ATOM: Automated Tracking, Orchestration and Monitoring of Resource Usage in Infrastructure as a Service Systems

Min Du, Feifei Li
School of Computing, University of Utah
mind@cs.utah.edu, lifeifei@cs.utah.edu

***Abstract*—We present ATOM, an efficient and effective framework to enable automated tracking, monitoring, and orchestration of resource usage in an Infrastructure as a Service (IaaS) system. We design a novel tracking method to *continuously track* important performance metrics with low overhead, and develop a principal component analysis (PCA) based approach with quality guarantees to continuously monitor and automatically find anomalies based on the *approximate tracking results*. Lastly, when potential anomalies are identified, we use introspection tools to perform memory forensics on virtual machines (VMs) to identify malicious behavior inside a VM. We deploy ATOM in an IaaS system to monitor VM resource usage, and to detect anomalies. Various attacks are used as examples to demonstrate how ATOM is both effective and efficient to track and monitor resource usage, detect anomalies, and orchestrate system resource usage.**

I. Introduction

The Infrastructure as a Service (IaaS) framework is a popular model in realizing cloud computing services. While IaaS model is attractive, 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 resource monitoring and security.

For example, Amazon Web Service (AWS) provides auto scaling and load balancing services to help cloud users make the best use of their (paid) resources. A critical module in achieving this is the ability to monitor resource usage from many virtual machines (VMs).

Security is another paramount issue while realizing cloud computing through an IaaS system. For instance, it was reported in late July 2014, adversaries attacked Amazon cloud by installing distributed denial-of-service (DDoS) bots on user VMs through exploiting a vulnerability in Elasticsearch [11]. We discover that resource usage data could provide critical insights to address security concerns. Thus, a cloud provider could utilize the constantly monitored resource statistics to do anomaly detection.

These observations illustrate that a fundamental challenge underpinning several important problems in an IaaS system is the *continuous tracking and monitoring* of resource usage, which motivates us to design and implement ATOM.

Eucalyptus is an open source cloud software that provides AWS-compatible environment and interface [17]. Eucalyptus provides an AWS-like service called *CloudWatch*. CloudWatch is able to monitor resource usage of each VM, collected by each Node Controller (NC), and then reported to the Cloud Controller (CLC). Clearly, gathering resource usage in real time introduces overhead in the system (e.g., communication overhead). When there are plenty of VMs to monitor, the problem becomes even worse. 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 achieve continuous understanding and protection of the system.

Another limitation in existing approaches like CloudWatch is that they only do passive monitoring. We observe that, e.g., in the aforementioned DDoS attack to Amazon cloud, alarming signals *can be learned* from resource usage data, which also provides the opportunities to trigger VM introspection (VMI) to debug the system. VMI is used to detect and identify malicious behaviors inside a VM, but go through the entire memory constantly is expensive without the knowledge of when and where. Our goal is to trigger and guide VMI only when needed in an automatic manner.

Our contribution. Motivated by these discussions, we present the ATOM framework. ATOM works with an IaaS system and provides *a*utomated *t*racking, *o*rchestration, and *m*onitoring 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 running 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 continuously analyzes the resource usage data reported by the online tracking module to detect anomaly. A naive method would be 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 developed based on PCA to do mining in the resource data, and generates anomaly information to aid further analysis by the orchestration component when this happens.

Lastly, ATOM leverages existing VMI tools into its

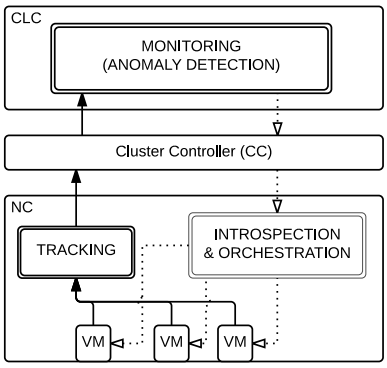


Figure 1. The ATOM framework.

orchestration component, which only introspects specific regions in the VM memory space based on the information provided by the online monitoring component, when it is triggered to do so.

Paper organization. The rest of this paper is organized as follows. Section II gives an overview on the design of ATOM, and the threat model it considers. Sections III and IV describe the online tracking and the online monitoring modules in ATOM. Section V introduces the orchestration module. Section VI evaluates ATOM using Eucalyptus cloud and shows its effectiveness. Lastly, section VII reviews the related work, and section VIII concludes the paper.

