ATOM: Efficient Tracking, Monitoring and Orchestration of Cloud Resources

N.Vishnu Vardhan[1], Dr.M.Sreedevi[2]
M.C.A, M.Phil., PhD

***Abstract*- The emergence of Infrastructure as a Service framework brings new opportunities, which also accompanies with new challenges in auto scaling, resource allocation, and security. A fundamental challenge underpinning these problems is the continuous tracking and monitoring of resource usage in the system. In this paper, we present ATOM, an efficient and effective framework to automatically track, monitor, and orchestrate resource usage in an Infrastructure as a Service (IaaS) system that is widely used in cloud infrastructure. We use novel tracking method to continuously track important system usage metrics with low overhead, and develop a Principal Component Analysis (PCA) based approach to continuously monitor and automatically find anomalies based on the approximated tracking results. We show how to dynamically set the tracking threshold based on the detection results, and further, how to adjust tracking algorithm to ensure its optimality under dynamic workloads. We demonstrate the extensibility of ATOM through virtual machine (VM) clustering. Lastly, when potential anomalies are identified, we use introspection tools to perform memory forensics on VMs guided by analyzed results from tracking and monitoring to identify malicious behavior inside a VM. We evaluate the performance of our framework in an open source IaaS system.**

***Index Terms*- Infrastructure as a Service, cloud, tracking, monitoring, anomaly detection, virtual machine introspection.**

1. INTRODUCTION

The Infrastructure as a Service (IaaS) framework is a popular model in realizing cloud computing services. In this model, a cloud provider manages and outsources her computing resources through an IaaS system. For example, Amazon offers cloud service with its Elastic Compute Cloud (EC2) platform [1], which is an IaaS system. While IaaS is an attractive model, since it enables cloud providers to outsource their computing resources and cloud users to cut their cost on a pay-per-use basis, it has raised new challenges in auto scaling, resource allocation, and security.

For example, auto scaling in the IaaS framework is the process to automatically add and remove computing resources based upon the actual resource usage. Cloud users want to pay for more resources only when they need them, and to make the best use of their (paid) resources by evenly distributing their workloads. Auto scaling and load balancing, two critical services provided by Amazon Web Service (AWS) [1] and other IaaS platforms, are designed to address these issues. A critical module in achieving auto-scaling and load balancing is the ability to monitor resource usage from many virtual machines (VMs) running on top of EC2. In Amazon cloud, resource usage information needs to be collected and reported back to a cloud controller, not only for the cloud controller to make various administrative decisions, but also for cloud users to query.

Security is another paramount issue while using an IaaS system. For example, it was reported in late July 2014, adver- saries attacked Amazon cloud by installing distributed denial-of- service (DDoS) bots on user VMs by exploiting a vulnerability in Elastic search [2]. Resource usage data could provide critical insights to address security concerns. Thus, a cloud provider needs to constantly monitor resource usage and utilize these statistics not only for resource allocation, but also for anomaly detection in the system. Until now, the best practices for mitigating DDoS and other attacks in AWS include using Cloud Watch to create simple threshold alarms on monitored metrics and alert users for potential attacks [3]. In our work we show how to detect the anomalies automatically while saving users the trouble on setting magic threshold values.

These observations illustrate that a fundamental challenge underpinning several important problems in an IaaS system is the continuous tracking and

monitoring of resource usage in the system. Furthermore, several applications (e.g., security) also need intelligent and automated orchestration of system resources, by going beyond passive tracking and monitoring, and introducing auto-detection of abnormal behavior in the system, and active introspection and correction once anomaly has been identified and confirmed. This motivates us to design and implement ATOM, an efficient and effective framework to automatically track, orchestrate, and monitor resource usage in an IaaS system.

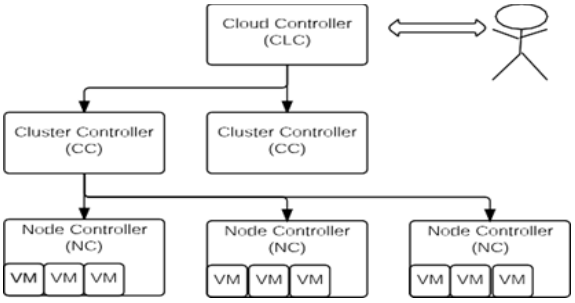


Fig. 2. A simplified architecture of Eucalyptus

A motivating example. Eucalyptus [4], [5] is an open source cloud software that provides AWS-compatible environment and interface. A simplified architecture of Eucalyptus, similar to other IaaS systems, is shown in figure 2. Cloud users interact with the cloud controller (CLC) to issue requests such as to allocate resources and query resource usage. CLC handles incoming user requests, collects information of the entire cloud, makes high-level decisions and controls other components such as cluster controller (CC) and node controller (NC). A CC forwards requests from the CLC to an NC, gathers status data on each NC, and reports back to the CLC. An NC controls the VMs running on it. One CLC controls several CCs and each CC could in turn control several NCs, on which multiple user VMs could be running. Note that only one CLC exists on each cloud.

Eucalyptus provides an AWS-like service called CloudWatch. CloudWatch is able to monitor resource usage of each VM. To reduce overhead, such data are only collected from each VM at every minute, and then reported to the CLC through a CC. Clearly, gathering resource usage in real time introduces overhead in the system (e.g., communication overhead from an NC to the CLC). When there are plenty of VMs to monitor, the problem becomes even worse and will bring significant overhead to the system. CloudWatch addresses this problem by collecting measurements only once every minute, but this provides only a discrete, sampled view of the system status and is not sufficient to providing continuous understanding and protection of the system.

Another limitation in existing approaches like CloudWatch is that they only do passive monitoring. No active online resource orchestration is in place towards detecting system anomalies, potential threats and attacks. We observe that, e.g., in the afore-mentioned DDoS attack to Amazon cloud, alarming signals can be learned from resource usage data. Active online resource mon- itoring and orchestration is very useful in achieving a more secure and reliable system. Active online resource monitoring gives us the opportunities to trigger VM introspection to debug the system and figure out what has possibly gone wrong. The introspection into VMs then allows to orchestrate resource usage and allocation in the IaaS system to achieve a more secure system and/or better performance. Note that VM introspection is expensive. Without continuous tracking and online monitoring and orchestration, it is almost impossible to figure out when to do VM introspection and what specific target to introspect in a host VM. Our goal is to automate this process and trigger VM introspection only when needed. We refer to this process as resource orchestration.

Our contribution. Motivated by these discussions, we present the ATOM framework. ATOM is an end-to-end framework that could be easily plugged into an IaaS system, to provide automated tracking, orchestration, and monitoring of resource usage for a potentially large number of VMs running on an IaaS cloud, in an online fashion.

ATOM introduces an online tracking module that runs at NC and continuously tracks various performance metrics and resource usage values of all VMs. The CLC is denoted as the tracker, and the NCs are denoted as the observers. The goal is to replace the sampled view at the CLC with a continuous understanding of system status, with minimum overhead.

ATOM then uses an automated monitoring module that con- tinuously monitors the resource usage data reported by the online tracking module. The goal is to detect anomaly by mining the resource usage data. This is especially helpful for detecting attacks that could cause changes in resource usage, for example,

one VM consumes all available resources and starves all other VMs running on the same physical computer [6]. The baseline for online monitoring is to simply define a threshold value for any metric of interest. Clearly, this approach is not very effective against dynamic and complex attacks and anomalies. ATOM uses a dynamic online monitoring method that is developed based on PCA. We design a PCA-based method that continuously analyzes the dominant subspace defined by the measurements from the tracking module, and automatically raises an alarm whenever a shift in the dominant subspace has been detected. Even though PCA-based methods have been used for anomaly detection in various contexts, a new challenge in our setting is to cope with approximate measurements produced by online tracking, and design methods that are able to automatically adapting to and adjusting the tracking errors.

Lastly, virtual machine introspection (VMI) is used to detect and identify malicious behavior inside a VM. VMI techniques such as analyzing VM memory space tends to be of great cost. If we don't know where and when an attack might have happened, we will need to go through the entire memory constantly, which is clearly expensive, especially if VMs to be analyzed are so many. ATOM provides two options here. The first option is to set a threshold for each resource usage measure (the baseline as discussed above), and we consider there may be an anomaly if the reported value is beyond (or below) the threshold for that measure and trigger a VMI. This is the method that existing systems like AWS and Eucalyptus have adopted for auto scaling tasks. The second option is to use the online monitoring method in the monitoring module to automatically detect anomaly and trigger a VMI, as well as guiding the introspection to specific regions in the VM memory space based on the data from online monitoring and tracking. We denote the second method as orchestration.

Comparison with UBL. UBL [7] stands for Unsupervised Behav-

ior Learning which is designed for monitoring virtualized cloud systems. It collects resource usage data from each VM, and trains Self-Organizing Maps (SOM) using normal data to predict future performance anomalies. UBL shows that SOM is an effective learning method for VM statistics and has better prediction ac- curacy compared with PCA/KNN in some experiments [7].

That said, note that ATOM is an end-to-end framework that integrates online tracking, online monitoring, and orchestration (for VM introspection) into one framework, whereas UBL focuses on anomaly detection in performance data without the integration of tracking and orchestration. Hence, UBL is "equivalent" to the monitoring component in ATOM.

More specifically, UBL can be plugged/integrated into ATOM's monitoring component as an alternative anomaly de- tection method to be more effective in capturing different types of anomaly. Note that PCA-based approach has the advantage of enabling us to analyze the theoretical bounds, when there are bounded tracking errors present in the continuously tracked measurements returned by the tracking component. UBL is more an empirical method which may perform really well on some instances, but it remains as an open problem to theoretically study its performance especially with approximate measurements when being used together with ATOM's tracking module. PCA- based approach also allows us to adjust the tracking threshold automatically in an online fashion by only adjusting the false alarm rate, as later shown in Section 5 where we have established the theoretical connection between the false alarm rate and the tracking threshold.

Lastly, SOM requires an explicit training stage and needs to be trained by normal data, while PCA identifies what is normal automatically and is able to adapt to the dynamic change from the underlying data. In contrast, SOM needs to be re-trained when the characteristics of the underlying workload has changed.

Paper organization. The rest of this paper is organized as follows.

