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 how the overhead of implementing orphan control scales with attack intensity.**

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 [1-9]. A number of studies have shown how poor design and coding errors [1-3, 7], data corruption [5], and events such as external malicious attacks [6, 10] can cause resource losses, which degrade system performance. Ultimately, if needed resources are depleted, or exhausted, a system can fail [4, 6, 9]. 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 [11-15], 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 [16]. Such attacks have occurred on both proprietary software [17] and open-source software [18-22]. 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 several 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 an open-source cloud to sustain a higher level of VM availability during message-discard attacks. In addition, we demonstrate that more than one orphan control method is necessary to prevent wholesale performance collapse. We also report how the overhead of implementing orphan control scales with attack intensity. In doing this, we hope to provide awareness of the potential for resource leakage in open-source clouds, and to stimulate research on techniques to improve cloud reliability.

The paper is organized into six main sections. Section II describes previous work on resource leakage in computer systems. Section III provides details of the cloud model used in our study. Section IV defines the concept of VM leakage in cloud systems, identifies potential causes of this leakage, and proposes some VM orphan control methods. Section V describes an experiment scenario, which we simulated with our cloud model, where a malicious attack upon an open-source cloud leads to significant VM leakage. Section VI gives results from this experiment, 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 [5-9], and some [10] use the term resource exhaustion to denote total depletion of needed resources. There have been studies on leakage of database records [5], DNS server resources [6], software objects in the Standard Widget Toolkit (SWT) [7], software handle resources [23], and other types of software components [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 leakage problem has not been studied for VMs as resources in computational clouds. This paper provides an initial view of the large-scale effects of VM leakage on resource allocation in cloud systems, and introduces concepts and terms specific to the context of cloud computing.

III. MODEL OF AN OPEN-SOURCE CLOUD

We based our study on Koala [24], a discrete-event simulator inspired by the Amazon Elastic Compute Cloud (EC2)[1] [25] and by the Eucalyptus open-source software [11]. Using published information describing the EC2 application programming interface (API) [26] and available virtual machine (VM) types [27], Koala models essential features of the interface between users and EC2. Intended to study resource allocation algorithms, Koala models only four EC2 commands: *RunInstances*, *DescribeInstances*, *Reboot Instances* and *TerminateInstances*. The internal structure of Koala is based on the Eucalyptus (v1.6) open-source cloud software. Specifically, Koala models three Eucalyptus components: *cloud controller*, *cluster controller* and *node controller*. As in Eucalyptus, Koala's simulated cloud, cluster and node controllers communicate using Web Services [28], which Koala also simulates.

Koala modifies the design of Eucalyptus in three ways. First, Koala extends the Eucalyptus *RunInstances* command to allow multiple VM types within a single request, which appears possible in EC2. Second, Koala avoids centralization of node information at the cloud controller, permitting simulation of clouds up to O(10^5) nodes. Third, Koala allows resource allocation to proceed partially in parallel (serializing only the commitment phase), which prevents long queuing delays during periods of intense user requests. In lieu of simulating details of a hypervisor and guest VMs, Koala includes an optional sub-model based on analytical equations representing VM behavior with or without tasks.

Koala is organized as five layers (see Fig. 1): (1) demand layer, (2) supply layer, (3) resource allocation layer, (4) Internet/Intranet layer and (5) VM behavior layer. We describe each layer in turn, omitting the VM behavior layer, which is not used in the experiment discussed here.