II. The ATOM Framework

Figure 1 shows the ATOM framework. For simplicity, only one CC and one NC are shown. ATOM adds three components to an IaaS system like AWS and Eucalyptus:

(1) Tracking component: ATOM adapts the optimal online tracking algorithm inside the data collection service on NCs. This enables continuous measurements at the CLC;

(2) Monitoring component (anomaly detection): ATOM adds this component in the CLC. It uses a modified PCA method which continuously analyzes a subspace defined by the measurements from the tracking module, and automatically raises an alarm whenever a shift in the 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 pinpoint the abnormal dimensions to assist the orchestration.

(3) Orchestration component (introspection and debugging): When a potential anomaly is identified by the monitoring component, an `INTROSPECT` request along with anomaly information is sent to the orchestration component on NC, in which VMI tools are triggered to introspect the specific regions inside a VM.

Thread Model. ATOM provides real time tracking and monitoring on the usage of cloud resources in an IaaS system. It further goes out to detect and prevent attacks that could cause notable or subtle 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 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.

III. Tracking Component

Consider Eucalyptus CloudWatch as an example. It is capable of collecting and aggregating data from resources such as VMs and storage for as frequently as one minute and store them for up to two weeks. It also allows cloud users to set some alarm (essentially, a threshold) for a specific measure, and be notified or let it trigger some predefined action if the alarm conditions are met.

Various measures are monitored overtime on each VM, each of which is called a *metric*. The measurement for each metric, for example, *Percent* for CPUUtilization, *Count* for DiskReadOps and DiskWriteOps, *Bytes* for DiskReadBytes, DiskWriteBytes, NetworkIn and NetworkOut, is called *Unit* and is numerical.

A continuous understanding of these values is much more useful than a periodic, discrete sampled view that is 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 CPUUtilization is *guaranteed* to be within $\pm 3\%$ 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. The CLC could use the last received value as an acceptable approximation 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.

To achieve this, a naive method would be to send the exact current value whenever it's more than Δ away from the last sent one. 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 could send only one message - the value $\frac{\Delta}{2}$. Formally, this is known as the *online tracking* problem, which is formalized and studied in [20]. In online tracking, an observer observes a function $f(t)$ in an online fashion, which means she sees $f(t)$ for any time t up to the current. 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 $f : \mathbb{Z}^+ \rightarrow \mathbb{Z}$ is the function that 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 t_{now}, if the observer does not send a new value $g(t_{now})$ to the tracker, it must satisfy $\left\|f(t_{now}) - g(t_{last})\right\| \leq \Delta$, where $g(t_{last})$ 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 $O(\log \Delta)$; that means *in the worst case*, its communication cost is only $O(\log \Delta)$ times worse than the cost of the offline optimal algorithm that knows the function $f(t)$ for entire time domain [20]. 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 [20], 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.

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 $\left\|f(t_{now}) - g(t_{last})\right\| > \Delta$;
  $S_{upper\_bound} = \min(S_{upper\_bound}, f(t_{now}) + \Delta)$;
  $S_{lower\_bound} = \max(S_{lower\_bound}, f(t_{now}) - \Delta)$;
end while /* *this algorithm is run by observer* */

The central idea of our algorithm is to always send the median value from the range of possible valid values, denoted by S, whenever $f(t_{now})$ has changed more than Δ (could be non-integer) from $g(t_{last})$. 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 $2\Delta/\gamma$, and since S is a finite set, it always becomes an empty set at some step following the above 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 $O(\log(\Delta/\gamma))$. Due to the space constraint, the proofs are omitted here but will be available in the extended version of this paper.

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 the CLC by following Algorithm 1. The CLC, as the tracker, stores and analyzes these values, whenever they are reported from an NC.

IV. Monitoring Component

By continuously tracking values of various 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 utilize the threshold approach used by CloudWatch to do auto scaling tasks [2], which allows user to set an alarm along with an alarm action that can be triggered if certain threshold is met. However, in practice, it is very hard to set a magic effective threshold in a dynamic environment like an IaaS system. Besides, it's inconvenient to change the threshold for each metric every time a user changes workloads. Thus an automated monitoring method would be very useful.

A. An overview of PCA method

Given a data matrix in $\mathbb{R}^d$ space, some dimensions in which are possibly correlated, the PCA method could transform this matrix into a new coordinate system, resulting 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 previous ones. The intuition to use PCA for anomaly detection, 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 into a new space, the original abnormal point would have a large projection length on the axis supposed to have very small variance. Hence anomalies can be detected by analyzing the projection length onto these axes. Using PCA for anomaly detection has been widely studied for network traffic analysis [14], [16].

That said, there are three challenges we need to address: 1) do online PCA using a sliding window; 2) pinpoint the abnormal dimensions once an anomaly is identified; 3) handle approximate data from the tracking module and take the tracking errors into analysis.