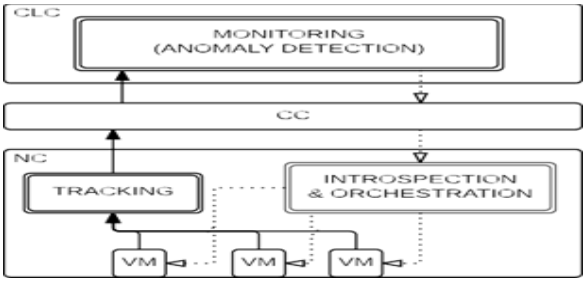


Fig. 3. The ATOM framework.

Section 2 gives an overview on the design of ATOM, and the threat model it considers. Sections 3 and 4 describe the online tracking and the online monitoring modules in ATOM. We further demonstrates the interaction between tracking component and monitoring component in section 5. Section 6 introduces the orchestration module. Section 7 shows an extension on VM clus- tering using the ATOM framework. Section 8 evaluates ATOM using Eucalyptus cloud and shows its effectiveness. Lastly, section 9 reviews the related work, and section 10 concludes the paper.

2. THE ATOM FRAMEWORK

Figure 3 shows the ATOM framework. For simplicity, only one CC and one NC are shown in this example. ATOM adds three components to an IaaS system like AWS and Eucalyptus:

(1) Tracking component: ATOM adapts the optimal online tracking algorithm for one-dimension online tracking inside the monitoring service on NCs. This dramatically reduces the over- head used to monitor cloud resources and enables continuous measurements to CC and CLC;

(2) Monitoring component (anomaly detection): ATOM adds this component in CLC to analyze tracking results by the tracking component, which provides continuous resource usage data in real time. It uses a modified PCA method to continuously track the divided subspace, as defined by the multi-dimensional values from the tracking results, and automatically detect anomaly by identi- fying notable shift in the interesting subspace. It also generates anomaly information for further analysis by the orchestration com- ponent when this happens. The monitoring component also adjusts the tracking threshold from the tracking component dynamically online based on the data trends and a desired false alarm rate.

(3) Orchestration component (introspection and debugging): when a potential anomaly is identified by the monitoring compo- nent, an INTROSPECT request along with anomaly information is sent to the orchestration component on NC, in which VMI tools (such as LibVMI [8]) and VM debugging tools (such as StackDB [9]) are used to identify the anomalous behavior inside a VM and raise an alarm to cloud users for further analysis.

In the following sections we investigate each component in further detail. Table 1 lists some frequently-used notations.

2.1 Threat Model

ATOM provides realtime tracking and monitoring on the usage of cloud resource in an IaaS system. It further goes out to detect and prevent attacks that could cause a notable change in resource usage from its typical subspace.

To that end, we need to formalize a threat model. We assume cloud users to be trustworthy, but they might accidentally run some malicious software out of ignorance. Also, despite various security rules and policies that are in place, it's still possible that

Symbol	Definition
Δ	tracking threshold
γ	finest resolution for floating point values
t	number of time instances in a sliding window
n	number of monitored VMs
dj	number of metrics for each VM
d	dj $\cdot$ n
M	data matrix ($t \times d$) of the most recent monitored data
avg j	mean of the j-th column in **M**
std $_j$	standard deviation of the j-th column in **M**
Y	standardized **M**, each value $y_{i,j} = (m_{i,j} - \text{avg } j)/\text{std}_j$
$tnow$	current time-stamp
A	consecutively abnormal data from $t_{now} - t$ to t_{now}
B	standardized A
z	the metric vector monitored at t_{now} (with d dimensions)
x	standardized **z**
$\mathbf{v}_i$	the i-th eigen vector output by PCA
λ_i	the i-th eigen value output by PCA
k	number of principal components output by PCA
α	input false alarm rate in PCA anomaly detection
Q_α	PCA anomaly detection threshold
μ	false alarm rate deviation, to control tracking threshold

TABLE 1 Frequently used notations

a smart attacker could bypass them and perform malicious tasks. The malicious behavior could very likely cause some change in resource usage. Note that, however, this is not necessarily always accompanied with more resource consumption! Some attacks could actually lead to less resource usage, or simply different ways of using the same amount of resources on average. All these attacks are targeted by the ATOM framework. The possibility of incorporating other types of attacks into ATOM is discussed in section 10.

3.TRACKING COMPONENT

This section introduces the tracking component in ATOM. Con- sider Eucalyptus CloudWatch as an example, which is an AWS CloudWatch compatible monitoring service that enables cloud users to monitor their cloud resources and make operational deci- sions based on the statistics. CloudWatch is capable of collecting, aggregating and dispensing data from resources such as VMs and storage. Cloud users can specify what they would like to monitor, and then query the history data for up to two weeks through the interface in the CLC. They can also set an alarm (essentially, a threshold) for a specific measure, and be notified or let it trigger some predefined action if the alarm conditions are met. Clearly, collecting such statistics continuously is expensive. Thus, the default in Eucalyptus and AWS is to ask an NC to only send measurements to the CLC at some predefined interval, e.g., once every minute in Eucalyptus.

A user VM in Eucalyptus is called an instance. In the follow- ing we will use the term “instance” and “VM” interchangeably. There are various variables that can be monitored overtime on each instance, each of which is called a metric. The measurement for each metric, for example, Percent for CPU Utilization, Count for Disk Read Ops and Disk Write Ops, Bytes for Disk Read Bytes, Disk Write Bytes, Network In and Network Out, is called Unit and is numerical.

A continuous understanding of these values is much more useful than a periodic, discrete sampled view that are only available, say, every minute. But doing so is expensive; an NC needs to constantly sending data to the CLC. A key observation is that, for most purposes, cloud users may not be interested in the exact value at every time instance. Thus, a continuous understanding of these values within some predefined error range is an appealing alternative. For example, it’s acceptable to learn that CPU Utilization is guaranteed to be within 3% of its exact value at any time instance.

This way NC only sends a value whenever the newest one is more than Δ away from last sent value on a measurement, where Δ is a user-specified, maximum allowed error on this measurement. CLC could use the last received value as an acceptable approx- imation for all values in-between. In practice, often time certain metrics on a VM do not change much over a long period. Thus far fewer values need to be sent to the CLC. Not only can we save the communication overhead from NC to the CLC, but also the database space on CLC used to store every value reported by NC (so that the history data could be kept for much longer than two weeks). Furthermore, instead of having only a sampled view at every minute, user now could query values at any time instance in the entire history that is available.

But unfortunately, this seemingly natural idea may perform very badly in practice. In fact, in the worst case, its asymptotic cost is infinite in terms of competitive ratio over the optimal offline algorithm that knows the entire data series in advance. For example, suppose the first value NC observes is 0 and then it oscillates between 0 and $\Delta + 1$. Then NC continues to send 0 and $\Delta + 1$ to the CLC. While the optimal offline algorithm who knows the entire f (t) at the beginning could send only one message to the CLC - the value Δ . Formally, this is known as the online tracking problem, which is formalized and studied in [10]. In online tracking, an observer observes a function f (t) in an online fashion, which means she sees f (t) for any time t before the current time (including the current time). A tracker would like to keep track of the current function value within some predefined error. The observer needs to decide when and what value she needs to send to the tracker so that the communication cost is minimized. Suppose function f : Z+ Z is the function observer observes overtime. g(t) stands for the value she chooses to send to the tracker at time t. The predefined error is Δ, which means at any time tnow, if the observer does not send a new value g(tnow) to the tracker, then it must satisfy f (tnow) g(tlast) Δ, where g(tlast) is the last value the tracker receives

from the observer. This is an online tracking over a one dimension positive integer function.

Instead of the naive algorithm that's shown above, Yi and Zhang provide an online algorithm that is proved to be optimal with a competitive ratio of only O(log Δ); that means in the worst case, its communication cost is only O(log Δ) times worse than the cost of the offline optimal algorithm that knows the function f (t) for entire time domain [10]. But unfortunately, the algorithm works only for integer values.

We observe that in reality, especially in our setting, real values (e.g., "double" for CPUUtilization) need to be tracked instead. To that end, we adapt the algorithm from [10], and design Algorithm

1 to track real values continuously in an online fashion. The algorithm performs in rounds. A round ends when S becomes an empty set, and a new round starts.

The central idea of our algorithm is to always send the median value from the range of possible valid values, denoted by S, whenever f (tnow) has changed more than Δ (could be non- integer) from g(tlast). The next key observation is that any real domain in a system must have a finite precision. Suppose γ is the finest resolution for the floating point values being tracked in the algorithm. Then at the beginning of each round, the number of possible values within S is 2Δ/γ, and since S is a finite set, it always becomes an empty set at some step following the above

Algorithm 1 One round of online tracking for real values

let $S = [f(t_{now}) - \Delta, f(t_{now}) + \Delta]$;
While $S_{upper\_bound} - S_{lower\_bound} > \gamma$ **do**
  $g(t_{now}) = (S_{upper\_bound} - S_{lower\_bound})/2$;
  send $g(t_{now})$ to tracker;
  Wait until $f(t_{now}) \quad g(t_{last}) > \Delta$;
  $S_{upper\_bound} = min\ (S_{upper\_bound}, f(t_{now}) + \Delta)$;
  $S_{lower\_bound} = max(S_{lower\_bound}, f(t_{now}) - \Delta)$;
end while /* this algorithm is run by observer */

algorithm. As long as S contains a finite number of elements in Algorithm 1, we can show its correctness and optimality with a competitive ratio of only O(log (Δ/γ)) for online tracking of real values.

Theorem 1. Algorithm 1 is correct and optimal for tracking double values, and has a competitive ratio of log (Δ/γ), where γ is the finest precision for floating point values.

Proof. Since γ is the finest resolution for the floating point values being tracked in the algorithm, then by multiplying every possible value in region S with integer 1/γ, all the values become integers. Therefore S becomes a region of integers, and all values we could choose to send to the tracker are integers. Now we could adapt the proof for tracking integers to prove the correctness and optimality of algorithm 1, and compute its competitive ratio. We denote the online algorithm as ASOL and the offline algorithm as AOPT .

Correctness. The correctness is obvious since Alice sends a value to Bob whenever the observation exceeds threshold Δ, and the value sent is within Δ of the observed value.

