Evaluating Scalability and Performance of a Security Management Solution in Large Virtualized Environments

Lishan Yang
College of William and Mary
Williamsburg, Virginia
lyang11@cs.wm.edu

Ludmila Cherkasova
HyTrust Inc
Mountain View, California
lucy.cherkasova@gmail.com

Rajeev Badgujar
HyTrust Inc
Mountain View, California
rbadgujar@hytrust.com

Jack Blancaflor
HyTrust Inc
Mountain View, California
jblancaflor@hytrust.com

Rahul Konde
HyTrust Inc
Mountain View, California
rkonde@hytrust.com

Jason Mills
HyTrust Inc
Mountain View, California
jmills@hytrust.com

Evgenia Smirni
College of William and Mary
Williamsburg, Virginia
esmirni@cs.wm.edu

ABSTRACT

Virtualized infrastructure is a key capability of modern enterprise data centers and cloud computing, enabling a more agile and dynamic IT infrastructure with fast IT provisioning, simplified, automated management, and flexible resource allocation to handle a broad set of workloads. However, at the same time, virtualization introduces new challenges, since securing virtual servers is more difficult than physical machines. HyTrust Inc. has developed an innovative security solution, called HyTrust Cloud Control (HTCC), to mitigate risks associated with virtualization and cloud technologies. HTCC is a virtual appliance deployed as a transparent proxy in front of a VMware-based virtualized environment. Since HTCC serves as a gateway to a customer virtualized environment, it is important to carefully assess its performance and scalability as well as provide its accurate resource sizing. In this work[1], we introduce a novel approach for accomplishing this goal. First, we describe a special framework, based on a *nested virtualization* technique, which enables the creation and deployment of a large-scale virtualized environment (with 30,000 VMs) using a limited number of physical servers (4 servers in our experiments). Second, we introduce a design and implementation of a novel, extensible benchmark, called *HT-vmbench*, that allows to mimic the session-based activities of different system administrators and users in virtualized environments. The benchmark is implemented using VMware Web Service SDK. By executing *HT-vmbench* in the emulated large-scale virtualized environments, we can support an efficient performance assessment of management and security solutions (such as HTCC), their overhead, and provide capacity planning rules and resource sizing recommendations.

KEYWORDS

Benchmark; scalability; performance; virtualization

ACM Reference Format:
Lishan Yang, Ludmila Cherkasova, Rajeev Badgujar, Jack Blancaflor, Rahul Konde, Jason Mills, and Evgenia Smirni. 2018. Evaluating Scalability and Performance of a Security Management Solution in Large Virtualized Environments. In *ICPE '18: ACM/SPEC International Conference on Performance Engineering, April 9–13, 2018, Berlin, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3184407.3184435

1 INTRODUCTION

Many companies adopt virtualization and its ability to slice larger, underutilized physical servers into smaller, virtual ones, to get significant cost savings resulting from server consolidation. Nowadays, in the era of large multi-core servers this approach became even more economically appealing to enterprise customers. Virtualization and cloud introduce a set of new challenges for reliably secure virtual servers and supporting additional authentication procedures across a large set of virtual machines in the enterprise environments. HyTrust Inc. developed a security solution, called HyTrust CloudControl (HTCC) [1–3], to mitigate the security risks and compliance gaps that exist in virtual and cloud environments. In essence, any action issued by a privileged user is proxied, evaluated, logged, and then forwarded to a vCenter (if approved). Since HTCC serves as a gateway to a customer virtualized environment, it is important to carefully assess its performance and scalability in order to minimize the introduced overhead and provide adequate resource sizing for managing and protecting customer's large-scale virtualized environment. To accomplish this goal there are two main challenges to be addressed:

- First, to assess the scalability of the HTCC solution, we need to emulate a large scale virtualized environment with tens of thousands of virtual machines. This is a real challenge since not every company or research organization has an access to a production size virtualized environment needed for such evaluation and performance experiments.

[1]This work was mostly completed during L. Yang's summer internship in 2017 at HyTrust Inc.
E. Smirni and L. Yang are partially supported by NSF grant CCF-1649087.

ICPE '18, April 9–13, 2018, Berlin, Germany

ACM ISBN 978-1-4503-5095-2/18/04…$15.00
https://doi.org/10.1145/3184407.3184435

- Second, we need to generate a variety of typical customer workloads to drive the performance and scalability assessment of HTCC in a large-scale virtualized environment. Currently, we are not aware of any available benchmark which is representative of system administrators' actions and users' activities typically performed in virtualized environments.

In this work, we introduce our solution for solving both challenges:

- First, we describe an approach, based on a *nested virtualization* technique [18, 19], which enables us to create a large scale virtualized environment (with 30,000 VMs) using a limited number of physical servers (4 servers in our experiments). Nested virtualization is the ability to run a hypervisor inside a virtual machine. This enables a recursive-based approach for creating a high number of additional "slim" VMs inside an original virtual machine deployed on a bare metal hypervisor. This allows deploying a large-scale virtualized environment without the need to actually have a large number of dedicated physical machines.
- Second, we design and implement a novel, extensible benchmark, called *HT-vmbench*, which allows us to mimic *session-based* activities of different system administrators and users in virtualized environments. The benchmark is implemented using VMware Web Service SDK [20]. *HT-vmbench* allows issuing a set of typical operations in virtualized environments. These operations have strong interdependencies and causalities. In addition, almost all the VM operations do take seconds (i.e., they are far from being "instantaneous"), and one needs to be very careful in avoiding the possibility of introducing *race conditions*, i.e., when an operation over a VM is issued while the previous operation over the same VM is not yet finished. The proposed benchmark design and implementation carefully considers such challenges and provides means to correctly handle them.