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

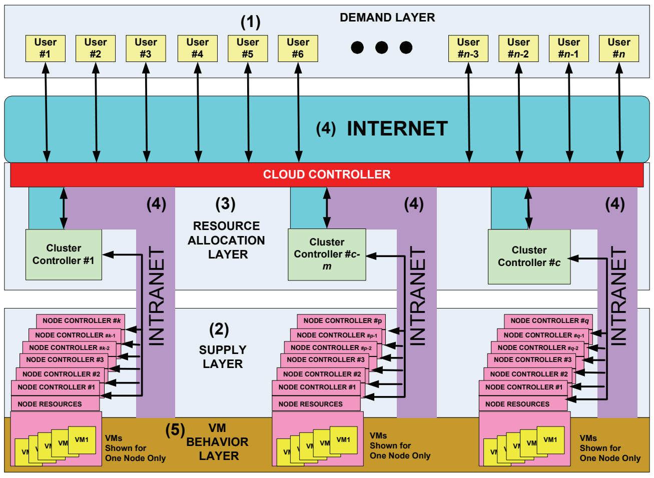


Figure 1. Schematic of Koala organization

A. *Demand Layer*

The demand layer consists of a variable number (500 here) of users who, after a random startup delay, each perform cyclically over a simulation run. During each cycle a user requests a minimum and maximum number of instances of one or more of the VM types shown in Table I. The VM types and quantities a user selects depend upon the user's type (see Table II), which is selected on each cycle with the probabilities shown. After selecting a type, a user randomly chooses a minimum (uniform 1 to max-min in Table II) and maximum (uniform max-min to max-max in Table II) number of instances to request for each associated VM type. The user then issues a corresponding *RunInstances* request to the cloud controller, which may respond 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.

TABLE I. Description of VM types simulated in Koala

	Virtual Cores		Virtual Block Devices				
VM Type	#	Speed (GHz)	#	Size (GB) of Each	# Virtual Network Interfaces	Memory (GB)	Instruct. Arch.
M1 small	1	1.7	1	160	1	2	32-bit
M1 large	2	2	2	420	2	8	64-bit
M1 xlarge	4	2	4	420	2	16	64-bit
C1 medium	2	2.4	1	340	1	2	32-bit
C1 xlarge	8	2.4	4	420	2	8	64-bit
M2 xlarge	8	3	1	840	2	32	64-bit
M4 xlarge	8	3	2	850	2	64	64-bit

If VM instances are allocated, the user selects a holding time, Pareto distributed with a mean (4 hours here) and shape (1.2 here). During the holding period, the user will first issue *DescribeInstances* requests to determine when all instances are running, and will subsequently randomly describe, reboot, and terminate running instances. (We use the term *intermediate termination* to refer to subsets of instances terminated randomly by the user.) Upon failure of any of a user's random requests, the user may retry some number of times (0 to 3 uniformly distributed here) for individual instances.

At the end of the holding period, the user will issue a *TerminateInstances* request to stop any remaining running instances. We use the term *final termination* to refer to this request. If all instances cannot be terminated, the user may retry some number of times (0 to 3 uniformly distributed here) before giving up. When all instances are terminated, or retries exhausted, the user will wait an exponentially distributed time (mean 7.5 minutes here) and then start a new request cycle.

TABLE II. Description of selected simulated user types: processing users (PU), distributed modeling and simulation (MS) users, peer-to-peer (PS) users, Web service (WS) users, and data search (DS) users

User Type	Prob.	VM Type(s)	Max-Min VMs	Max-Max VMs	User Type	Prob.	VM Type(s)	Max-Min VMs	Max-Max VMs
PU1	0.20	M1 small	10	100	PS1	0.10	C1 medium	3	10
					PS2	0.01		10	50
PU3	0.01		100	500	WS1	0.15	M1 large M2 xlarge C1 xlarge	1	3
PU2	0.20	M1 large	10	100	WS2	0.07	M1 large M2 xlarge C1 xlarge	3	9
PU4	0.01		100	500	WS3	0.03	M1 large M2 xlarge C1 xlarge	9	12
MS1	0.10	M1 xlarge	10	100	DS1	0.10	M4 xlarge	10	100
MS3	0.01		100	500	DS2	0.01		100	500

If the cloud controller responds to a *RunInstances* request with a NERA, then the user waits an exponentially distributed time (mean 7.5 minutes here) before retrying the request. A user will retry a failed request over a random period (mean 2 hours here) before resting for a random period (mean 8 hours here). If a user request cannot be honored within a random number of rest periods (mean 4 here), then the user abandons the request and starts a new cycle.

To represent user errors in formulating *RunInstances* requests, we elected to have Koala simulate some probability ($5x10^{-3}$ here) that the user generates a request that is unrecognizable to the cloud controller. In such cases, the cloud controller returns a fault to the user, who must then retry the request using procedures already explained above.