Competitive Ratio. The competitive ratio follows by two facts: In each round, i) ASOL sends at most log(Δ/γ) messages. This is because the cardinality of S decreases by half each time, and the initial range of S is 2Δ/γ. ii) AOPT sends at least one message. S is maintained as t [f (t) Δ, f (t) + Δ] for up to tnow in current round. If no value has been sent in this round, then the value (call it y) sent at the end of last round is within Δ range of all observations in current round, which makes y still lie in range S, a contradiction to the fact that S becomes empty in the end.

Optimality. The optimality holds because any online algorithm needs to send at least log(Δ/γ) messages in an extreme case. Suppose an adversary Carole operates function f . Whenever Alice sends some value to Bob, if the value is above the median of S, Carole decreases f until Alice sends a new value; otherwise Carole increases f until Alice sends a new one. This way the cardinality of S decreases at most half, so any online algorithm needs to send at least log(Δ/γ) messages. Whereas AOPT only needs to send out one value at the beginning of each round that's within the final S until the current round ends. In this case the lower bound of competitive ratio is log(Δ/γ). Hence the optimality of Algorithm 1 is proved.

The competitive ratio for algorithm 1 thus becomes O(log(Δ/γ)), which is optimal among all online tracking functions for floating point values.

In an IaaS system, an NC obtains the values for a metric of interest and acts as an observer for these values, and then chooses what to send to CLC by following Algorithm 1. The CLC, as the tracker, simply stores the values into its local database, whenever a value is reported from an NC. This is how ATOM's tracking module is able to save the network communication overhead from NC to CLC,

and the storage overhead in CLC. Note that the tracking algorithm is applied independently per dimension, meaning that the more VMs being tracked/monitored, the more savings ATOM will lead to, as evaluated in Section 8.4.

4.MONITORING COMPONENT

With the continuous tracked values of various metrics, compared with having only discrete, sampled views on these metrics, ATOM is able to do a much better job in monitoring system health and detecting anomalies.

To find anomalies in real-time, a naive method is to use the threshold approach. For example, Eucalyptus and AWS Cloud- Watch allow users to set an alarm along with an alarm action that can be triggered if the alarm condition is met. The alarm action is optional, which could be some predefined auto scaling policy such as changing disk capacity. The alarm condition consists of a threshold value T on a metric E of interest. The condition is met when the value v_τ from the metric E has exceeded T (or gone below T) at any time instance τ. However, in practice, it is very hard for cloud users to set a magic value as the threshold for a metric that will be effective in a dynamic environment like that in an IaaS system. Besides, it's inconvenient to change the threshold for each metric every time a user does some different tasks (which may invalidate the old threshold value). Thus an automated monitoring method would be very useful.

4.1An overview of PCA method

Given a data matrix in Rd , some dimensions in which are possibly correlated, the PCA method could transform this matrix into a new coordinate system, where each dimension is orthogonal. By mapping the original matrix onto the new coordinate system, we get a set of principal components. The first principal component points to the direction with the largest variance, and the following principal components each points to the largest variance direction that is orthogonal to all the previous ones. The intuition to use PCA as an anomaly detection method, is that the abnormal data points most likely do not fit into the correlation between each dimension in the original space. Thus by transforming the data matrix onto a new space using PCA, the original anomaly point would have a large projection length on the axes supposed to have very small variance (or so-called "residual subspace" in our following analysis). This way anomaly can be detected by analyzing the projection length onto these axes. A simple example when d = 2 is shown in Figure 4. PCA rotates the original coordinates into a new space, where the first axis points to the direction having the largest data variance while the remaining axis forms the residual subspace. The abnormal point is detected by comparing its projection length onto the residual subspace (second axis) against a threshold (detailed in Section 4.3.3). Using PCA for anomaly detection has been widely studied in the context of network traffic analysis and monitoring, e.g., [11], [12].

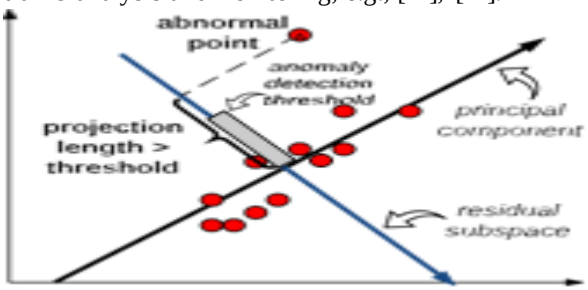


Fig. 4. An example of PCA anomaly detection in 2-dimensional space

To the best of our knowledge, there is no prior work in adapt- ing PCA for online monitoring and anomaly detection over VMs in an IaaS system. That said, there are three new challenges that we need to address: 1) unlike most existing work that use PCA for anomaly detection in an offline batch setting [12], ATOM needs to do online monitoring; 2) once anomaly is identified, ATOM needs to figure out which metrics from which VM instance(s) might have caused the anomaly; 3) the input data to ATOM's online monitoring module are approximate results from the tracking module, which have an error that is bounded by Δ. We need to take into account such tracking errors into the analysis. Next we will explain our method in detail.

4.2 The data matrix

Given dj metrics reported by the tracking module for each VM and t is the length of a time-based sliding window, PCA could be performed on these data which form a t dj matrix.

A more general and more interesting case is to perform online monitoring over a data matrix composed of multiple VMs' data, e.g., d = dj n

dimensions. For VMs hosted on the same physical node, or even the same cloud, it's quite possible that one VM may attack another [13], or some VMs are attacked by the same process simultaneously. Detecting anomaly on a d-dimensional space makes it easier to discover such correlations. It also provides better detection accuracy. Performing PCA on multiple VMs' statistics yields a higher residual dimension space, leading to more accurate anomaly detection.

Recall that ATOM's tracking module ensures that at any time point τ, for each metric E, CLC can obtain a value vjτ that is within vτ Δ, where vτ is the exact value of this metric at time τ from a VM instance of interest. Next we will show how to design an online PCA method to detect anomaly using a t d matrix M. Each data value in this matrix is guaranteed to be within Δ of the true exact value for the same metric at that same time instance.

4.3 Our approach

The following matrices are used in our construction: M, Y, A, B, whose definitions could be found in Table 1.

At first, a standard, offline batch PCA analysis [12] is applied to the data using the newest t time instances to find potential anomalies. If anomalies are found, we eliminate data correspond- ing to those time instances, and use the rest as the initial data matrix M to find the residual subspace S through a regular PCA analysis. Afterwards, for each z at tnow, we use the latest residual subspace S to perform anomaly detection.

In summary, our monitoring method has 5 steps: (1) process data from M to form Y; (2) build the PCA model based on Y;(3) find the residual subspace of the PCA model; (4) do anomaly detection for data at each new time instance using the latest PCA model; and if the newest time instance data z is normal, move it to M and update the PCA model; otherwise move it to A in case it doesn't agree with the residual subspace; (5) if z is abnormal, do metrics identification to find which metrics of which VM instances might have caused the anomaly. Step 1 is trivial by the definition of Y. The details of steps (2) to (5) are as follows.

4.3.1 Building the PCA model

To build the PCA model, we perform eigenvalue decomposition on the covariance matrix of Y, and get a set of eigen vectors V = (v1, v2, ..., vd) sorted by their eigen values. These eigen vectors form the new axes in the transformed coordinate system, with the first principal axis v1 pointing to the direction that has the largest variance in Y and the following principal axes each points to the largest variance direction orthogonal to previous ones. The corresponding eigen values are $\lambda 1 \geq \lambda 2 \geq ... \geq \lambda d \geq 0$.

4.3.2 Find the residual subspace

We define the principal subspace and the residual subspace as follows. The principal subspace S stands for the space spanned by the first several principal axes in V, while residual subspace S stands for the space spanned by the rest. The number of significant principal components in the principal subspace is denoted as k. Hence, the first k eigen vectors form the principal subspace, and the rest (d k) eigen vectors form the residual subspace that could be used to detect anomalies. Of numerous methods to determine k, we choose cumulative percent variance (CPV) method [14] for its ease of computation and good performance in practice as shown by previous work. For the first A principal components, CPV(A) $=\sum A\ \lambda i=1\ \lambda \bullet 1\ \lambda \cdot 100\%$, and we choose k to be: k = argmin(CPV (A) > 90%)

4.3.3 Anomaly detection

Unlike previous methods, e.g., [12], that perform offline, batched backbone network anomaly detection, we are not required to detect newest vector z at tnow. That's because we have classified data into the (normal) data matrix M and the abnormal matrix A, and the real-time detection of ongoing anomalies is based on the PCA model built from M To do this, we first standardize z using the mean and standard deviation of each column in M. We use x to denote the standard- ized vector.

Given the normal subspace S : P1 = [v1, ..., vk], and the residual subspace S : P2 = [vk+1, ..., vd], x is divided into two parts by being

projected on these two subspaces:

$x = x\hat{} + x\tilde{} = P1P1T\ x + P2P2T\ x.$

If z is normal, it should fit the distribution (e.g. mean and variance) of the normal data. Moreover, the values of $x\tilde{}$, which are the projection onto P2 by x, are supposed to be small. Specifically,

we define the squared prediction error (SPE) to quantify this:

SPE(x) = "x˜"2 = P2P2T x = (I − P1P1T) x

Let Q = x˜ 2, a classic result for the PCA model is that the following variable c approximately follows a standard normal distribution with zero mean and unit variance [15]:

$$c = \frac{\theta_1[(Q/\theta_1)^{h_0} - 1 - \theta_2 h_0(h_0 - 1)/\theta^2]}{2\theta_2 h_0^2}, \quad (1)$$

where $\theta = \sum_{j=k+1}^{d} \lambda_j^i \; i = 1, 2, 3; \; h = 1 - \frac{2\theta_1\theta_3}{3\theta_2^2}$

And we consider x to be *abnormal* if SPE(x) > Q_α, where the threshold Q_α is derived from the distribution c:

$$Q_\alpha = \theta_1\left[\frac{c_\alpha \sqrt{2\theta_2 h_0^2}}{\theta_1} + 1 + \frac{\theta_2 h_0(h_0 - 1)}{\theta_1^2}\right]^{\frac{1}{h_0}}$$

and cα is the (1-α) percentile in a standard normal distribution, with α being the false alarm rate.