By executing *HT-vmbench*, on an emulated large scale virtualized environment, we can measure the latency of performed operations as well as the benchmark throughput, and estimate the overhead introduced by the HTCC proxy.

This paper is organized as follows. Section 2 provides background on HTCC. Section 3 describes nested virtualization for creating large-scale virtualized environment using a limited set of physical resources. Section 4 introduces the design of *HT-vmbench* and discusses its main features. Section 5 evaluates the proposed framework by using measurement and a variety of performance experiments. Section 6 outlines related work. Section 7 summarizes our contribution and gives directions for future work. For more details on the architecture of *HT-vmbench*, we direct the interested reader to the extended version of this paper [23].

2 BACKGROUND ON HYTRUST CLOUD CONTROL SOLUTION

As companies continue to embrace virtualization and cloud by migrating from physical to virtualized environments, they are under increased pressure to secure user and corporate data and follow multiple compliance regulations (many of them are country and industry specific). With tighter control and higher penalties, the new compliance laws in US and EU [3] enforce stricter requirements for data protection of private and sensitive data.

In a traditional data center with physical servers, there is a well understood set of demarcation lines for managing different resources, e.g., server's OS and storage within one physical server is managed by a system administrator while the network is managed by a networking specialist. Similarly, many security policies rely on the separation of duties as well as physical barriers.

Virtualization and cloud have blurred these boundaries by merging many IT roles and functions during provisioning and configuration tasks over virtualized physical infrastructure. Moreover, workload consolidation results in a higher density of information assets with very different security requirements and data sensitivity levels. All these challenges reduce the ability of security administrators to adequately monitor and audit workloads for compliance and security management in virtualized environments.

HyTrust Cloud Control Solution (HTCC) was purposely designed from ground up to address the security and compliance gaps existing in virtual environments. The HTCC solution provides a powerful set of policy-based access controls as well as an automated enforcement of industry specific compliance templates. HTCC is a virtual appliance deployed as a transparent proxy in front of a VMware-based virtualized environment (vSphere) as shown in Figure 1.

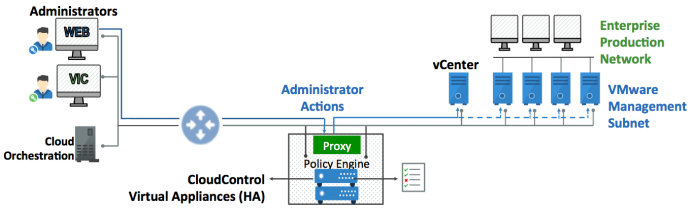

Figure 1: vSphere Architecture with HTCC Appliance.

HTCC transparent proxy allows a single-entry gateway to a protected virtual environment. It offers a non-intrusive security control to all administrators and users actions. In essence, any action issued by a privileged user (through any of VMware management tools) is proxied, evaluated against the specified access rules, logged for the audit trail, and then forwarded to vCenter (only if it was approved).

The HTCC transparent proxy acts as a central point of control: it seamlessly monitors and intercepts all the administrative requests, originated from a variety of possible access mechanisms in VMware vSphere as shown in Figure 2. The access mechanisms include vSphere Client and vSphere Web Client to vCenter, as well as the direct SSH user sessions to ESXi hosts.

The HTCC transparent proxy relies on the support of the following five key components as shown in Figure 2:

- *HTCC Policy Engine* is the main enforcement mechanism which allows to define Role Based Access Control in the organization among its privileged users.
- *HTCC Authentication Engine* supports an enhanced security layer along with the usual username and passwords.
- *HTCC Inventory Engine* supports the up to date "map" (inventory) of the protected virtual infrastructure under control. It is a critical component since VMs can be migrated, and the entire infrastructure can be changed with just a few clicks, e.g., adding a new vCenter.
- *HTCC Compliance Engine* provides and enforces the compliances templates for various industry and government standards, e.g., HIPPA, DISA, PCI-DSS.

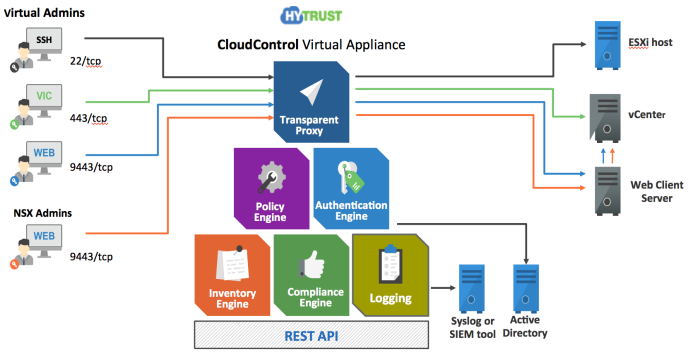

Figure 2: HyTrust CloudControl Architecture.

- *HTCC Logging Engine* supports a complete audit trail of privileged users' activities. The HTCC's logging engine also captures the attempted actions that have been denied by security policy.