B. Supply Layer

The supply layer consists of a number (20 here) of clusters that each manages a number (200 here) of nodes. Koala defines a fixed set of 22 possible platform configurations for nodes. Here we used only the four platform types shown in Table III. Upon creation each node manifests, with equal probability here, one of these four configurations. Nodes retain their established configurations for the duration of a simulation run. For an instance to be allocated to a node, available resources on the node must be sufficient for the requirements specified by the instance's VM type.

C. Resource Allocation Layer

Koala patterns resource allocation after Eucalyptus procedures, which involve two decisions: (1) on which cluster should the 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 makes good sense because inter-VM communications would be local to a single cluster. While Koala can simulate various resource allocation algorithms, here we elected to have Koala simulate algorithms implemented by Eucalyptus.

TABLE III. Description of selected platform types simulated in Koala

Platform Type	Prob.	Physical Cores		Memory (GB)	# Physical Disks by Size				# Network Interfaces	Instruct. Arch.
		#	Speed (GHz)		250 GB	500 GB	750 GB	1000 GB		
C8	0.25	2	2.4	32	0	3	0	0	1	64-bit
C14	0.25	4	3	64	0	4	0	3	2	64-bit
C18	0.25	8	3	128	0	0	4	3	4	64-bit
C22	0.25	16	3	256	0	0	0	7	4	64-bit

At the cluster level, we elected to have Koala simulate the Eucalyptus *first-fit* algorithm to choose nodes for VMs. First-fit simply searches the nodes by identifier from first to last until a node is found that can accommodate a given VM type. In making an accommodation decision, Eucalyptus compares resources required by a VM type against a node's availability of: (1) virtual cores, (2) disk space and (2) memory. We elected to have Koala simulate some probability (10^{-3} here) that a selected node develops a failure preventing it from accepting a VM that had appeared to fit. In such cases, the cluster controller *reallocates* the VM to the next node on the list. This process continues until the VM is created or until all nodes have been exhausted. If no nodes can create the VM, then the cloud controller receives a NERA fault.

At the cloud level, Eucalyptus can accommodate a choice of algorithms to select a cluster to which to assign all VMs in a request, but we elected to have Koala simulate the only algorithm that Eucalyptus actually implements. The implemented algorithm, called *least-full-first*, 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 appropriate 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.

D. Internet/Intranet Layer

Koala assigns the cloud controller, cluster controllers and users to *sites* (1000 here) randomly located at x,y coordinates on a grid (8000×8000 miles here). Before a simulation commences, cloud and cluster controllers are randomly placed on some number (4 here) of sites. Node controllers are placed on the same site as the related cluster controller. At the beginning of each user cycle, a user is assigned randomly to one of the (996 here) sites not occupied by cloud components. This arrangement divides message communications into two categories: (1) inter-site (Internet) and (2) intra-site (Intranet).

Koala components communicate through simulated Web Services (WS) messages, which each comprise a uniformly distributed number (1 to 10 here) of 1500-byte packets. Individual packets are subjected to transmission delay (1 Gigabits per second rate here) and propagation delay. For inter-site messages, propagation delay depends on distance and simulated router hops, while propagation delay within sites varies randomly (mean 250 nanoseconds here). Individual packets are also subjected to a loss rate (10^{-6} here for intra-site packets). To simulate Internet congestion, the loss rate for inter-site packets varies uniformly within a range (10^{-1} to 10^{-6} here). Lost packets are retransmitted, but only for a maximum number (3 here) of attempts, after which the related WS message is declared undeliverable.

Eucalyptus relies on an open-source Web server. As explained in Sec. I, software distribution sites can be subjected to insertion of Trojan code. Koala simulates a Trojan procedure that randomly discards arriving and departing WS messages. Below, in Sec. VI, we vary this discard probability over a wide range to gauge the effects of lost messages on an infrastructure cloud.

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 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 in Sec. III.*A*. 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 the VMs.