Finally, if z is normal, we add it to M and matrix B. Matrices A and B need to contain time-consecutive data only (so that we detect anomaly corresponding to a continuous event), thus, they are cleared if its last vector is not consecutive in time with the new incoming vector.

4.3.4 Metrics identification

When an anomaly is detected, we need to do further analysis to identify which metrics on which VM instance(s) from the d = dj n dimensions might have caused the anomaly, to assist the orchestration module. Our identification method consists of three steps. It compares the abnormal data matrix A (and the corresponding standardized matrix B), and normal matrix M (and Y). Suppose there are m vectors in A (B) and t vectors in M (Y).

Step 1. Since the anomaly is detected by x˜ 2, it is natural to compare the residual data between B and Y. Suppose yi is the transpose of the i-th row vector in Y, and y˜i = P2PTyi is its residual traffic, then

$$(\tilde{y}_1, \tilde{y}_2, \ldots, \tilde{y}_t) = (P P_2 (y_1, y_2, \ldots, y_t)) = YP P_2$$

forms a residual matrix of Y, denoted as $\mathbf{Y_r}$. Similarly, $\mathbf{A_r} = \mathbf{AP_2P_2^T}$. For each dimension $j \in [1, d]$, let

$$a_j = \frac{1}{m}(\mathbf{A})_r \text{ and } y_j = \frac{1}{t}(\mathbf{Y_r})_j^2,$$

where $(\mathbf{A_r})_j$ is the j-th column in $\mathbf{A_r}$ and $(\mathbf{Y_r})_j$ the j-th column in $\mathbf{Y_r}$. Then $rd_j = (a_j - y_j)/y_j$.

where (Ar) j is the j-th column in Ar and (Yr) j the j-th column in Yr. Then rd j = (a j y j)/y j.

Step 2. If for some dimension j, rd j b1 for some constant b1, we measure the change in A and M. In particular, for each such dimension j, we calculate how much the abnormal data in A are away from the standard normal deviation of the normal data along that dimension in M. Specifically, we calculate stddev j = 1 ∑m 1 |ai j − avg j |/ std j. A dimension j is considered abnormal if stddev j b2 for some constant b2. In practice, we find that setting b1 and b2 to small positive integers works well, say b1 = 2 and b2 = 3.

Step 3. For a dimension j that's been considered abnormal in Step 2, we measure the difference between the mean of abnormal and normal data. Specifically, we want to measure mean diff j =(1 ∑m 1 ai j − avg j)/ avg j . m i=

Step 1 reveals which dimension has a larger projection on residual subspace than the normal data, however it is hard to map such change back to the original data. Furthermore, as shown in Section 8, this measure is not highly reliable and could be omitted to save some computation cost.

Step 2 is a useful measure to show which dimension has a significant different pattern compared to the normal data. However, it does not tell us whether some metric usage goes up or down. Thus we use step 3 at last to find this pattern.

Step 3 it self is not good enough to indicate a pattern, mean of some dimension in A appear benign. Thus, the output request, to the orchestration module on the corresponding NC(s), that administrates the identified VM instance(s). Section 8 shows how information identified from these three steps could facilitate the orchestration module to find a "real cause" of what might have gone wrong and how wrong it is.

Delete the oldest data in M, and update the PCA model accordingly. Otherwise it is added to A, and the corresponding standardized x is moved to

4.3.5 Other remarks

Raising alarms to cloud users. Once a data vector is detected as abnormal, it is moved to the abnormal data matrix, on which metrics identification is performed. Suppose there are totally m vectors in the abnormal data matrix A, an alarm will be raised with an alarm level m. The alarm level indicates how serious the detected anomaly is; intuitively, the larger number of data vectors contained in A, the longer duration of the currently detected anomaly is. The alarm can be raised either right after the metrics

identification step, or wait until the VMI (virtual machine inspection) from the orchestration module has finished (so that more information are gathered). The alarm notifies the user about the potential abnormal behavior in the IaaS system and lets user identify whether the ongoing behavior on his/her VM(s) is normal. If this is because that the tasks on a VM have changed, the corresponding data vectors in the abnormal matrix should be moved to the normal data matrix and used to build the PCA model to accommodate and reflect the new behavior. Abnormal data matrix is cleared once the anomaly on VM is removed, or is identified as normal by the cloud user.

Scalability.The computation complexity of monitoring module is evaluated in Section 8.4 (Figure 12). Although its computation cost increases with the increasing number of VMs, it remains as a very small overhead. The average computation cost per sliding window for the monitoring module is less than 3 milliseconds in message savings from ATOM's tracking module, both the PCA-icantly improve the detection accuracy, meaning smaller false alarm rates, which is due to the fact that the monitoring component uses a larger data matrix that helps find normal subspace more reliably, as also evaluated in Section 8.4 in Figure 12.

5. INTERACTION BETWEEN TRACKING AND MONI- TORING COMPONENTS

5.1Deriving the tracking error threshold

As mentioned earlier, the input data to the monitoring module is produced by the tracking module and each value may contain an approximate error of at most Δ (away from the true value at that time instance for that metric). The approximation error introduced by the tracking module may degrade the performance quality of ATOM's monitoring module. Thus, a formal analysis is needed to We made the observation that μ monotonically increases with ε. Hence the idea is to use a binary search to approximate ε: we first guess a value εj, then calculate a μj and compare it with the user-input μ, and finally adjust the value of εj and compute μj again. We repeat this process until the difference between μj and μ is within a desired precision. Then we could treat εj as ε, the input for the next step. The way to calculate μ using ε could be derived as follows. Given that c approximately follows a normal distribution, then $\mu = Pr[c\alpha\ \eta c < U < c\alpha + \eta c]$, where $\eta c = c\hat{}c$, and U is a random variable following the normal distribution N(0, 1). ηc could be approximated from ε using the Monte Carlo sampling technique according to equation 1. For each loop, we generate a random value $\lambda\hat{}$ in the range of $[\lambda\ \ \varepsilon, \lambda + \varepsilon]$ and then compute $c\hat{}$ based on equation 1, and compute the difference with c which is calculated by λ . This loop is repeated a constant number of times and the largest difference is assigned to ηc, which could be then used to calculate μ.

Step 2. Once having the eigen-error ε, using stochastic matrix perturbation method we could get the relation between eigen-error ε and the variance $\sigma 2$ along each dimension:

$$2\bar{\lambda}\sum_{i=1}^{d}\sigma_i^2 + (\frac{1}{t}+\frac{1}{d})\sum_{i=1}^{d}\sigma_i^4 = \varepsilon,$$

where $\bar{\lambda}$ is the average of eigen values, t is the number of points used to build the PCA model, and d is the number of dimensions. Then the estimation of tracking error Δ is based on the following assumptions:

1)the errors between the approximated values sent to the tracker (the CLC) and the true values observed by the observer (an NC) are independently and uniformly distributed within the threshold, accordin√g to which the tracking threshold for the i-th dimension is $\delta i = 3\sigma i$.

we use homogeneous slack allocation, which is to assume a uniform distribution of tracking error δ on each dimension.

Applying these two assumptions, we get a tracking threshold:

$$\delta = \frac{\sqrt{3\lambda n + 3\varepsilon\sqrt{m^2+mn}} - \sqrt{3\lambda n}}{\sqrt{m+n}} \qquad (2)$$

value as the error threshold Δ for each metric in the tracking module.

As shown in Section 4.3.3, the random variable c follows a normal distribution, and the SPE threshold Qα is computed after an α value is specified. However, we do not have c from the exact data matrix, instead, the approximate data matrix leads to the value $c\hat{}$. The SPE threshold is computed using a user-specified α value. However, the threshold

calculated by the approximated matrix does not represent confident limit 1 α anymore, instead it leads to a corresponding approximation 1 αˆ . We want to understand the relationship between αˆ and α. Formally, the cloud user specifies α and a maximally allowed deviation rate μ such that our tracking and monitoring methods guarantee that αˆ α μ (even though c is unknown). Thus, we need to establish the relationship between μ and the tracking error threshold Δ for each metric dimension used by the tracking module.

We achieve this objective via two steps: 1) given μ, find an approximate error bound ε on the average eigen values produced by PCA; 2) once having the error bound ε on eigen values, calculate the tracking threshold Δ based on ε.

Step 1. We could approximate μ according to ε from Equation 1, yet the reverse could not be done with a closed-form formula.

Note we cannot send this threshold directly to observers since the data matrix used to build the PCA model has been standardized. Recall stdi is the standard deviation along the i-th dimension of matrix M, then the original variance is Σi = (stdi σi)2. Thus, the tracking threshold for the i-th dimension is calculated as

Δi = 3Σi = 3(stdi σi)2 = stdi δi. The CLC calculates the results for each metric dimension whenever there is a PCA update, and then send the new tracking threshold to corresponding NCs (observers), which use the updated thresholds to adjust its tracking algorithm. A possible improvement is to allocate the tracking slack for each metric dimension according to the frequency of message passing sent to the CLC. By giving the dimensions being sent more frequently larger tracking error thresholds, and other dimensions smaller tracking error thresholds, the tracking overhead could be potentially further reduced.

5.2 Accommodating dynamic tracking thresholds

We can show that doing this style of "lazy update of the tracking threshold value" could ensure that the competitive ratio is the max of log Δ for all possible Δ (or log(Δ/γ) where γ is the finest precision for "double" values) in a tracking period; and it is optimal. It also guarantees that on the monitoring component, the PCA detection result calculated by the approximated tracking val- ues has a false alarm rate αˆ that is within user-specified deviation value μ of the true false alarm rate α (i.e., αˆ [α μ, α + μ]).

Claim 2. When the tracking threshold Δ changes at NC, by

simply changing the Δ value in Algorithm 1 during a round, the correctness and optimality of the tracking algorithm still hold. The competitive ratio with dynamically changing values of Δ becomes the log of the maximum Δ value for integers, and log of the maximum Δ/γ value for floating point values, where γ is the finest precision.