B. The data matrix

Suppose there are d' metrics reported for each VM. Then, PCA could be performed on these data which form a $t \times d'$ matrix, where t is the length of a time-based sliding window. A more general and interesting case is to perform online monitoring over a data matrix composed of *multiple VMs' data*, e.g., $d = d' \cdot n$ dimensions where n is the number of VMs. For VMs hosted on the same physical

node, or even the same cloud, it's quite possible that one VM may attack another [19], or several VMs are attacked by the same process simultaneously. Detecting anomaly on a d-dimensional space makes it easier to discover such correlations. Moreover, performing PCA on multiple VMs' statistics yields a higher residual space, leading to more accurate anomaly detection.

Recall that ATOM's tracking module ensures that at any time point τ, for each metric E, the CLC can obtain a value v'_τ that is within $v_\tau \pm \Delta$, where v_τ is the exact value of this metric at time τ from a VM of interest. Next we show how to design an online PCA method to detect anomaly using a $t \times d$ matrix $\mathbf{M}$, where t is a fixed number of recent time instances, and d is the number of metrics for each VM times total number of VMs being monitored. 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.

C. Monitoring in detail

The following matrices are used in our construction besides the original reported data matrix $\mathbf{M}$: (1) matrix $\mathbf{Y}$, the *standardized version* of $\mathbf{M}$. An element $y_{i,j}$ in the ith row and jth column from $\mathbf{Y}$ is $y_{i,j} = (m_{i,j} - \mathrm{avg}_j)/\mathrm{std}_j$, where $m_{i,j}$ is the element at the ith row and jth column in matrix $\mathbf{M}$ (i.e., the value of the jth metric at the ith time instance in the sliding widow of size t), while avg_j and std_j are the mean and the standard deviation for the j-th column in $\mathbf{M}$ respectively. (2) matrix $\mathbf{A}$: it contains all data considered abnormal in consecutive time instances from $(t_{now} - t)$ to t_{now}. (3) matrix $\mathbf{B}$: standardized version of $\mathbf{A}$ where each element in $\mathbf{B}$ is $b_{i,j} = (a_{i,j} - \mathrm{avg}_j)/\mathrm{std}_j$.

Our monitoring method has 5 steps: (1) process data from $\mathbf{M}$ to form $\mathbf{Y}$; (2) build the initial PCA model based on $\mathbf{Y}$; (3) do anomaly detection for every newest time instance data $\mathbf{z}$ using the latest PCA model; (4) if $\mathbf{z}$ is normal, move it to $\mathbf{M}$ and update the PCA model, and continue step 3; (5) if $\mathbf{z}$ is abnormal, move it to $\mathbf{A}$, and do *metrics identification* to find which metric(s) of which VM(s) might have caused this anomaly, meanwhile continue step 3. Next we will explain the key procedures in detail.

1) Build the PCA model: To build the PCA model, we perform eigenvalue decomposition on the covariance matrix of $\mathbf{Y}$, and get a set of eigen vectors $\mathbf{V} = (\mathbf{v}_1, \mathbf{v}_2, ..., \mathbf{v}_d)$ sorted by their eigen values $\lambda_1 \geq \lambda_2 \geq ... \geq \lambda_d \geq 0$.

We define the principal subspace and the residual subspace as follows. The principal subspace S stands for the space spanned by the first k principal axes in $\mathbf{V}$, while residual subspace $\widetilde{S}$ stands for the space spanned by the rest $(d-k)$ eigen vectors, which could be used to detect anomalies. Of numerous methods to determine k, we choose cumulative percent variance (CPV) method [15] for its ease of computation and good performance in practice. For the first ℓ principal components, $CPV(\ell) = (\sum_{i=1}^{\ell} \lambda_i / \sum_{i=1}^{d} \lambda_i) \cdot 100\%$, And we choose k to be: $k = \underset{\ell}{\mathrm{argmin}}(CPV(\ell) > 90\%)$.

2) Anomaly detection: Unlike previous methods that perform offline batched PCA anomaly detection, we only need to detect anomalies for the newest vector $\mathbf{z}$ at t_{now}. That's because we have classified data into the (normal) data matrix $\mathbf{M}$ and the abnormal matrix $\mathbf{A}$, and the real-time detection of ongoing anomalies is based on the PCA model built from $\mathbf{M}$. To do this, we first standardize $\mathbf{z}$ using the mean and standard deviation of each column in $\mathbf{M}$. We use $\mathbf{x}$ to denote the standardized vector.