Since the HTCC proxy acts as a main gateway to protected virtualized and cloud environments and it controls and intercepts all the administrative requests as shown in Figure 2, it is important to measure and evaluate the performance overhead introduced by the HTCC proxy. In order to answer the scalability questions, i.e., evaluate HTCC proxy overhead for different sizes of the protected environment, we should be able to efficiently emulate these large-scale virtualized environments. In the next section, we describe our approach for accomplishing this goal.

3 USING NESTED VIRTUALIZATION FOR EMULATING LARGE-SCALE VIRTUAL ENVIRONMENTS

Nested virtualization is a novel virtualization technique first proposed by IBM, which offers an opportunity to run a hypervisor inside a virtual machine. It has actively evolved over the last decade and is available on many virtualization platforms, such as Xen, VMware, and KVM. The general theory behind the nested virtualization, its implementation and performance characteristics are described in more detail in [7].

Applied to VMware solutions, the nested ESXi involves running ESXi on top of ESXi within a virtual machine. As shown in Figure 3, a typical deployment of ESXi and virtual machines consists of Layer 0 through Layer 2.

The hypervisor that runs on the real hardware is called Level 0 (or L0); the hypervisor that runs as a guest on L0 is called Level 1 (or L1); a guest that runs on the L1 hypervisor is called a Level 2 (or L2). In a nested ESXi deployment the third layer (Layer 3) is introduced by creating one level of recursion within the nested ESXi hypervisor at Layer 2.

With nested virtualization and its ability to run a hypervisor inside a virtual machine, we can apply a recursive-based approach for creating a high number of additional "slim" VMs inside an original virtual machine deployed on a bare metal hypervisor. Let us show how the scale of the deployed environment is defined (i.e., the numbers of deployed ESXI hosts and VMs) when we use the *traditional* approach for creating the virtual environment versus the *nested virtualization* approach. Let us consider a modern two

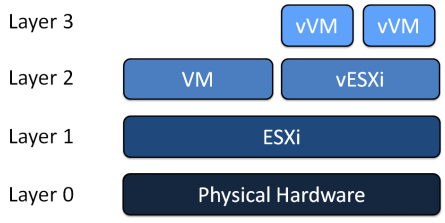

Figure 3: Nested Virtualization Approach.

socket, multi-core server (say, with 10 physical cores per socket, or 20 hyper-threaded virtual cores), with 256GB DRAM.

- Under the *traditional* approach with a ESXi hypervisor that runs on real hardware (at Layer 1), we can realistically deploy, say, 100 VMs per server at Layer 2. Therefore, if we need to create a virtual environment of 10,000 VMs, we would need to have 100 physical hosts under the *traditional* approach.
- Under the *nested virtualization* approach, we could recursively deploy inside of each VM at Layer 2, vESXi hypervisor (as shown in Figure 3). Then on top of each vESXi hypervisor, we can deploy 100 "slim" VMs at Layer 3. This way, we could get a large-scale virtual environment with 10,000 VMs deployed on a single physical host.

Following the described above logic, one can create a variety of different templates of large-scale environments for evaluating the scalability dimensions of the tested management solution under the test. Note, that similarly, we can deploy a number of vCenter appliances which manage the deployed ESXi hosts and VMs, and could be used as a gateway for created virtual environments.

4 HT-VMBENCH DESIGN AND IMPLEMENTATION

In this section, we present the design, main features, and implementation of *HT-vmbench*. First, we introduce the main terms and notations. Then we explain the idea and the formal algorithm of organizing VM operations as user sessions, and the execution of user sessions. Finally, we discuss the set of metrics reported at the end of the benchmark execution.

4.1 VM Operations and Interdependencies

A lifecycle of a VM starts with its creation and ends up with its deletion as shown in Figure 4. User can *create* a VM, then perform operations such as *clone*, *migrate*, *snapshot* before *delete* operation. In addition, *power_on* and *power_off* operations can happen on a created VM, and user can *reboot* a powered_on VM.

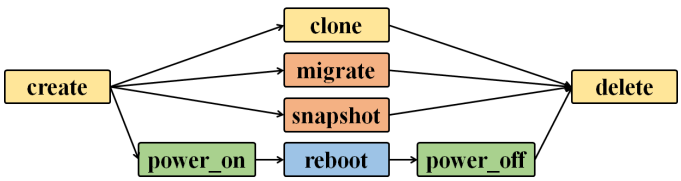

Figure 4: Lifecycle of a VM.

Some operations can be considered as *paired* operations (or *pairs*, for brevity), such as:

- (*create*, *delete*),
- (*clone*, *delete*),
- (*power_on*, *power_off*).

As operations paired together, every *delete* VM operation follows after a *create* or *clone* VM operation. Thus every *delete* VM operation has an existing VM to be performed on. Similar, for *power_on* and *power_off*, with the concept of paired operations, every *power_off* VM operation has a powered_on VM to execute on.

For the remaining operations, *snapshot*, *migrate* and *reboot*, they are regarded as *single* operations. To make things more structured, we introduce the concept of *operation units*. An operation unit is either a *single* operation or a *pair* of operations. Some operation units may require preconditions:

- The *paired* operation (*create*, *delete*) can be executed if there are enough resources available for creating a new VM;
- The *paired* operations (*clone*, *delete*), (*power_on*, *power_off*), and the *single* operations *migrate* and *snapshot* require a precondition that there is at least one VM existing in the system;
- The *reboot* operation requires a precondition that there is at least one VM powered on to execute.