Proof. Here we prove for the case to track integer values. The extension to real values is straightforward following the proof for Claim 1. We use the same notation as in Section 3. Recall that a range S is initialized as [f (t0) Δ, f (t0) + Δ], where f (t0) is the value observed at first, and updated as the intersection of [f (t)

Δ, f (t) + Δ] up to tnow. A round is from the initialization of S until

S becomes empty.

Correctness. When the tracking error bound changes from Δ1 to Δ2, Alice sends Bob a new value whenever the newest value observed is beyond Δ2 range of last sent one.

Competitive Ratio. Note that in Algorithm 1, ASOL uses binary search, to guess what value AOPT might have sent in each round. The range S contains all the possible values that AOPT might have sent, and it decreases at least half upon the sending of each message (median of S). So that in each round, AOPT sends out only one value while ASOL sends out at most log Δ. Even if Δ changes in the middle, as shown in figure 5, it won't affect the fact that S decreases at least half upon each message sent. When the tracking error bound changes from Δ1 to Δ2, use S1 to denote the region of S at that time, and S2 to denote [y Δ2, y + Δ2]. x is the median of S1, the last sent value, and y is the first value observed that exceeds Δ2 of x after Δ changes. According to our "lazy update" method, the new S is the intersection of S1 and S2. Because y Δ2 > x, so new S = S1(upper bound) (y Δ2) < S1 /2. Hence no matter Δ2 is bigger or smaller than Δ1, S1 decreases at least half when

this change happens. If S1 and S2 do not intersect, then a new round starts and Δ2 becomes the initial threshold of the new round. Therefore, the competitive ratio for each round ONLY matters with

the initial size of S. If the initial threshold of a round is Δ, thento Δ2. So the optimality of algorithm 1 still holds even with the tracking error bound changing.

6. ORCHESTRATION COMPONENT

The monitoring component in Section 4 detects the abnormal state and identifies which measurement on which VM might be respon- sible. In this section, we describe how orchestration component is able to automatically mitigate the malicious behavior after an anomaly is detected.

Modern IaaS cloud vendors offer services mostly in the form of VMs, which makes it critical to ensure VM security in order to attract more customers. VMI technique has been widely studied to introspect VM for security purpose. There are also several popular open source general-purpose VMI tools such as LibVMI [8], Volatility [16], and StackDB [9], for researchers to explore and develop more sophisticated applications. LibVMI has many basic APIs that support memory read and write on live memory. Volatility itself supports memory forensics on a VM memory snapshot file, and it has many Linux plugins that are ready to use. StackDB is designed to be a multi-level debugger, while also serves well as a memory-forensics tool. Other more sophisticated techniques developed for special-purpose VMI anomaly detection are generally based on these tools. Blacksheep [17], for instance, utilizes Volatility and specifically developed plug-ins to imple- ment a distributed system for detecting anomalies inside VMs among groups of similar machines. However, as most other VMI strategies to secure VMs, it needs to dump the whole memory space of the target VM, and then analyze each piece, typically by comparing with what's defined a "normal" state. Thus to protect VMs in real time, the whole memory space needs to be analyzed constantly, introducing much overhead into the production system. ATOM implements its orchestration component based on Volatility (with LibVMI plug-in for live introspection) and StackDB. A crucial difference with other systems is that, ATOM only introspects the VM when an anomaly happens, and only on the relevant memory space of the suspicious VMs. The monitoring component in ATOM serves as a trigger to inform VMI tools when and where to do introspection. The anomalies are found by analyzing previously monitored resource usage data, in monitoring component, which is much more lightweight than analyzing the whole memory space. Then the metrics identification process in monitoring component could locate which dimensions are suspi- cious, indicating the relevant metrics on some particular VMs. This information is sent to orchestration component along witha VMI request, which then only introspects the relevant memory space, reducing the overhead dramatically. For example, if it is detected and identified that the network usages on VM-2 and VM- 3 are unusual, as shown in Figure 6, then ATOM could only introspects the network connections using Volatility network plug- ins on VM-2 and VM-3, in contrast to other VMI-based detection strategies which typically need to walk over the whole process list, opened network sockets, opened files, etc..

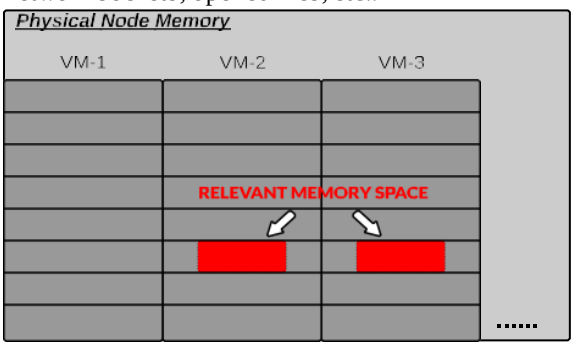


Fig. 6. Memory space introspected by ATOM.

After the orchestration component identifies potential abnor- mal processes, an alarm is raised with associated information identified by VMI tools. The alarm and such information are provided to the VM user. If user confirms this as an abnormal behavior, ATOM is able to terminate the malicious processes inside a VM instance by using tools like StackDB [9]. StackDB could be used to debug, inspect, modify, and analyze the behavior of running programs inside a VM instance. To kill a process, it first finds the task_struct object of the running process using process name or id, and then passes in SIGKILL signal. Next time the process is being scheduled, it is killed immediately.

Although the anomalies that could be detected by ATOM is limited compared with other systems which analyze the whole memory space, we argue the framework of ATOM could be easily extended to detect more complex attacks. First, more metrics could be easily added to monitor for each VM. Also,

many other auto- debugging tools could be developed, which are useful to find various kinds of attacks and perform different desirable actions.

Note that killing the identified, potentially malicious process is just one possible choice provided by ATOM, which is performed only if user agrees to (ATOM is certainly able to automate this monitoring modules for each cluster. For instance, a cloud provider may want to evenly distribute the VMs having similar resource usage patterns to different physical nodes, in order to make sure the physical resources are fully utilized and fewer VMs may suffer from resource starvation. In another example, we may want to use different anomaly detection techniques for VMs running a database server workload than those running a web server.

The basic idea of our proposed approach is as follows. The monitoring component in ATOM, using its PCA-based approach, transforms the original coordinates to a new coordinate system where the principal components (PCs) are ordered by the amount of variations on each direction (as explained in Figure 4). Thus, if two VMs share similar workloads, the directions of the corre- sponding PCs between the two should also be similar. That said,

Step 1. On CLC, a data matrix for each VM is maintained, where the columns are metric types and rows are time instances (i.e., a t dj matrix for each VM with a sliding window of t), and is updated over time.

Step 2. ATOM performs a PCA on each VM data matrix without standardization; since for clustering purposes, not only the variations on each direction is important, but also the average usage on each dimension. For example, a VM having a disk usage that oscillates between 10,000 and 20,000 bytes is obviously not the same as one having oscillation between 100 and 200 bytes on the same dimension; whereas a standardization procedure which first performs mean-center and then normalization will make the two oscillations look similar.

This step yields a set of PCs for each VM. The direction of each PC is denoted by the corresponding eigen vector while the variation is shown by the associated eigen value.

Step 3. Suppose VM1 has eigen vectors (v11, v12, ...) and corresponding eigen values (λ11, λ12, ...), while VM2 has (v21, v22, ...) and (λ21, λ22, ...). We measure the distance be- tween two directions using cosine distance; defined as (1 cosine similarity). Intuitively, the bigger the angle between two directions (the less similar they are), the smaller their cosine similarity is, hence the larger the cosine distance becomes. Finally, the distance between the two VMs is defined as: VMdist(VM1, VM2) = $|\lambda 11 - \lambda 21|(1 - v11 \bullet v21) + |\lambda 12 - \lambda 22|(1 -$as well if desired). Alternatives could be to terminate the networkv12 •v22 |v12|•v22v11 v21

notet is simply the sum of the cosine distance connections or to close file handles. A more sophisticated way is to study a rich dataset of known attacks (e.g., Exploits Database) and design rule-based approaches to mitigate attacks based on different patterns. We refer these active actions, together with introspection, as ATOM's orchestration module. Orchestration in ATOM can be greatly customized to suite the needs for different tasks, such as identification of different attacks, and dynamic resource allocation in an IaaS system.

7. VM CLUSTERING

ATOM enables a continuous understanding of the VMs in an IaaS system. In addition to anomaly detection, this framework is also useful for many other decision making and analytics applications. Hence, in addition to using a PCA-based approach in the monitoring component, we will demonstrate that it is possible to design and implement a VM clustering module to be used in the monitoring component.

The objective of VM clustering is to cluster a set of VMs into different clusters so that VMs with similar workload characteris- tics end up in the same group. This operation assists making load balancing decisions, as well as developing customized, fine-tunedof each corresponding pair of eigen vectors from VM1 and VM2, weighted by the difference of the corresponding eigen values to ensure that the variations do not differ a lot.

Step 4. Using VMdist as the distance measure between any two VMs, we use DBSCAN [18] to cluster similar VMs together. DBSCAN is a threshold-based (aka density based) clustering algorithm which requires two parameters: ε which is the density threshold, and minPts which is the number of minimum points to form a cluster. DBSCAN expands a cluster from an un- visited data

point towards all its neighboring points provided the distance is within ε, and then recursively expands from each of the neighboring point. Points are marked as an outlier if the number of points in their cluster is fewer than minPts. Compared with other popular clustering methods like k-means, density-based clustering algorithm does not require the prior-knowledge on the number of clusters, neither does it need to iteratively compute an explicit "centroid" and re-cluster at every iteration.

By default, ATOM sets minPts=10, and computes the thresh- old value ε using a sampling based approach. More specifi- cally, we randomly select n pairs of VMs and compute their VMdist. We sort the n VMdist values, and set ε = VMdisti if

VMdisti+1 > 5 VMdisti. The intuition is that for any point the distance to a point in a different cluster is much longer than the distance to a point in the same cluster, and we want to find a large enough "inner cluster" distance and use it as the threshold value ε to determine whether two points belong to the same cluster.

8. EVALUATION

We implemented ATOM using Eucalyptus as the underlying IaaS system. The virtual machine hypervisor running on each NC is the default KVM hypervisor. Each VM has an m1.medium type on Eucalyptus. ATOM tracks 7 metrics from each VM instance: CPU Utilization, NetworkIn, NetworkOut, DiskReadOps, DiskWriteOps, DiskReadBytes, DiskWriteBytes. All experiments are executed on a linux machine with an 8-core Intel(R) Core(TM) i7-3770 CPU @ 3.40GHz computer.