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 can be detected and removed automatically by the Eucalyptus protocol. 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 here) after boot up, the monitor declares the VM to be an orphan, terminates it, and releases the VM's resources for future 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 here). Once activated, the *cloud persistent terminator* resends termination requests to a cluster controller at specified intervals (90 s here) until one of the desired responses is received, or until a maximum termination period (2 h here) expires. After the first three retries, the cloud persistent terminator lengthens the retry period (to 150 s here) and then doubles it on each retry. If the maximum termination period expires, the cloud persistent terminator ceases and notifies an administrator that manual intervention is needed to terminate the orphaned VMs, and free related resources.

When activated, the *cluster persistent terminator* also attempts three retries to the node controller (every 90 s here) before increasing the retry interval in the manner described for the cloud persistent terminator. This process continues for a shorter maximum termination period (900 s here), since the retry encompasses only a single orphaned VM.

Since persistent termination adds complexity and overhead to the cloud and cluster controllers, we elected to limit persistent terminators to *TerminateInstances* requests associated with final terminations, issued by users to liquidate all allocated VMs, rather than to intermediate termination requests directed at a subset of allocated VMs. This decision limits proliferation of termination processes, which might otherwise require substantial additional computation and communication. However, this decision also means that lost messages related to intermediate terminations can allow affected VMs to persist until a final termination request succeeds. We refer to such affected VMs as *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 associated with orphan control and how might they affect performance as the rate of orphan creation 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.*C*. 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 simulation execution, we measured system performance at intervals of 1 h using metrics discussed below.

VI. Results and Discussion

In each of the 24 cloud simulations, we counted the number of VMs held by both users and node controllers at the end of the 1000–hour simulated period and the number of orphans created.

A. The Effects of VM Leakage and Orphan Control

With no orphan control, Fig. 2 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 8000 more VMs than known to users, which know of just over 6000 VMs.

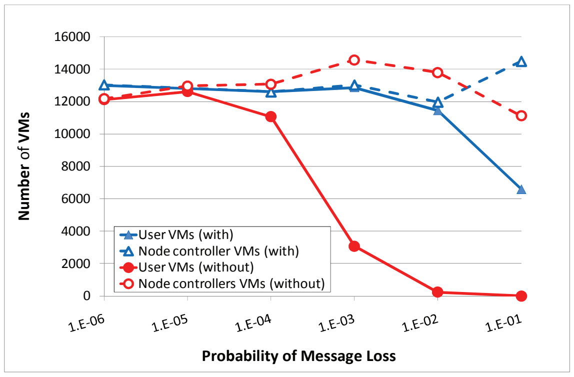


Figure 2. Number of VMs held by users and node controllers with (blue) and without (red) creation orphan control and persistent termination at the end of the1000-hour simulated period as the message loss rate increases.

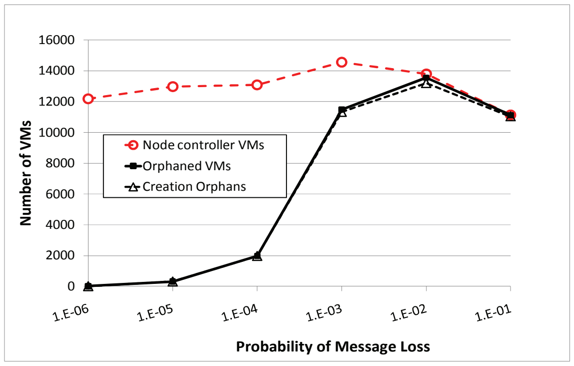
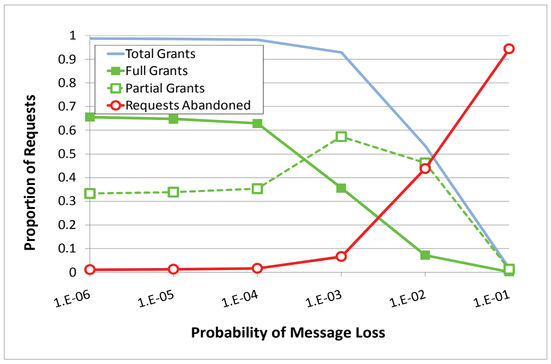


Figure 3. Number of VMs held by node controllers and number of orphans at the end of the 1000-hour simulated interval as the rate of message loss increases. Counts are plotted for the case without persistent termination and creation orphan control.