All the *paired* operations change the system state. We use two counters *Num_VM* and *Num_VM_Powered_On* to represent the system state:

- *Num_VM:* the number of VMs existing in the system. *Create*, *clone* and *delete* operations change *Num_VM*;
- *Num_VM_Powered_On:* the number of powered_on VMs. It can be changed by *power_on* and *power_off* operations.

With these two counters, all these preconditions can be easily satisfied and implemented.

4.2 User Sessions

The goal of *HT-vmbench* is to simulate VM operations performed by administrators on managed virtualized environments. We introduce the concept of a *user session*, a sequence of VM operations issued by an administrator. Each user session follows a *close-loop* model, i.e., the next request is issued only when the response to the previous request is received by the user. To reflect the ordering of VM operations, we use a pseudo-timeline. The pseudo-timestamps are numbers generated randomly, for the purpose of operation ordering, i.e., for defining which VM operation should be executed before the other one.

Every user session has its own counters *Num_VM* and *Num_VM_Powered_On*. To construct a user session, *HT-vmbench* organizes VM operations as operation units, and inserts them into the user session's timeline according to the interdependencies between the operations.

4.3 Transaction Mix

A transaction mix determines the percentage and combination of operations in a single benchmark run. User sessions should follow the specified transaction mix. A *well-formed* mix should satisfy the following constraints:

- $Num\_Create + Num\_Clone = Num\_Delete$;
- $Num\_Power\_On = Num\_Power\_Off$;
- If $Num\_Reboot > 0$, then $Num\_Power\_On > 0$.

Although we define $Num\_Create + Num\_Clone = Num\_Delete$, our design still supports $Num\_Create + Num\_Clone \geq Num\_Delete$. This relaxation of restriction allows the creation of special "test" transaction mixes, such as 100% *create* VM operations. These "test" cases are executed with an additional *clean-up* phase, implemented in the benchmark, to meet the close-loop benchmark design.

4.4 User Session Creation and Execution

This section discusses in detail the algorithm of the *user session* creation, and the execution of user sessions.

4.4.1 Priority. Every operation unit has its own priority. Priorities are assigned according to interdependencies between operations, shown in Table 1 (a smaller number means a higher priority):

Table 1: Priorities of Operation Units

Operation Unit	Priority
create & delete	1
power_on & power_off	2
clone & delete	3
migrate	4
snapshot	5
reboot	6

- The *create & delete* pair has the highest priority, since they start and end a VM's lifecycle.
- The *power_on* and *power_off* pair has the second highest priority. Note, because of the significant resource pressure introduced by *power_on* operation, we force the *power_on & power_off* pairs to be inserted as neighbours, i.e., there is no other *power_on* or *power_off* VM operations between a pair of *power_on & power_off*. By putting *power_on* and *power_off* as neighbours, the randomness of insertion is decreased. To remain as much randomness as possible, *power_on* and *power_off* pairs are given the second highest priority.
- The *clone & delete* pair has the third highest priority, because they also change counter *Num_VM*.
- The other operations such as *migrate*, *snapshot*, and *reboot* VM operations have the lowest priorities.

4.4.2 Algorithm for user session creation. Algorithm 1 below shows the pseudo-code of constructing user sessions. To explain the algorithm, consider the following example. Let the transaction mix be as shown in the second column of Table 2, with the number of operations $Num\_Ops = 24$, and the number of user sessions $Num\_User\_Session = 2$. First, all operations in the transaction mix are evenly distributed among the specified number of user sessions (Algorithm 1, Line 2). In our example, the two specified user sessions have the same subset of operations, see the third column of Table 2.

Lines 3-6 (Algorithm 1) refer to the preprocessing process before inserting operations. For every user session, operations are paired according to the operation interdependencies, and each operation unit is assigned a priority based on Table 1. Then, we sort the operation units by priority, and get a list of operation units, as shown in Figure 5.

Next, we construct a sequence for every user session (Algorithm 1, Line 7). In our example with two user sessions, we first construct a sequence for user session A, and then in a similar way, for user session B. All operation units in user session A are inserted into the pseudo-timeline from high priority to low priority:

Algorithm 1 Construct a sequence for a user session

Input: *Transaction_Mix*; *Num_User_Session*; *Num_Ops*
Output: List of user sessions *User_Session_List*

```
1: function CREATE_USER_SESSIONS()
2:     evenly distribute operations to user sessions
3:     for all user session do
4:         pair the operations (create,delete), (clone,delete) and (power_on,power_off)
5:         assign priority to every operation unit
6:         sort operation units by priority to get Op_Unit_List
7:         CONSTRUCT_A_SEQUENCE()
```

Table 2: Example of a Transaction Mix of a Single Benchmark Run

Operation Type	Number of Operations	Number of Operations per User Session
create	6	3
delete	8	4
power_on	2	1
power_off	2	1
clone	2	1
migrate	2	1
reboot	2	1

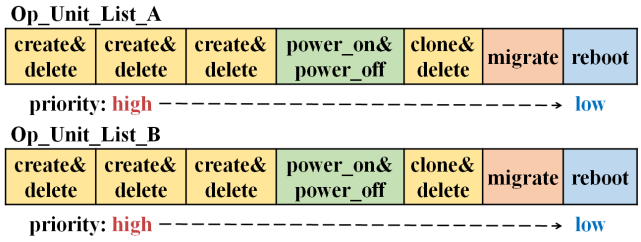

Figure 5: An Example of Operation Unit List.