Given the normal subspace $S : \mathbf{P_1} = [\mathbf{v}_1, ..., \mathbf{v}_k]$, and the residual subspace $\widetilde{S} : \mathbf{P_2} = [\mathbf{v}_{k+1}, ..., \mathbf{v}_d]$, $\mathbf{x}$ is divided into two parts by projecting on these two subspaces: $\mathbf{x} = \hat{\mathbf{x}} + \tilde{\mathbf{x}} = \mathbf{P_1}\mathbf{P_1}^T\mathbf{x} + \mathbf{P_2}\mathbf{P_2}^T\mathbf{x}$.

If $\mathbf{z}$ is normal, it should fit the distribution (e.g. mean and variance) of the normal data, and the values of $\tilde{\mathbf{x}}$ should be small. Specifically, the *squared prediction error* (SPE) is defined to quantify this:

$$\mathrm{SPE}(\mathbf{x}) = \|\tilde{\mathbf{x}}\|^2 = \left\|\mathbf{P_2}\mathbf{P_2}^T\mathbf{x}\right\|^2 = \left\|(\mathbf{I} - \mathbf{P_1}\mathbf{P_1}^T)\mathbf{x}\right\|^2.$$

$\mathbf{x}$ is considered to be *abnormal* if $\mathrm{SPE}(\mathbf{x}) > Q_\alpha$. The threshold Q_α is derived as:

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

where $\theta_i = \sum_{j=k+1}^{d} \lambda_j^i,\ i = 1, 2, 3;\ \ h_0 = 1 - \frac{2\theta_1\theta_3}{3\theta_2^2}$, and c_α is the $(1-\alpha)$ percentile in a standard normal distribution, with α being the false alarm rate [12].

Finally, if $\mathbf{z}$ is normal, we add it to $\mathbf{M}$ and delete the oldest data in $\mathbf{M}$, then update the PCA model accordingly. Otherwise it is added to $\mathbf{A}$, and the corresponding standardized $\mathbf{x}$ is moved to matrix $\mathbf{B}$.

3) Abnormal metrics identification: When an anomaly is detected, we need to do further analysis to identify *which metric(s) on which VM(s)* from the d dimensions might have caused the anomaly, to assist the orchestration module. Our identification method consists of 3 steps. It compares the abnormal data matrix $\mathbf{A}$ (and the corresponding standardized matrix $\mathbf{B}$), and normal matrix $\mathbf{M}$ (and $\mathbf{Y}$). Suppose there are m vectors in $\mathbf{A}$ ($\mathbf{B}$) and t vectors in $\mathbf{M}$ ($\mathbf{Y}$).

Step 1. Since the anomaly is detected by $\|\tilde{\mathbf{x}}\|^2$, it is natural to compare the residual data between $\mathbf{B}$ and $\mathbf{Y}$. Suppose $\mathbf{y_i}$ is the transpose of the i-th row vector in $\mathbf{Y}$, and $\tilde{\mathbf{y}}_\mathbf{i} = \mathbf{P_2}\mathbf{P_2^T}\mathbf{y_i}$ is its residual traffic, then $(\tilde{\mathbf{y}}_\mathbf{1}, \tilde{\mathbf{y}}_\mathbf{2}, ..., \tilde{\mathbf{y}}_\mathbf{t})^T = (\mathbf{P_2}\mathbf{P_2^T}(\mathbf{y_1}, \mathbf{y_2}, ..., \mathbf{y_t}))^T = \mathbf{Y}\mathbf{P_2}\mathbf{P_2^T}$ forms a residual matrix of Y, denoted as $\mathbf{Y_r}$. Similarly, $\mathbf{A_r} = \mathbf{A}\mathbf{P_2}\mathbf{P_2^T}$. For each dimension $j \in [1, d]$, let $a_j = \frac{1}{m}\left\|(\mathbf{A_r})_j\right\|^2$ and $y_j = \frac{1}{t}\left\|(\mathbf{Y_r})_j\right\|^2$, 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 $\mathrm{rd}_j = (a_j - y_j)/y_j$.

Step 2. If for some dimension j, $\mathrm{rd}_j \geq b_1$ for some constant b_1, we measure the change in $\mathbf{A}$ and $\mathbf{M}$. In particular, for each such dimension j, we calculate how much the abnormal data in $\mathbf{A}$ are away from the standard normal deviation of the normal data along that dimension in $\mathbf{M}$. Specifically, we calculate $\mathrm{stddev}_j = \frac{1}{m}\sum_{i=1}^{m} |\mathbf{a}_{ij} - \mathrm{avg}_j| / \mathrm{std}_j$. A dimension j is considered abnormal if $\mathrm{stddev}_j \geq b_2$ for some constant b_2. In practice, we find small positive integers work well for b_1 and b_2, say $b_1 = 2$ and $b_2 = 3$.