Figure 3 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. 3 also shows that almost all of the leaked VMs arise from creation orphans. We say more about this below.

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 grants and partial grants. We also recorded the proportion of users not granted VMs, users who subsequently abandoned the request process.

Figure 4(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. 4(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. 4(a), is attributable to resource exhaustion due to orphan VMs. This conclusion is supported by more detailed analysis below.

(a)


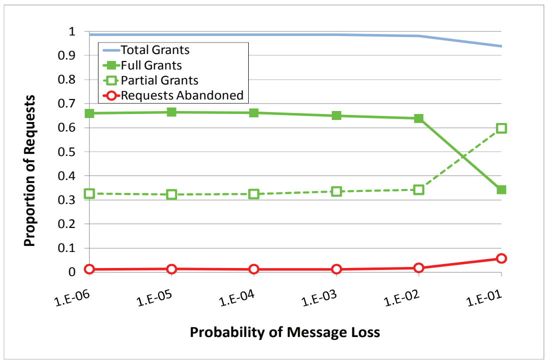
(b)


Figure 4. Disposition of user requests (a) without orphan control and (b) with creation orphan control and persistent termination.

The results described so far do not mean that the orphan control procedures we designed will free a cloud of all effects from VM leakage. Figure 4(b) also shows that even with orphan control operating, 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 become more likely. This change can be related to Fig. 2, 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 cover 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. 2 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.

Recall that in Fig. 3 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 (within 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. 4(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.

B. Analysis at the Resource Allocation and Supply Layers

To assess the internal operation of the cloud in our experiment, we observed processes in the resource allocation and supply layers. Our analysis of these processes supports the conclusions reached in Sec. VI.*A*. Figure 5 shows, both with and without orphan control, the number of user *RunInstances* requests received by the cloud and the number of NERA responses as the message loss rate increases. 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. 5 shows increasing loss rate leads to only a modest rise in requests, most of which are granted.

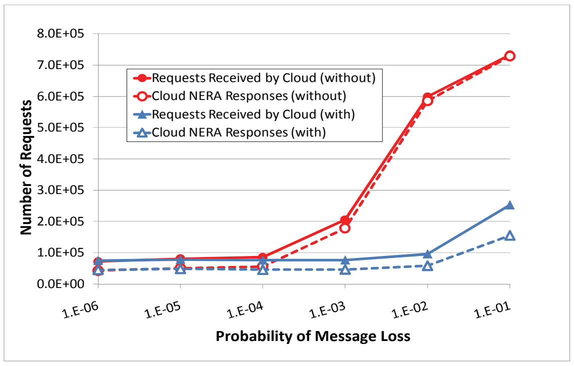


Figure 5. User requests received by cloud controller and NERA responses as the message loss rate increases, with (blue) and without (red) creation orphan control and persistent termination.

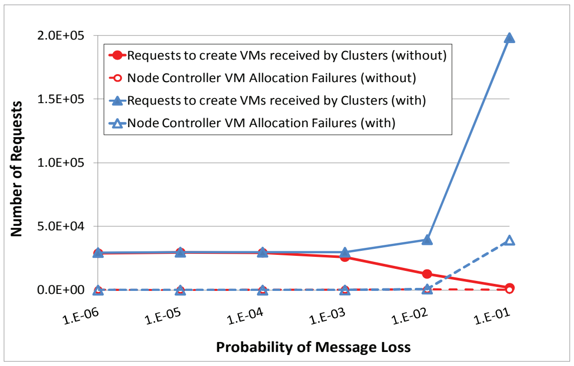


Figure 6. Allocation requests received by cluster controllers, along with node controller failures to allocate individual VMs, as the message loss rate increases, with (blue) and without (red) creation orphan control and persistent terminations.