- Referring to the list *Op_Unit_List_A* in Figure 5, the first operation unit is a pair of operations, *create* & *delete*. The *create* VM operation is given a random number representing the pseudo-time, and inserted into the timeline; then the *delete* VM operation is randomly inserted into the interval after the *create* VM operation, as shown in Figure 6(a). The counter *Num_VM* is calculated, shown in the Table below the timeline.
- Other two *create* and *delete* pairs are inserted in the same way, resulting in Figure 6(b). *Create* and *delete* pairs can be neighbours, as the ( ) pair and [ ] pair; also, *create* and *delete* pair can be nested, as the ( ) pair and { } pair. Each time when the pair is inserted, the counter *Num_VM* is refreshed.
- After inserting all *create* & *delete* pairs, the next step is the insertion of *power_on* & *power_off* pairs. The pair of *power_on* & *power_off* should be inserted as neighbours into the intervals, where *Num_VM* > 0 (as shown in Figure 6(c)). Counter *Num_VM_Powered_On* is calculated after each insertion.
- Figure 6(d) shows the insertion of *clone* & *delete* pair. First, we find some interval, where *Num_VM* is larger than 0, and insert *clone* VM operation into this interval; then *delete* VM operation is randomly inserted after the *clone* into the same interval. After each insertion, the counter *Num_VM* is updated.
- After the insertion of *clone* & *delete* pair, all the pairs are inserted into the timeline. The remaining single operations are a *migrate* VM operation and a *reboot* VM operation. The *migrate* VM operation can be inserted into any interval, where *Num_VM* > 0; the *reboot* VM operation can only be inserted into the intervals, where *Num_VM_Powered_On* > 0, i.e., between a pair of *power_on* & *power_off*. In this way, we get the timeline of user session A as shown in Figure 6(e).

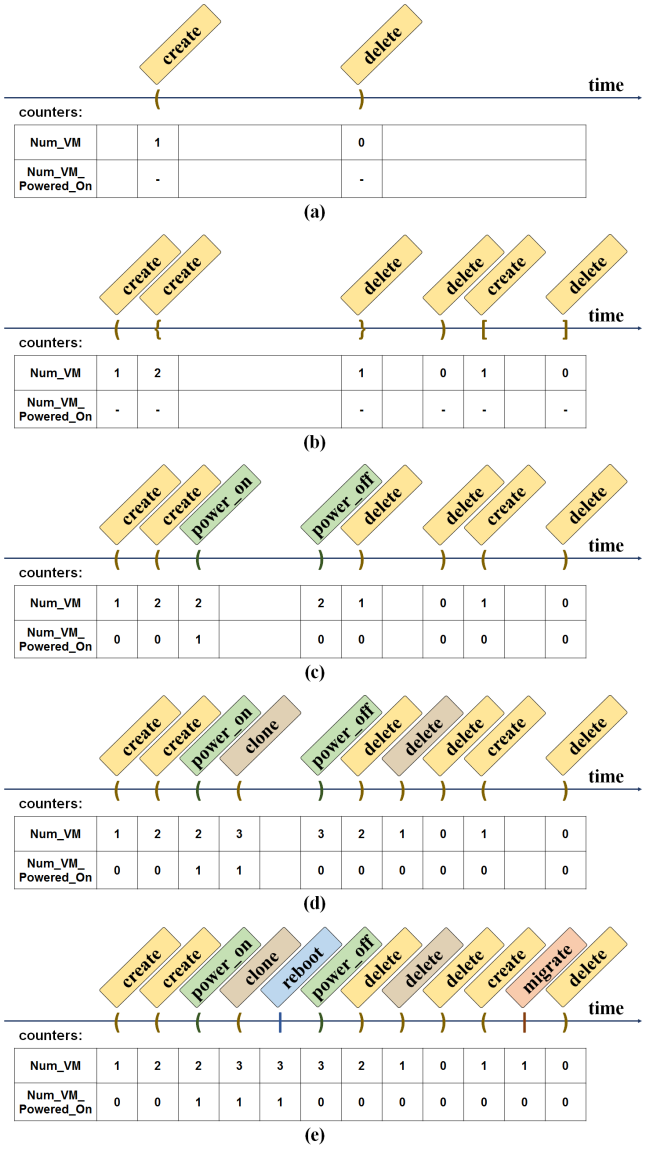

Figure 6: An Example of User Session Creation.

Table 3 (left column) shows a sequence of operations that corresponds to the constructed user session A. Similarly, we can construct a user session B, resulting in the sequence of operations shown in the right column of Table 3.

Table 3: Constructed User Sessions

User Session A	User Session B
create	*create*
create	*delete*
power_on	*create*
clone	*clone*
reboot	*power_on*
power_off	*migrate*
delete	*delete*
delete	*reboot*
delete	*power_off*
create	*delete*
migrate	*create*
delete	*delete*

The detailed pseudo-code of *Construct_a_sequence* is shown in Algorithm 2, with four basic insertion functions.