Step 3. For a dimension j that's been considered abnormal in Step 2, the difference between the mean of abnormal and normal data is measured. Specifically, we want to measure $\mathrm{meandiff}_j = (\frac{1}{m}\sum_{i=1}^{m} \mathbf{a}_{ij} - \mathrm{avg}_j)/\mathrm{avg}_j$.

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. Step 2 is a useful measure to show which dimension has a significant different pattern compared to the normal data, but 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 itself is not good enough to indicate a pattern, because the oscillation of metric usage statistics might make the mean of some dimension in matrix $\mathbf{A}$ appear benign. Thus, the output of steps 2 and 3 are sent together to the orchestration module on the corresponding NC(s). Section VI evaluates 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.

4) Interaction between tracking and monitoring: As mentioned earlier, the input data to the monitoring module are produced by the tracking module and each value may contain an approximate error of at most Δ. The approximation error introduced by the tracking module may degrade the detection accuracy of our monitoring module. ATOM allows user to set a deviation μ on detection accuracy, and the tracking error for the i-th metric is computed by: $\Delta_i = \mathrm{std}_i \cdot \sqrt{3}\sigma_i$, where std_i is the same standard deviation mentioned in section IV-C, and σ_i could be approximated using $2\sqrt{\frac{\bar{\lambda}}{t} \cdot \sum_{i=1}^{d} \sigma_i^2} + \sqrt{(\frac{1}{t} + \frac{1}{d}) \sum_{i=1}^{d} \sigma_i^4} = \epsilon$. $\bar{\lambda}$ is the average of eigen values, t is the number of points used to build PCA model, d is the number of dimensions, and σ_i^2 is the variance along each dimension. The deduction details will be available in the extended version of this paper.

V. Orchestration Component

Many VMI tools have been implemented and studied for IaaS systems. Our monitoring component provides VMI tools apriori knowledge of what might have gone wrong. It also serves as a trigger to tell VMI tools when and where to do introspection. With such information, the overhead of using VMI techniques is greatly reduced.

As an example, consider LibVMI and Volatility which are two open source VMI tools. Our orchestration module could use any such tools to find out which process might have caused the anomaly. 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 ready to use. By using Volatility together with PyVMI (a python wrapper for LibVMI) plugin, we can get rich information about a live VM and greatly facilitate the introspection.

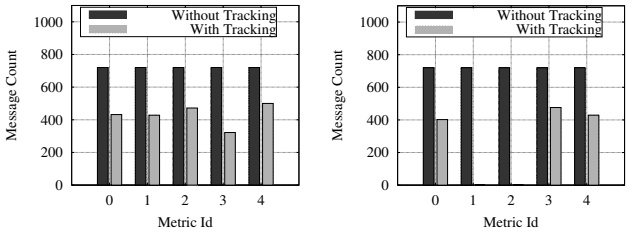
After the potentially abnormal processes are identified by VMI tools, an alarm is raised with associated abnormal information to the VM user. The number of rows in the abnormal data matrix $\mathbf{A}$ could be used as the alarm level to indicate the length of the ongoing anomaly. If user believes this is caused by normal workload change, then ATOM monitoring module will automatically add matrix $\mathbf{A}$ to $\mathbf{M}$ and adjust PCA model to the new workload accordingly. Otherwise if user confirms this as a malicious behavior, ATOM is able to terminate the malicious processes inside a VM by using tools like StackDB [13]. StackDB is designed to be a multi-level debugger, while also serves well as a memory-forensics tool. ATOM uses StackDB to debug, inspect, modify, and analyze the behavior of running programs inside a VM. 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.

To this end, many auto-debugging tools could be added, which is useful to find various kinds of attacks and perform different desirable actions. We refer these active actions, together with introspection, as ATOM's orchestration module. Orchestration in ATOM can be greatly customized to suit the needs for tasks such as identification of different attacks.

VI. Evaluation

We implement ATOM using Eucalyptus as the underlying IaaS system. The VM hypervisor running on each NC is the default KVM. Each VM has an m1.medium type on Eucalyptus. ATOM tracks 7 metrics from each VM: CPUUtilization, 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.

A. Online tracking



(a) VM: idle; Δ: 0

(b) VM: TPC-C on MySQL; Δ: average in 2 hours when VM is idle

Figure 2. A comparison on number of values sent by NC for each metric.

We set the data collection time interval at the NC (the observer) to 10 seconds, which produces 360 raw values for each metric per hour. Instead of sending every value to the CLC (the tracker), the modified CloudWatch with ATOM's online tracking component selectively sends certain values based on Algorithm 1. Figure 2 shows the number of values sent for each metric over 2 hours, with different workloads 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 both experiments.