8.1 Online tracking

In the evaluation the data collection time interval is set to 10 seconds, i.e., raw values for different metrics are collected every 10 seconds on a NC (observer), which produces 360 values for each metric per hour.Instead of sending every value to CLC (the tracker), the modified CloudWatch with ATOM's online tracking component selectively sends certain values based on Algorithm 1, from NC to CLC. Figure 7 shows the number of values sent for each metric over 2 hours, with different workloads (e.g., TPC- C benchmark over MySQL) and different Δ values. Among the 7 metrics for each VM, only the first 5 ones are shown in each sub-figure, as DiskReadBytes/DiskWriteBytes follow the same patterns with DiskReadOps/DiskWriteOps in all experiments.

0 and 0.2% when VM is idle, so Δ value for this metric is only (roughly) 0.01%. This figure shows that even when allowing a very small error, tracking component already leads to significant savings. Figure 7(c) shows the results when VM is running TPC-C benchmark on a MySQL database, which involves large disk reads and writes. Δ is set as the average of the exact values in 2 hours when VM is idle. This is reasonable even for users who do not allow any error, because

Δ is merely the average of the amount consumed by an idle VM.

Note that in this figure, NetworkIn and NetworkOut only have 2 values sent to CLC in 2 hours with the tracking component. This figure tells us that even if VM is intensively used and almost no error is allowed, the tracking component is still highly effective. Figure 7(d) demonstrates the result when VM is running the same workload, while Δ value for each metric is now set as 10% of the average value when the VM has been running the same workload for 2 hours, i.e., larger errors are allowed. Clearly, the tracking component becomes even more effective.

Figure 8 explains how the online tracking component works. It shows both values sent by standard CloudWatch (without tracking) and values sent by modified CloudWatch with ATOM tracking, with a time interval of 1000 seconds for the NetworkOut metric from Figure 7(b). This clearly illustrates that at each time instance, with online tracking, the current (exact) value is not sent if it is within Δ threshold of the last sent value; and at each time point,the last value sent to CLC is always within Δ of the newest value observed on NC. The values sent by the tracking method closely approximate those exact values, with much smaller overhead.

8.2 Automated online monitoring and orchestration

We design three experiments to illustrate the effectiveness of ATOM's monitoring module. For each experiment, we use a false alarm rate α = 0.2% and its deviation μ = 1% (to set the tracking error bound). Meanwhile the Qα threshold with α = 0.5% is also calculated to compare against. The online tracking error Δ is calculated dynamically according to the equations in Section

5.1 at the CLC, and set using the algorithm in Section 5.2 on each NC. Three VMs with a type of m1.medium co-located in one Eucalyptus physical node are monitored for each experiment, which form a t 21 data matrix. Dimensions 1-7 belong to VM 1, 8-14 are for VM 2, whereas VM 3 owns the rest.

We use two types of normal workloads and two kinds of at- tacks in all three experiments. The two types of normal workloads include network and disk workloads. For the network workload, an Apache web server is installed and constantly responding WebBench network requests. The disk workload is TPC-C bench- mark against MySQL database [19]. The two types of attacks are DDoS attack and resource-freeing attack [13]. In our experiment, DDoS attack treats the affected VM as a compromised zombie and sends malicious traffic to the target IP address. Resource-freeing attack is launched by VM 3 targeting the web server on VM 2 to gain more cache usage. Note that there is a 4-th VM running WebBench and a 5-th VM running Apache web server as the target of DDoS bots. The first two hours are used to build PCA model for each experiment, while the anomaly happens at the third hour. The settings for each experiment is shown in Table 2.

Experiment	Workload	Attack
1	VM 1, 3 idle; VM 2 network workload	DDoS attacinside VM 2 k
2	VM 1 idle; VM 2, 3 network workload	DDoS attacinside VM 2,k 3
3	VM 1 idle; VM 2 network workload; VM 3 disk workload	Resource-freeing attack from VM 3 to VM 2

TABLE 2 Online monitoring experiment setup.

In the first experiment, VM 2 runs an Apache web server while the other 2 VMs are idle. A DDoS attack turns VM 2 to be a zombie at the third hour, using it to generate traffic towards the target IP (the 5th VM in our experiment). Note that this attack is hard to detect using the simple threshold approach in existing IaaS systems. The normal workload on VM 2 is a network workload, which already has a large amount of NetworkIn/NetworkOut usage, sending out malicious traffic only changes roughly 10% − 30% to the mean of normal statistics.

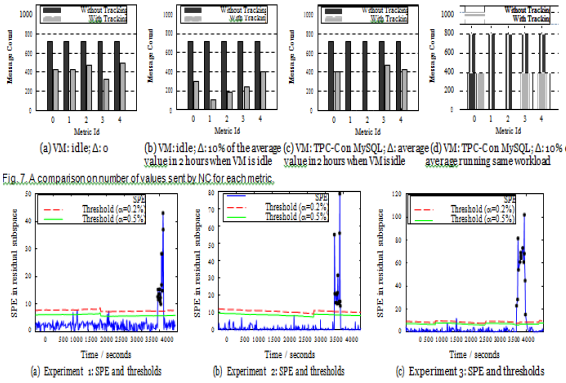
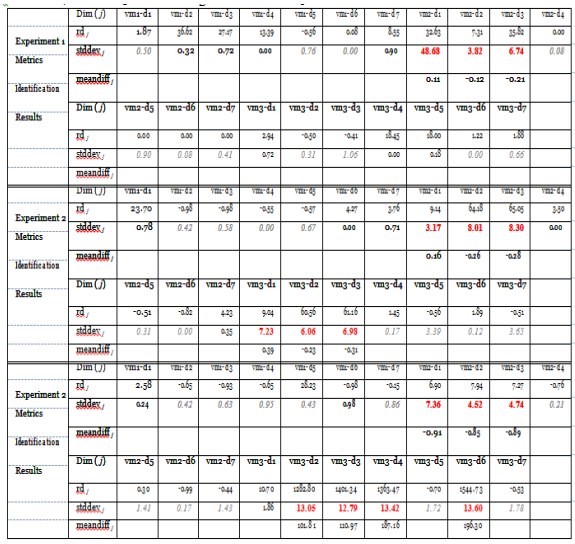


Fig. 9. Time series plots of SPE against thresholds Qα with α = 0.2% and 0.5%.



Hence it is difficult to set an effective threshold value even for an experienced user due to the fact that the underlying normal traffic might oscillate within a range. Yet ATOM's monitoring module successfully finds the underlying pattern, and detects time instances that are abnormal (when attacks are ongoing). Figure 9(a) shows the online monitoring and detection process. The dashed line corresponds to threshold Qα for α = 0.2%, and the solid line shows Qα for α = 0.5%. SPE of the approximate data matrix projected onto the residual subspace is plotted, where the black dots indicates the time instances when DDoS attack happens. Clearly, ATOM has successfully identified all abnormal time instances correctly.

Once a time instance is considered abnormal, ATOM imme- diately runs metrics identification procedure to find the affected VMs and metrics. As described in Section 4.3.4, ATOM firstly finds out potential abnormal dimension(s) by analyzing the aver- age

change portion rd j between abnormal data points and normaldata points projected onto residual subspace. Then for dimensions that have significant changes, ATOM computes stddev j as sug- gested in Section 4.3.4, and also calculates the average change meandiff j if stddev j is above a threshold. Recall m is the number of consecutive abnormal time instances until tnow. The results when m = 5 are shown in the first table of Table 3. Note that only for the dimensions having large enough residual portion (rd j) does ATOM computes the standard deviation error (stddev j). Among the 3 VM instances being tracked and monitored, ATOM correctly identifies an anomaly happening on VM 2, and more specifically, it discovers that the anomaly is from its first three dimensions (CPUUtilization, NetworkIn, NetworkOut), indicated by the bold values. Note that NetworkIn and NetworkOut actually go down because of DDoS attack. Our guess is that WebBench tends to saturate the bandwidth available for the VM, while the DDoS attack we use launches many network connections but not sending as much traffic. The CPUUtilization, however, goes up due to theattack. Nevertheless, ATOM is able to identify all three abnormal metric dimensions.

In the second experiment, both VM 2 and VM 3 are running the network workload, and the same DDoS attack turns both VMs to be zombie VMs simultaneously. Not only ATOM is able to detect an anomaly happened as shown in Figure 9(b), but also it finds similar patterns on the correct metrics from both VM 2 and VM 3 as illustrated in the second table of Table 3, which shows the metrics identification results when m = 5. By sending this information to the orchestration component, the introspection overhead could be saved by first introspecting one VM, and then checking if another one has the same malicious behavior going on. The third experiment illustrates ATOM's ability to detect a different type of attack, the resource-freeing attack [13], a subtle attack where the goal is to improve a VM's performance by forcing a competing VM to saturate some bottleneck and shift its usage VM 3 launches GoldenEye attack, which achieves a denial- of- service attack on the HTTP server running on VM 2 by consuming all available sockets, and is paired with cache control. We show that ATOM successfully finds the two VMs, and by its metrics identification procedure, it suggests the possibility of an resource- freeing attack and provides useful data to its orchestration module in assisting the VMI procedure on VMs 2 and 3.

Figure 9(c) plots the monitoring and the detection process. The black dots indicate the time instances when abnormal behavior happens. This figure, as before, only shows that an anomaly has happened. While the second table in Table 3 analyzes where the anomaly has originated. The stddev j values show what the abnormal dimensions are, on VM 2: CPUUtilization (vm2-1), NetworkIn (vm2-2), NetworkOut (vm2-3); on VM 3: NetworkIn (vm3-2), NetworkOut (vm3-3), DiskReadOps (vm3-4), DiskRead- Bytes (vm3-6).