- *insert_create_delete_pair_op():* The *create* & *delete* pairs can be inserted randomly into any place in the sequence, with *delete* following after *create*. After insertion, the counter *Num_VM* is calculated and updated as a set of intervals *Exist_VM_Intervals_Set*, shown in the algorithm 2, Line 7.
- *insert_power_on_off_pair_op():* The *power_on* & *power_off* pairs are inserted to a randomly chosen interval, where the counter *Num_VM* is larger than 0 as neighbours. In other words, there is no operation between the *power_on* and *power_off* operations which come from the same pair. Similarly, counter *Num_VM_Powered_On* is refreshed after inserting *power_on* & *power_off* pairs.
- *insert_generic_pair_op():* This method inserts a generic pair to a proper position according to the requirement of preconditions. If the operation pair requires at least one VM powered on, the pair will be inserted to a randomly chosen interval where the counter *Num_VM_Powered_On* is larger than 0; if the operation pair requires at least one VM but not necessarily powered on, the pair will be inserted to a randomly chosen interval where the counter *Num_VM* is larger than 0; if the operation pair has no preconditions, the pair can be inserted to any position in the sequence.
- *insert_single_op():* Every single operation is inserted into a sequence according to the random number assigned under the requirements of preconditions.

Algorithm 2 takes the operation units from *Op_Unit_List*, checks the type of this operation unit, and finally calls the corresponding function to do the insertion. After inserting all the operation units, a sequence is constructed.

4.4.3 User Session Execution. After constructing all user sessions, each user session has a sequence of VM operations. Next, *HT-vmbench* starts the preparation of VM operation execution, including creating user session cache, loading vCenter information, and populating host information. Then, all user sessions are submitted to start their execution. Finally, after all the executions are finished, results are processed. There is an additional *clean-up* phase to clean up the remaining VMs created by the benchmark execution.

HT-vmbench gets the access to execute VM operations on the virtualized environment by invoking the APIs provided by *vSphere*. Every user session holds a close-loop model. Corresponding to the time spent by administrators to make decision between operations, we introduce *think time* between operations in a single user session, defined as the random variable *Think_Time*. To give the system maximum pressure, we set the think time to 0 as default.

When generating user sessions, we focus on the operation types (specified by the transaction mix) rather than specifying which

Algorithm 2 Construct a sequence for a user session

Input: List of operation units *Op_Unit_List*
Output: Sequence *S*

```
1:  function Construct_a_sequence()
2:      Exist_VM_Intervals_Set = null
3:      Powered_On_VM_Intervals_Set = null
4:      for each operation unit op_unit in Op_Unit_List, priority from high to low do
5:          if op_unit is Create_Delete_Pair_Op then
6:              insert_create_delete_pair_op()
7:              find all intervals of Num_VM > 0, and add to Exist_VM_Intervals_Set
8:          else if op_unit is Power_On_Off_Pair_Op then
9:              insert_power_on_off_pair_op()
10:             find all intervals of Num_VM_Powered_On > 0, and add to Powered_On_VM_Intervals_Set
11:         else if op_unit is Generic_Pair_Op then
12:             insert_generic_pair_op()
13:         else if op_unit is Single_Op then
14:             insert_single_op()
15: function insert_create_delete_pair_op()
16:     insert Create_Delete_Pair_Op to Sequence S
17: function insert_power_on_off_pair_op()
18:     randomly choose an interval (a, b) from Exist_VM_Intervals_Set
19:     insert Power_On_Off_Pair_Op to Sequence S between a and b as neighbours
20: function insert_generic_pair_op()
21:     if Generic_Pair_Op requires at least one VM powered on then
22:         randomly choose an interval (a, b) from Powered_On_VM_Intervals_Set
23:         insert Generic_Pair_Op to Sequence S between a and b
24:     else if Generic_Pair_Op requires at least one VM existing then
25:         randomly choose an interval (a, b) from Exist_VM_Intervals_Set
26:         insert Generic_Pair_Op to Sequence S between a and b
27:     else if no precondition then
28:         insert Generic_Pair_Op to Sequence S
29: function insert_single_op()
30:     if Single_Op requires at least one VM powered on then
31:         randomly choose an interval (a, b) from Powered_On_VM_Intervals_Set
32:         insert Single_Op to Sequence S between a and b
33:     else if Single_Op requires at least one VM existing then
34:         randomly choose an interval (a, b) from Exist_VM_Intervals_Set
35:         insert Single_Op to Sequence S between a and b
36:     else if no precondition then
37:         insert Single_Op to Sequence S
```

specific VM is going to be operated on. A VM operation may take seconds to finish, so if different user sessions submit VM operations performing on the same VM, a *race condition* occurs. To avoid these race conditions, a *blocking queue* is used to record all the VMs involved in the current state of the benchmark. When a VM operation is issued, a VM (in the required state) is dequeued from the blocking queue, and the VM operation is performed on this VM. After the operation is finished, the VM is enqueued into the blocking queue again. Note, the *create* VM operation creates a VM directly, and puts the new VM into the blocking queue; the *delete* VM operation gets a VM from the blocking queue and deletes this VM. This blocking queue ensures that there is always at most one operation performing on the same VM. This way the race conditions are avoided.

4.5 Benchmark Output and Metrics

To characterize the performance characteristics of a deployed virtualized environment as well as the performance of the HTCC proxy protecting it, the benchmark measures two sets of metrics: *latency* of performed operations (in seconds) and the overall system *throughput* in operations per second (*ops/sec*). Additionally, the benchmark collects and outputs *(i)* the detailed log of performed operations with measured latency of each operation and *(ii)* the summary output (computed from the detailed log) with the average latency per performed operation.

The execution of benchmark can be divided into three phases:

- *warm-up:* Before the slowest user session starts its execution;
- *main:* All the user sessions are executing their VM operations;
- *cool-down:* After the fastest user session finishes its execution.

The reported throughput and operation latencies are calculated in the *main* phase of the benchmark execution.