Delving more deeply, Fig. 6 shows that cloud controller requests to clusters to create VMs remain roughly constant, until the probability of message loss exceeds 10^{-4}, whether or not orphan control is in force. Above this rate and without orphan control, the number of requests to clusters for VMs falls to nearly zero, which again reflects the inability of the cloud to find clusters that can satisfy incoming requests, due to the fact that resources are now almost completely exhausted by VM leakage. On the other hand, with orphan control, Fig. 6 shows that cloud controller requests to clusters to create VMs rises with increasing message loss. Since orphan control recovers computing resources held by orphans, cloud controller requests to clusters can now succeed. However as the message loss rate rises, failed intermediate terminations result in temporary orphans, which occupy significant resources, and so, as Fig. 6 shows, at the highest loss rate node controllers reject most VM allocation requests. As a result, clusters then reject the related requests from cloud controllers to create VMs, forcing the cloud controller to search for another cluster. Though the actions of both orphan control processes mean that the cloud controller is likely to find a cluster that will accept a request, Fig. 4(b) shows there is a greater likelihood of a partial rather than full grant, because temporary orphans have reduced resource availability.

Finally, Fig. 7 shows the total number of messages sent across all layers in the cloud system as message loss rate increases. Without orphan control, the overall number of messages increases with loss rate. This reflects increased effort expended as users retry requests, causing the cloud to make failed allocation attempts, as VM leakage drains needed resources. Fig. 7 also shows a slight dip in messages sent at the highest loss rate. This is likely due to the increasing frequency with which users enter resting phases of their request cycle (as explained in Sec. III.*A*). This effect is also present in Fig. 2 and in Fig. 3. With orphan control, the smaller rise in message traffic reflects a greater effort needed to make successful allocations, as discussed previously. The inset shows that the increase in message traffic is only marginally due to messages (about 0.44 %) related to persistent termination (creation orphan control requires no messages).

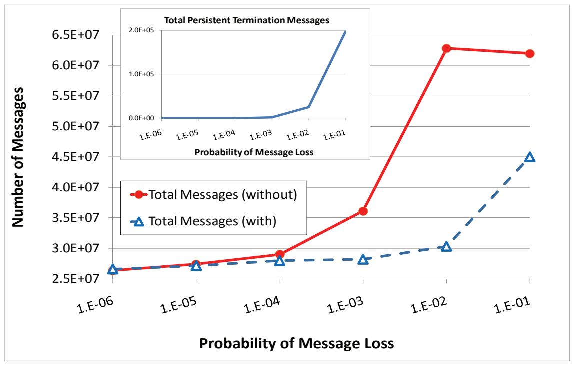


Figure 7. Total messages with (blue) and without (red) creation orphan control and persistent termination; compared with messages attributable to persistent termination (inset).

VII. Conclusions

This paper has addressed the potential problem of resource leakage in cloud systems and introduced the concepts of VM leakage and orphan VMs. The paper has demonstrated that VM leakage is a potentially serious vulnerability that can lead to resource exhaustion in open-source clouds. Using a scenario, in which a Trojan attack introduces malicious code modifications into one part of an open-source cloud implementation, we have shown 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 intended 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 design methods that more completely address the orphan control problem, such as, for instance, methods that extend persistent termination to temporary orphans and methods that curtail the onset of creation orphans. We believe that the work presented in this paper will aid designers and implementers in improving the reliability of open-source cloud systems.

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. A. Nuno, 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. Weber, P. A. Karger, and A. Paradkar, "A software flaw taxonomy: aiming tools at security", *Proceedings of the 2005 workshop on Software engineering for secure systems\—building trustworthy applications* (SESS '05). New York, NY, USA, 2005, pp. 1-7.

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

[10] 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.

[11] 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.

[12] 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/

[13] 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/

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

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

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

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

[18] IT World Canada Staff, Trojan Horse Attacks GNU Project, PC World, Aug. 18, 2003.

[19] Staff, Attacker attempts to plant Trojan in Linux, ZDNet UK, Nov. 7, 2003.

[20] R. Singel, Firefox Infects Vietnamese Users With Trojan Code, WIRED, May 7, 2008.

[21] T. Forenski, Open-source hacks - sneaky Skype Trojan code released, ZDNet, August 27, 2009.

[22] K. J. Higgins, Open-Source Project Server Hacked, Software Rigged With Backdoor Trojan, Dark Reading, Dec. 2, 2010.

[23] "Handle leak", Wikipedia, September 10, 2010. http://en.wikipedia.org/wiki/Handle_leak, accessed August 1, 2011.

[24] 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.

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

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

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

[28] 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.