Figure 2(a) shows the base case when VM is idle and $\Delta = 0$. The result shows that our tracking component has already achieved significant savings. Figure 2(b) shows the result 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 allowing no error, as Δ is merely the average of the amount consumed by an idle VM. Here NetworkIn and NetworkOut only have 2 values sent to the CLC. Clearly significant amount of values could be saved for the other metrics if more errors are allowed.

Figure 3 explains how the online tracking component works. It shows the NetworkOut values sent by standard CloudWatch (without tracking) and by modified CloudWatch with ATOM tracking, for a time interval of 1000 seconds. Here VM is idle and Δ is set to be 10% of the average value within last two hours. This clearly illustrates that at each time instance, with online tracking, the current (exact) value is not sent if it is within Δ of the last sent one; and at each time point, the last value sent to the CLC is always within Δ of the newest one observed on NC. The values sent by the tracking method closely approximate those exact ones, with much smaller overhead.

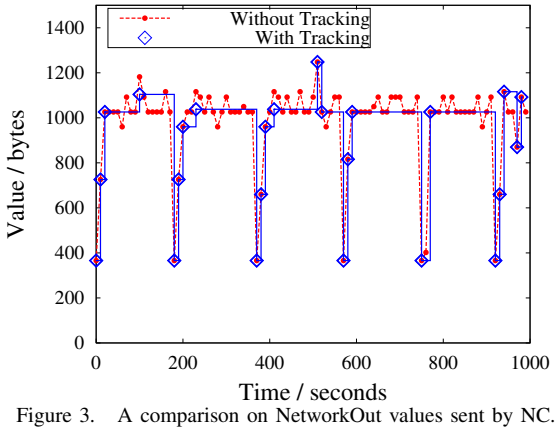
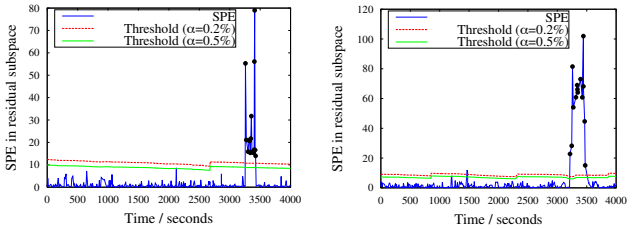


Figure 3. A comparison on NetworkOut values sent by NC.

B. Automated online monitoring and orchestration

We design two experiments to illustrate the effectiveness of ATOM's automated online monitoring module. The results are found not sensitive to false alarm rate α, deviation μ, sliding window size or collection interval. Here we show results of $\alpha = 0.2\%$ and deviation $\mu = 1\%$ (to set the tracking error bound). The Q_α threshold with $\alpha = 0.5\%$ is also calculated to compare against. Three VMs co-located in one Eucalyptus physical node are monitored with a 2-hour sliding window and 10-seconds collection interval for each experiment, forming a 720×21 data matrix. Dimensions $1-7$ belong to VM 1 and $8-14$ are for VM 2. There are also a 4-th VM running WebBench and a 5-th VM running Apache web server as the target of DDoS bots.



(a) Experiment 1: SPE and thresholds (b) Experiment 2: SPE and thresholds

Figure 4. Time series plots of SPE and Q_α with $\alpha = 0.2\%$ and 0.5%.

In the first experiment, both VM 2 and 3 run an Apache web server while VM 1 is idle. A DDoS attack turns both VM 2 and 3 to be zombies at the third hour, using them to generate traffic towards the target IP. This attack is hard to detect using the simple threshold approach [2] in the current setting because the normal workload on VM 2/3 already has a large amount of NetworkIn/NetworkOut usage. Sending out malicious traffic only changes roughly $10\%-30\%$ to the mean of normal statistics. Yet ATOM's monitoring module successfully finds the underlying pattern, and detects time instances that are abnormal. Figure 4(a) shows the online monitoring and detection process, where *the black dots indicate the time instances when DDoS attack happens*.

Once a time instance is considered abnormal, ATOM immediately runs metric identification procedure to find the affected VMs and metrics. As described in Section IV-C3, ATOM firstly analyzes rd_j, then for dimensions that have $\mathrm{rd}_j > 2$, ATOM computes if $\mathrm{stddev}_j > 3$, and also calculates the average change $\mathrm{meandiff}_j$ if so. The results are shown in the first table of table I. The gray italic numbers in this table need not to be calculated by ATOM because the residual portions on these dimensions are not large enough. We show them regardless for comparison. Among the 3 VMs being tracked and monitored, ATOM correctly identifies the same pattern of anomalies happening on VM 2 and 3, and more specifically, from their first three dimensions (CPUUtilization, NetworkIn, NetworkOut), indicated by the bold values. Note that NetworkIn and NetworkOut actually go down. Our guess is that WebBench tends to saturate the bandwidth available, while the DDoS attack we use launches many network connections but not sending as much traffic.