5 EVALUATION

The emulated large scale virtualized environment is deployed on four physical servers, each machine is DellC6320, with two sockets Intel Xeon E5-2640 v4, 2.4 GHz processors (Broadwell family), each processor with 10 physical cores (20 virtual processors, due to hyper-threading), i.e., 40 virtual processors per server, with 256 GB of RAM. Each server had 2 x 10Gb/s network.

We deployed (via nested virtualization) the VMware-based virtualized environment with two clusters: 10K and 20K VMs. Each cluster is configured with two vCenters. In such a way, by combining the resources of these clusters, we can perform experiments in large-scale virtualized environments having 10K, 20K, and 30K VMs.

By executing *HT-vmbench* on an emulated large scale virtualized environment, we measured the latency of performed operations. We performed a set of experiments using two types of environments:

- *virtual-original*: an emulated large scale VMware-based virtualized environment (with 10K, 20K, 30K VMs);
- *virtual-protected*: an emulated large scale VMware-based virtualized environment (with 10K, 20K, 30K VMs) protected by HTCC solution deployed as a gateway (transparent proxy) to the original virtualized environment.

We performed 500 operations for every user session during each *HT-vmbench* run, configured with the following transaction mix:

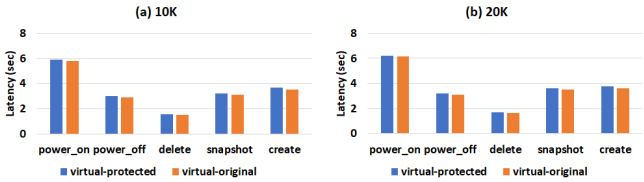

Figure 7: Average Latency of Different VM operations.

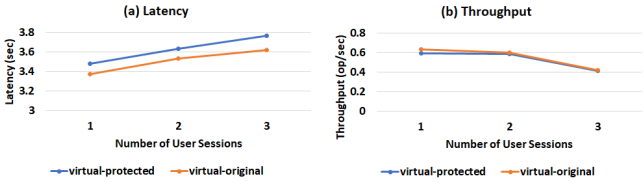

Figure 8: Average Latency and Throughput of VM operations in 10K VMs Environment.

- *create* =20%
- *delete* =20%
- *power_on* =20%
- *power_off* =20%
- *snapshot* =20%

Figure 7(a) shows the average latency of VM operations in *virtual-original* or *virtual-protected* environment of 10K object clusters with 3 user sessions. When running in *virtual-protected* environment, the execution time of operations is slightly higher than the execution time in *virtual-original* environment, due to the overhead introduced by HTCC. As results show, HTCC has a very reasonable performance overhead for most operations in virtualized environment tested by our benchmark.

Figure 7(b) shows the average latency in 20K object clusters with 3 user sessions, for both *virtual-original* and *virtual-protected* environment. These numbers are similar compared to the 10K cluster result shown in Figure 7(a). It indicates that the scale of virtualized environment does not have a significant impact on the performance overhead introduced by HTCC proxy.

Figure 8 shows the latency and throughput of VM operations as a function of the number of user sessions. As the number of user sessions grow, latency and throughput also increase. The latency is slightly higher, since the system is more crowded, and the throughput is increasing rapidly due to the paralellism.

By executing *HT-vmbench* in the emulated large-scale virtualized environments, we can support an efficient performance assessment of management and security solutions (such as HTCC), their overhead, and provide capacity planning rules and resource sizing recommendations.

6 RELATED WORK

Benchmarking Virtualized Environments: Over the last decade virtualization has gained popularity in enterprise and cloud environments as a software-based solution for creating shared hardware infrastructures. VMware released a *VMmark* benchmark [21] for quantifying the performance of virtualized environments. This benchmark aims to provide some basis for comparison of different hardware and virtualization platforms in *server consolidation* use cases, and therefore, aims to evaluate very different scenarios compared to *HT-vmbench*.

Application performance and resource consumption in virtualized environments might be quite different compared to their execution on bare metal hardware because of additional virtualization overheads, which are typically caused by I/O processing and the application interactions with the underlying virtual machine monitor (VMM). Different papers describe various VMM implementations and analyze virtualization overhead when executing specially selected microbenchmarks or macrobenchmarks (e.g., [6, 8, 12, 22]). The reported virtualization overhead greatly depends on the server hardware used in such experiments. This extensive body of previous benchmarks characterizes performance in virtualized environments from a very different angle compared to the goals and functionality of *HT-vmbench*.

Nested Virtualization Technique: During the last decade software virtualization solutions for x86 systems were broadly adopted, forcing both Intel and AMD to add virtualization extensions to their x86 platforms [5, 17]. There was a stream of efforts to incorporate nested virtualization in Xen hypervisor [4, 10]. Nested virtualization has many potential uses: e.g., platforms with hypervisors embedded in firmware might need to support other hypervisors as guest virtual machines. In the Cloud, IaaS providers might offer a user the ability to run the user-controlled hypervisor as a VM. In such a way, the user can manage his own virtual environment with with the choice of his favorite hypervisor. This might significantly simplify many management tasks, such as the live migration of their virtual machines as a single entity, e.g., for disaster recovery or load balancing. It could also be used for testing, demonstrating, benchmarking and debugging hypervisors and virtualization setups.