Further analysis on meandiff j finds out NetworkIn and Net- workOut statistics on VM 2 decrease nearly by an order, while VM 3 sees significant increase in NetworkIn, NetworkOut and

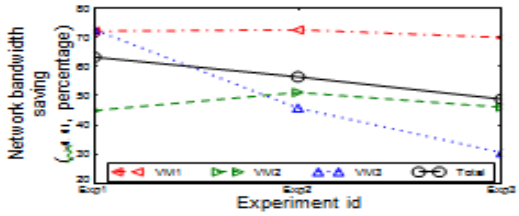


Fig. 10. Network bandwidth saving in each experiment in Figure 9. especially its disk read statistics (DiskReadOps and DiskRead- Bytes). This is a typical resource freeing attack as described in [13], where network resource has become the bottleneck of a target VM, and the beneficiary VM gains much of the shared cache usage by showing a significant increase in disk read statistics. The sudden increase in NetworkIn/NetworkOut in VM 3 also suggests that VM 3 might be the attacker of VM 2 by sending malicious traffic to it.

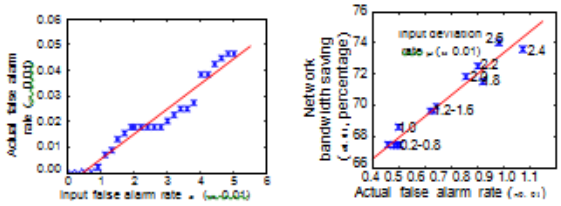
Further analysis by VMI in ATOM's orchestration module shows that most of VM 2's sockets are occupied by connecting to VM 3, thus the anomaly could be mitigated by closing such connections and limiting future ones. Of course, VM 3 could use a helper to establish such malicious connections with VM 2 as suggested in paper [13], yet ATOM is still able to raise an alarm to end user and suggest a possible ongoing resource-freeing attack. Users should be aware that the larger an alarm level m is, the more accurate the metrics identification process is, and the more likely a bigger damage an attack has caused.

Lastly, Figure 10 shows the communication overhead saving achieved by ATOM in these three experiments. For the first hour in each experiment, each VM should have sent 7 360 = 2520 values to CLC. However, by using online tracking and set- ting the tracking error Δ dynamically, the number of values sent are significantly reduced. Since each message is of the same size, we calculate the network bandwidth saving as (1 - number of messages sent with tracking module number of messages sent without tracking module

8.3Sensitivity analysis

There are only two parameters to set up for ATOM: the desired false alarm rate α which is used to calculate the anomaly detection threshold Qα in Section 4.3.3, and the maximally allowed false alarm deviation rate μ as defined in Section 5 which is used to bound and adjust the tracking threshold Δ.

We design two experiments to analyze the sensitivity of α and μ respectively. We use the same dataset for these experiments to clearly demonstrate the impacts of having different values for α and μ. The first experiment varies the values of α and counts the actual number of false alarms. The second experiment gradually increases the values of μ, and measures the actual number of false alarms and the network savings achieved by the tracking module (when its tracking threshold Δ is dynamically adjusted by ATOM). The result of the first experiment is shown in Figure 11(a), which shows how the actual false alarm rate changes with the



(a) Actual false alarm rate increases (b) With the increase of μ (annotated with, and is much smaller than, the in the figure), both actual false alarm input theoretical false alarm rate α. rate and communication saving go up.

Fig. 11. Sensitivity analysis.

increase of values of α. When α increases from 0.1% (confidence level 99.9%) to 5%, the actual false alarm rate also increases. But the actual false alarm rate is always much smaller, ranging from just 0.006% to 0.05%. The common practice is to choose α < 0.5%, and far fewer actual false alarms (less than 0.5 per 100 data points) will be produced.

Figure 11(b) shows the results of the second experiment. We vary the false alarm deviation rate μ from 0 to 2.6%, with a step size of 0.2%. For each value of μ, we run the experiments for 10 times, where α value ranges from 0.1% to 0.5%, with a step size of 0.1%. Finally the average results are computed with respect to each μ. We are interested in network bandwidth saving in percentage (y axis) and the actual false alarm rate (x axis). Larger μ means a bigger Δ threshold is used by ATOM, and thus leads to more savings in network bandwidth. However the growth of μ is also accompanied with an increase in the number of actual false alarms, which suggests a trade-off between using more network bandwidth and having fewer false alarms.

But generally speaking, a small value for μ is sufficient to provide enough communication savings. Note that all attacks were still detected in all experiments, achieving false negative rate of 0%. As shown in Figure 9, using both α = 0.2% and α = 0.5%, ATOM could easily identify the attacks. A higher α value leads to a lower threshold value for attack detection, meaning attacks are more easily to be detected though it may lead to more false alarms. ATOM allows users to control the false alarm rate and the tracking threshold by adjusting α and μ.

In our analysis, a wide range of thresholds suffice to detect denial of service attacks or resource starving attacks while achiev- ing large communication saving. However in production systems, it is important to provide users feedback about the effects of their error setting. ApproxHadoop [21] and Social Trove [22] use statistical models from extreme value theory to estimate the effects of delta. In our models, statistical models can help over short periods, but over long periods we would expect malicious attackers to adapt their attacks to reduce their chances of being caught. In this situation, it is important to use offline benchmarking to assess the effect of the error threshold as shown in [23], in which the authors provide techniques to overlap online executions with different delta settings, allowing us to understand the effects of delta empirically without degrading throughput.

8.4 ATOM scalability evaluation

To evaluate the scalability of ATOM, we evaluated the key performance metrics of ATOM with an increasing number of VMs (from 2 to 6). In each configuration, we perform online monitoring using the adapted PCA-based anomaly detection using a sliding window of size 100 (time instances), combined with either online tracking or no tracking (i.e., send everything). We report the average for the false alarm rate, the average PCA running time, and the total number of messages sent from NC to the CLC, per sliding window. The results are shown in Figure 12.

Larger number of VMs leads to higher communication cost in ATOM. However, the tracking component of ATOM becomes more effective with more VMs, as shown in Figure 12(a). This is because there are more opportunities for communication savings when there is a higher probability of temporal locality on one of the many VMs' performance metrics.

The computation cost in ATOM is linear to the number of VMs, as shown in Figure 12(b), which is as expected. Neverthe- less, the overall computation overhead of ATOM is still fairly small (in just a few milliseconds per sliding window).

The measured false alarm rate actually decreases initially with more VMs. But when the number of VMs keeps increasing, the measured false alarm rate will eventually start to increase, as indicated in Figure 12(c). Initially, when presented with more data, the PCA-based approach becomes more effective in "learning" the normal subspace, hence results in a reduced false alarm rate. But as number of VMs continues to increase, the dimensionality of the data matrix becomes larger, eventually making it less effective to detect abnormal subspace after dimensionality reduction. Never- theless, ATOM remains very effective in all cases; the false alarm rates are smaller than 1% in nearly all test cases.

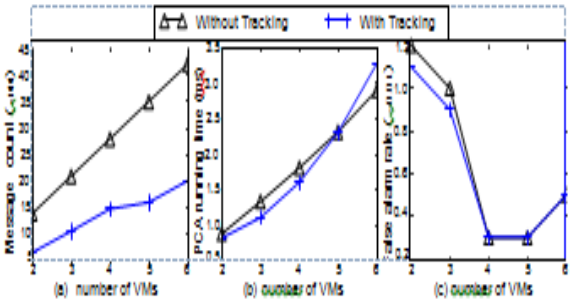


Fig. 12. Impacts to ATOM's performance with respect to the increasing number of VMs.

8.5 ATOM vs. classic offline PCA anomaly detection

In this section we show what happens if we simply apply the classic offline PCA method (offlinePCA) that has been widely used for anomaly detection in previous literature [12].

Specifically, to use offlinePCA, each time we delete the oldest time instance and add the newest one in the sliding window, and then use the data matrix inside this sliding window to do PCA. Each time only the newest time instance data need to be verified. After transforming the original data to the rotated PCA space, we measure each dimension at the newest time instance, use the first dimension that exceeds 3 times of standard deviation along that axis as the starting dimension of the residual subspace, and if no such dimension exists, SPE need not be calculated and checked against the anomaly threshold.

To compare the performance of this method with our approach, we run another experiment with the same setting as that in experiment 1 shown in Table 2 and apply the above method. Figure 13 shows the result of anomaly detection using this method. Note that SPE is not calculated for all time points as explained above. The blue dots indicate SPE of certain time instances and red crosses show the threshold Q_α to compare against at the same time instances. Anomaly happens at the end of first hour (3600 on the time dimension in Figure 13), and it continues ever since. So we would expect at the second hour there should be blue dots and red crosses at every time point, and all blue dots should be above the red crosses. However as shown in Figure 13, it only takes less than 4 minutes (only the first 20 time instances after 3600 have been detected being abnormal) for the attack to escape from the monitoring and detection, and make itself be identified as normal behavior. Note that towards the end the SPE values in Figure 13 here is different from that in Figure 9 where SPE values are normal again in the end which is because the attacks were mitigated in the experiment behind Figure 9 by ATOM's orchestration module. The key difference causing this is that ATOM's online method based on PCA ensures only normal data points are used for anomaly detection, while the abnormal data points may skew the PCA model built by the naive offline method.

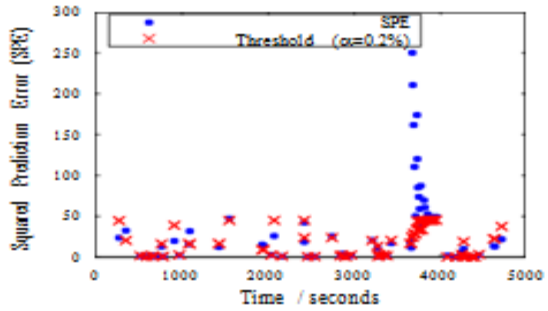


Fig. 13. offlinePCA using the settings in Experiment 1 from Table 2 (SPE is computed as in Section 4.3.3).

8.6 VM clustering evaluation

To evaluate the accuracy and robustness of our VM clustering method, we run an experiment using 102 VM data vectors (each VM data vector is a 7-dimension vector with 7 performance metrics, collected from a VM running a particular workload at the time of data collection). Among the 102 VM data vectors, 34 were idle, another 34 were running a TPCC benchmark on MySQL database and the rest 34 were running an Apache web server. We run the VM clustering for 10 times and calculate the average results. The experiment result shows that our method is able to precisely identify the 3 clusters, with a few points marked as outliers each time. The average clustering precision is 96.08%, the average clustering recall is 95.10%, and the average clustering F-measure is 95.59%.