After abnormal metrics are identified, a VMI request is sent to the corresponding NC for introspection. Since both VMs have the same anomalous pattern, the introspection overhead could be saved by first introspecting one VM, and then checking if another one has the same malicious behav-

Experiment 1 Metrics Identification Results	Dim (j)	vm1-d1	vm1-d2	vm1-d3	vm1-d4	vm1-d5	vm1-d6	vm1-d7	vm2-d1	vm2-d2	vm2-d3	vm2-d4
	rd_j	23.70	-0.98	-0.98	-0.55	-0.57	4.27	3.76	9.14	64.18	65.05	3.50
	stddev_j	0.78	*0.42*	*0.58*	*0.00*	*0.67*	0.00	0.71	**3.17**	**8.01**	**8.30**	0.00
	$\mathrm{meandiff}_j$								0.16	-0.26	-0.28	
	Dim (j)	vm2-d5	vm2-d6	vm2-d7	vm3-d1	vm3-d2	vm3-d3	vm3-d4	vm3-d5	vm3-d6	vm3-d7	
	rd_j	-0.51	-0.82	4.23	9.04	60.56	61.16	1.45	-0.56	1.89	-0.51	
	stddev_j	*0.31*	*0.00*	0.35	**7.23**	**6.06**	**6.98**	*0.17*	*3.39*	*0.12*	*3.65*	
	$\mathrm{meandiff}_j$				0.39	-0.23	-0.31					
Experiment 2 Metrics Identification Results	Dim (j)	vm1-d1	vm1-d2	vm1-d3	vm1-d4	vm1-d5	vm1-d6	vm1-d7	vm2-d1	vm2-d2	vm2-d3	vm2-d4
	rd_j	2.58	-0.65	-0.93	-0.65	28.23	-0.98	-0.15	6.90	7.94	7.27	-0.76
	stddev_j	0.24	*0.42*	*0.63*	*0.95*	*0.43*	0.98	*0.86*	**7.36**	**4.52**	**4.74**	*0.21*
	$\mathrm{meandiff}_j$								-0.91	-0.85	-0.89	
	Dim (j)	vm2-d5	vm2-d6	vm2-d7	vm3-d1	vm3-d2	vm3-d3	vm3-d4	vm3-d5	vm3-d6	vm3-d7	
	rd_j	0.30	-0.99	-0.44	10.70	1282.80	1401.34	1363.47	-0.70	1544.73	-0.53	
	stddev_j	*1.41*	*0.17*	*1.43*	1.86	**13.05**	**12.79**	**13.42**	*1.72*	**13.60**	*1.78*	
	$\mathrm{meandiff}_j$					101.81	110.97	187.16		196.30		

Table I
METRICS IDENTIFICATION RESULTS FOR EXPERIMENTS IN FIGURE 4

ior going on. ATOM's orchestration module first identifies this as a possible network problem, and then calls Volatility to analyze the network connections on that VM, which then finds out the numerous network connections targeting at one IP address, a typical pattern of DDoS attacks. Volatility is then called again to find out related processes and their parent process of these network connections. At this time ATOM raises an alarm notifying user about the findings. If user identifies them to be malicious, he/she could either investigate the VM in further details, or use ATOM's orchestration module to do auto-debugging and kill malicious processes automatically through StackDB. Figure 4(a) shows that SPE goes back to normal after the attack is mitigated on the affected VM.

The second experiment illustrates ATOM's ability to detect resource-freeing attack [19], 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 on the target resource (often times with legitimate behavior). Strong VM isolation seems the only way to avoid such attacks. In this experiment, VM 2 runs an Apache web server constantly handling network requests. VM 3 runs TPC-C benchmark on MySQL database. In the third hour VM 3 launches GoldenEye attack [4], which achieves a DoS attack on the HTTP server running on VM 2 by consuming all available sockets, making network resource to be a bottleneck for VM 2, and shift its usage on cache.

