[22] K. Mills, J. Filliben and C. Dabrowski, "An Efficient Sensitivity Analysis Method for Large Cloud Simulations", Proceedings of the 4th International Cloud Computing Conference, IEEE, Washington, D.C., July 5-9, 2011.

[23] Amazon Elastic Compute Cloud (Amazon EC2) http://aws.amazon.com/ec2/, 2010.

[24] Amazon Elastic Compute Cloud API Reference API Version 2009-08-15.

[25] Amazon EC2 Instance Types http://aws.amazon.com/ec2/instance-types/, 2010.

[26] F. Curbera, et al. "Unraveling the Web services web: an introduction to SOAP, WSDL, and UDDI", Internet Computing, IEEE, March/April, 2002, pp. 86-93.

[27] C. Dabrowski and K. Mills, "Extended Version of VM Leakage and Orphan Control in Open-Source Clouds", NIST Publication 909325.

[28] K. Mills, J. Filliben and C. Dabrowski, "Comparing VM-Placement Algorithms for On-Demand Clouds", *Proceedings of IEEE CloudCom 2011*, Nov. 29-Dec. 1, 2011, Athens.