Since we used a density-based clustering algorithm which groups nearby VM data vectors together, to test its robustness, for each VM data vector we measured its distance to the closet neighbor inside the same cluster (denoted as "inner cluster dis- tance") and the closest distance to a VM data vector in a different cluster (denoted as "inter cluster distance"). A histogram of the two measures for all 102 VM data vectors are shown in Figure 14. We can see that there is a large gap between the two distances, which shows the robustness of our clustering method and it is fairly robust and insensitive to a wide range of threshold ε values.

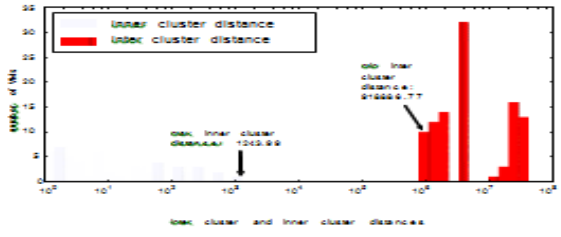


Fig. 14. Histogram of clustering distances.

8.7 Discussion

The choice of Δ. Larger Δ values lead to more savings, but with less accurate data matrix. However cloud users don't have to worry about setting Δ values; ATOM only needs the user to specify a tolerable deviation rate μ on the detection threshold. ATOM is then able to adjust Δ values dynamically online.

Possible false alarms. Resource usage pattern may simply change due to normal changes in user activities, in which case ATOM may raise false alarms. Nevertheless, ATOM is able to raise alarms and let users decide the right course of actions to take by assisting users with its orchestration module. ATOM also uses the new workload chracteristics to adjust its monitoring component to adapt to a new workload dynamically and automatically in an online fashion.

Overhead. The tracking module, by simply apply algorithm 1 before sending out each value, introduces only O(1) overhead. The monitoring module could leverage a recursive update proce- dure, so that it is possible to use the current PCA model to do incremental update instead of computing from scratch, e.g., [14], [24]. Depending on the PCA algorithm used, it is polynomial to the sliding window size and number of dimensions. In contrast, the overhead saved by ATOM is significant. Not only a major fraction of network traffic could be saved from CC to CLC, but also the effort to apply VMI. The orchestration module orchestrates and introspects only the affected VMs and metrics, and only when needed, hence, leads to much smaller overhead than full-scale VM introspection that are typically required.

Other attacks. Our experiments use the same set of metrics that are monitored by CloudWatch and demonstrate two different types of attacks. But ATOM can easily add any additional metric with- out much overhead. This means that it can be easily extended when necessary with additional metrics for monitoring and detecting different kinds of attacks.

9. RELATED WORK

To the best of our knowledge, none of existing IaaS platforms is able to provide continuous tracking,

monitoring, and orchestration of system resource usage. Furthermore, none of them is able to do intelligent, automated monitoring for a large number of VMs and carry out orchestration inside a VM.

Cloud data monitoring. Most existing IaaS systems follow the general, hierarchical architecture as shown in figure 2. Inside these systems, there are imperative needs for the controller to continuously collect resource usage data and monitor system health. AWS [1] and Eucalyptus [4], [5] use CloudWatch [25] service to monitor VMs and other components in some fixed intervals, e.g., every minute. This provides cloud users a system- wide visibility into resource utilization, and allows users to set some simple threshold based alarms to monitor and ensure system health. OpenStack [26] is developing a project called Ceilome- ter [27], to collect resources utilization measurements. However, these approaches only provide a discrete, sampled view of the system. Several emerging startup companies such as DATADOG [28] and librato [29] could monitor in a more fine-grained granu- larity, provided the required softwares are installed. However, this inevitably introduces more network overhead to the cloud, which becomes worse when the monitored infrastructure scales up. On the contrary, ATOM significantly reduces the network overhead by utilizing the optimal online tracking algorithm, while providing just about the same amount of information. Furthermore, all these cloud monitoring services offer very limited capability in monitoring and ensuring system health. Astrolabe [30] is a mon- itoring service for distributed resources, to perform user-defined aggregation (e.g. number of nodes that satisfy certain property) on-the-fly for the host hierarchy. It is intended as a "summarizing mechanism". Similar to Astrolabe, SDIMS [31] is another system that aggregates information about large-scale networked systems with better scalability, flexibility, and administrative isolation. Ganglia [32] is a general-purpose scalable distributed monitoring system for high performance computing systems which also has a hierarchical design to monitor and aggregate all the nodes and has been used in many clusters. These efforts are similar to the CloudWatch module currently used in AWS/Eucalyptus, and they reduce monitoring overhead by simple aggregations. While the purpose of ATOM's tracking module is to reduce data transfer, but it does so using online tracking instead of simply aggregating which delivers much more fine-grained information.

STAR [33] is a hierarchical algorithm for scalable aggregation that reduces communication overhead by carefully distributing the allowed error budgets. It suites systems like SDIMS [31] well. InfoEye [34] is a model-based information management system for large-scale service overlay networks through a set of monitoring sensors deployed on different overlay nodes with reduced overhead achieved by ad-hoc conditions filters. InfoTrack [35] is a monitoring system that is similar to ATOM's tracking module, in that it tries to minimize continuous monitoring cost with most information precision preserved, by leveraging temporal and spatial correlation of monitored attributes, while ATOM uti- lizes an optimal online tracking algorithm that is proved to achieve the best saving in network cost without any prior knowledge on the data. MELA [36] is a monitoring framework for cloud service which collects different dimensions of data tailored for analyzing cloud elasticity purpose (e.g. scale up and scale down). ATOM may use MELA to collect, track, and monitor different types of metrics than those already available through Cloud Watch. Cloud security. IaaS system also brings us a new set of secu- rity problems. Leading cloud providers have developed advanced mechanism to ensure the security of their IaaS systems. AWS [37] has many built-in security features such as firewalls, encrypted storage and security logs. OpenStack use a security component called Keystone [38] to do authentication and authorization. It also has security rules for network communication in its network component Neutron [39]. Other IaaS platforms have similar se- curity solutions, which are mainly firewalls and security groups. Nevertheless, it is still possible that hackers could bypass known security policies, or cloud users may accidentally run some mali- cious software. It is thus critical to be able to detect such anomaly in near real-time to avoid leaving hackers plenty of time to cause significant damage. Hence we need a monitoring solution that could actively detect anomaly, and identify potentially malicious behavior over a large number of VM instances. AWS recently adopts its Cloud Watch service for DDoS attacks [3], but it re- quires user to check historical data and set a "magic value" as the threshold manually, which is

unrealistic if user's underlying workloads change frequently.

In contrast, ATOM could automatically learn the normal behavior from previous monitored data, and detect more complex attacks besides DDoS attacks using PCA. PCA has been widely used to detect anomaly in network traffic volume in backbone networks [11], [12], [40], [41], [42], [43]. As we have argued in Section 4.1, adapting a PCA-based approach to our setting has not been studied before and presented significant new challenges.

The security challenges in IaaS system were analyzed in [6], [44], [45], [46]. Virtual machine attacks is considered a major security threat. UBL [7] uses VM usage data to train Self-Organizing Maps for anomaly prediction, which serves a similar purpose to ATOM's monitoring component, and has been analyzed in details in Section 1. PerfCompass [47] could identify whether a VM performance anomaly is caused by internal fault like software bugs, or from an external source such as co-existing VMs, through collecting system call traces and checking the execution units being affected. In contrast, ATOM's introspection component leverages existing open source VMI tools such as Stackdb [9] and Volatility [16] to pinpoint the anomaly to the exact process.

VMI is a well-known method for ensuring VM security [48], [49], [50], [51]. It has also been studied for IaaS systems [52], [53], [54]. However, to constantly secure VM using VMI tech- nique, the entire VM memory needs to be traversed and analyzed periodically. It may also require the VM to be suspended in order to gain access to VM memory. Blacksheep [17] is such a system that detects rootkit by dumping and comparing groups of similar machines. Though the performance overhead is claimed to be acceptably low to support real-time monitoring, clearly user programs will be negatively affected. Another solution was suggested [55] for cloud users to verify the integrity of their VMs. However, this is not an "active detection and reaction" system. In contrast, ATOM enables triggering VMI only when a potential attack is identified, and it also helps locate the relevant memory region to analyze and introspect much more effectively and efficiently using its orchestration component.

10. CONCLUSION

We present the ATOM framework that can be easily integrated into a standard IaaS system to provide automated, continuous tracking, monitoring, and orchestration of system resource usage in nearly real-time. ATOM is extremely useful for anomaly detection, auto scaling, and dynamic resource allocation and load balancing in IaaS systems. Interesting future work include extending ATOM for more sophisticated resource orchestration and incorporating the defense against even more complex attacks in ATOM.

11. ACKNOWLEDGMENTS

Min Du and Feifei Li were supported in part by grants NSF CNS-1314945 and NSF IIS-1251019. We wish to thank Eric Eide, Jacobus (Kobus) Van der Merwe, Robert Ricci, and other members of the TCloud project and the Flux group for helpful discussion and valuable feedback. The preliminary version of this paper appeared in IEEE BigData 2015 [56].

REFERENCES

[1] Amazon. http://www.aws.amazon.com/. Accessed Nov. 5, 2016.

[2] ITWORLD. http://www.itworld.com/security/428920/attackers-install- ddos-bots-amazon-cloud-exploiting-elasticsearch-weakness. Accessed Nov. 5, 2016.

[3] Amazon. AWS Best Practices for DDoS Resiliency. https://d0.awsstatic.com/whitepapers/DDoS White Paper June2015.pdf. Accessed Nov. 5, 2016.

[4] Eucalyptus. http://www8.hp.com/us/en/cloud/helion-eucalyptus.html. Accessed Nov. 5, 2016.

[5] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Yous- eff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in CCGRID, 2009.

[6] W. Dawoud, I. Takouna, and C. Meinel, "Infrastructure as a service security: Challenges and solutions," in INFOS, 2010.

[7] D. J. Dean, H. Nguyen, and X. Gu, "Ubl: Unsupervised behavior learning for predicting performance anomalies in virtualized cloud systems," in ICAC, 2012.