Nested virtualization enables new approaches to security in virtualized environments, such as honeypots capable of running hypervisor-level rootkits [14], hypervisor-level rootkit protection [13, 15], hypervisor-level intrusion detection [9, 11] for both hypervisors and operating systems. Nested virtualization is a foundation of the AERIE reference architecture [16]: it supports a set of components and services in a managed platform to reduce the level of trust required for IaaS providers. It helps to increase control and isolation and improve the system security and data protection.

In our work, we applied nested virtualization for creating a large scale virtualized environment using a limited number of physical servers to perform scalability assessment of security management solution (HTCC). This approach is of interest to many startups, small companies, and research organizations, which might not have access to a production size virtual environment needed for their scalability studies and performance experiments.

7 CONCLUSIONS

Engineering teams face many challenges when they implement new management and security solutions in large-scale virtual environment. They need to assess performance and scalability of such management solutions, analyze their performance overheads, and perform solution's capacity planning and resource sizing. In this paper, we introduce a novel approach for accomplishing these performance goals. We offer an extensible benchmark, called *HT-vmbench*, which allows users to mimic *session-based* activities of system administrators in virtualized environments. To perform scalability studies with *HT-vmbench*, the users need access to large-scale testbeds (that mimic the production virtual environments). To solve this challenge, we describe and promote an approach, based on a nested virtualization technique, which enables us to create a large scale virtualized environment (with 30,000 VMs) using a limited number of physical servers (4 servers in our experiments).

We believe that more interesting opportunities are available for constructing specialized virtual environment using this approach. Combined with an extensible nature of *HT-vmbench*, the proposed framework offers a powerful solution for performance assessment of different management and security solutions in large-scale virtual environments.

8 ACKNOWLEDGMENT

The research presented in the paper has been partially supported by NSF grant CCF-1649087.

REFERENCES

[1] HyTrust Inc. Cloud Control Virtual and Private Cloud Security. https://www.hytrust.com/products/cloudcontrol/.
[2] HyTrust Inc. HyTrust Cloud Control: Security, Compliance and Control for Virtual Infrastructure. https://www.hytrust.com/uploads/HyTrust-CloudControl.pdf.
[3] HyTrust Inc. Protecting Sensitive Data and achieving compliance in a multi-cloud world. https://www.hytrust.com/uploads/Compliance-in-a-Multi-Cloud-World_WP.pdf.
[4] Nested Virtualization on Xen. https://wiki.xenproject.org/wiki/Nested_Virtualization_in_Xen.
[5] AMD. Secure Virtual Machine Architecture Reference Manual. https://www.mimuw.edu.pl/~vincent/lecture6/sources/amd-pacifica-specification.pdf.
[6] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the Art of Virtualization. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, 2003.
[7] M. Ben-Yehuda, M. D. Day, Z. Dubitzky, M. Factor, N. Har'El, A. Gordon, A. Liguori, O. Wasserman, and B.-A. Yassour. The Turtles Project: Design and Implementation of Nested Virtualization. In *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation*, OSDI'10, 2010.
[8] L. Cherkasova and R. Gardner. Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, 2005.
[9] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium*, 2003.
[10] Q. He. Nested Virtualization on Xen. In *Proceedings of Xen Summit Asia*, 2009.
[11] J.-C. HUANG, M. MONCHIERO, and Y. TURNER. Ally: OS-Transparent Packet Inspection Using Sequestered Cores . In *Proceedings of Seventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS)*, 2011.
[12] S. T. King, G. W. Dunlap, and P. M. Chen. Operating System Support for Virtual Machines. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '03, 2003.
[13] R. Riley, X. Jiang, and D. Xu. Guest-Transparent Prevention of Kernel Rootkits with VMM-Based Memory Shadowing. In *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection*, RAID '08, 2008.
[14] J. Rutkowska. Subverting Vista Kernel for Fun and Profit. In *Proceedings of SyScan'06 and Black Hat Briefings*, Aug, 2006.
[15] A. Seshadri, M. Luk, N. Qu, and A. Perrig. SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes. In *Proceedings of Twenty-first ACM SIGOPS Symposium on Operating Systems Principles*, SOSP '07, 2007.
[16] M. Shtern, B. Simmons, M. Smit, and M. Litoiu. An Architecture for Overlaying Private Clouds on Public Providers. In *Proceedings of 8th International Conference on Network and Service Management (CNSM)*, 2012.
[17] L. Smith, A. Kagi, F. C. Martins, G. Neiger, F. Leung, D. Rodgers, A. Santoni, S. Bennett, R. Uhlig, and A. Anderson. Intel virtualization technology. *Computer*, 38, 2005.
[18] VMware. Running Nested VMs | VMware Communities. https://communities.vmware.com/docs/DOC-8970.
[19] VMware. VMware vSphere 6.5 Nested Virtualization. http://jermsmit.com/vmware-vsphere-6-5-nested-virtualization-create-and-install-esxi-6-5/.
[20] VMware. VMware vSphere Web Services SDK Documentation. https://www.vmware.com/support/developer/vc-sdk/.
[21] VMware VMmark Virtualization Benchmark, http://www.vmware.com/products/vmmark.html.
[22] T. Wood, L. Cherkasova, K. Ozonat, and P. Shenoy. Profiling and Modeling Resource Usage of Virtualized Applications. In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware*, Middleware '08, 2008.
[23] L. Yang, L. Cherkasova, R. Badgujar, J. Blancaflor, R. Konde, J. Mills, and E. Smirni. Evaluating Scalability and Performance of a Security Management Solution in Large Virtualized Environments.