Figure 4(b) plots the monitoring and the detection process. The black dots indicate the time instances when abnormal behavior happens. While the second table in table I analyzes where the anomaly has originated. The stddev_j values show that the abnormal dimensions are, on VM 2: CPUUtilization, NetworkIn, NetworkOut; on VM 3: NetworkIn, NetworkOut, DiskReadOps, DiskReadBytes. Further analysis on $\mathrm{meandiff}_j$ finds out NetworkIn and NetworkOut statistics on VM 2 has decreased nearly by an order, while VM 3 has a large increase in NetworkIn, NetworkOut, and more considerably, on DiskReadOps and DiskReadBytes. This is a typical resource freeing attack, 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 the sockets in VM 2 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 [19], yet ATOM is still able to raise an alarm to end user and suggests a possible ongoing resource-freeing attack.

C. Discussion

Possible false alarms. Resource usage change due to normal changes in user activities might cause ATOM to 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 could also use the new workload statistics to adjust its monitoring component dynamically and automatically in an online fashion.

Computation efficiency. The tracking module introduces only $O(1)$ computation overhead to each NC controller while resulting in substantial saving of communication. For the monitoring module, the PCA computation overhead is only polynomial of sliding windows size and number of dimensions. 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 monitored by CloudWatch and demonstrate two different types of attacks. But ATOM can be easily extended when necessary with additional metrics for monitoring without much overhead, and detecting different kinds of attacks.

VII. Related Work

Most existing IaaS systems follow the general, hierarchical architecture like AWS. Inside these systems, there are imperative needs for the controller to continuously collect resource usage data and monitor system health. AWS and Eucalyptus use CloudWatch [1] 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. OpenStack uses a service called Ceilometer [3] to collect measurements. However, as explained earlier, existing approaches only provide a discrete, sampled view of the system, and offer very limited capability in monitoring and ensuring system health. To the best of our knowledge, none of the 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.

Leading cloud providers have developed advanced mechanism to ensure the security of their IaaS systems, most of which are security policies. The security challenges in IaaS system were analyzed in [6], [18]. Virtual machine attacks are considered a major security threat. Performance issue (while ensuring security) is identified to be a key research challenge. Our work introduces novel monitoring method that has low overhead and good detection quality guarantees.

VMI has been a well-known method for ensuring VM security [7], [8]. It has also been studied for IaaS systems [10]. However, all of these systems require the VM to be suspended to gain access to its memory, resulting in user programs negatively affected. Thus a monitoring method to know when and where to trigger a VMI would save much overhead. Another solution was suggested for cloud users to verify the integrity of their VMs [5], which however is not an "active detection and reaction" system.

Lastly, PCA has been widely used to detect anomaly in network traffic volume in backbone networks [9], [14], [16]. As we have argued in Section IV-A, adapting a PCA-based approach to our setting has not been studied before and it presents significant new challenges.

VIII. 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 resource monitoring and anomaly detection in IaaS systems. Interesting future work include extending ATOM for more sophisticated resource orchestration and incorporating the defense against even more complex attacks.

IX. Acknowledgment

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.

References

[1] AWS CloudWatch. http://aws.amazon.com/cloudwatch/.
[2] AWS CloudWatch Alarm. http://docs.aws.amazon.com/AWSCloudFormation/latest/UserGuide/aws-properties-cw-alarm.html.
[3] Ceilometer. https://wiki.openstack.org/wiki/Ceilometer.
[4] Goldeneye attack. https://github.com/jseidl/GoldenEye.
[5] B. Bertholon, S. Varrette, and P. Bouvry, "Certicloud: a novel tpm-based approach to ensure cloud iaas security," in *IEEE Cloud Computing*, 2011.
[6] W. Dawoud, I. Takouna, and C. Meinel, "Infrastructure as a service security: Challenges and solutions," in *INFOS*, 2010.
[7] B. Dolan-Gavitt, T. Leek, M. Zhivich, J. Giffin, and W. Lee, "Virtuoso: Narrowing the semantic gap in virtual machine introspection," in *IEEE S&P*, 2011.
[8] T. Garfinkel and M. Rosenblum, "A virtual machine introspection based architecture for intrusion detection." in *NDSS*, 2003.
[9] L. Huang, M. I. Jordan, A. Joseph, M. Garofalakis, and N. Taft, "In-network pca and anomaly detection," in *NIPS*, 2006.
[10] A. S. Ibrahim, J. Hamlyn-Harris, J. Grundy, and M. Almorsy, "Cloudsec: a security monitoring appliance for virtual machines in the iaas cloud model," in *NSS*, 2011.
[11] ITWORLD. http://www.itworld.com/security/428920/attackers-install-ddos-bots-amazon-cloud-exploiting-elasticsearch-weakness.
[12] J. E. Jackson and G. S. Mudholkar, "Control procedures for residuals associated with principal component analysis," *Technometrics*, vol. 21, no. 3, 1979.
[13] D. Johnson, M. Hibler, and E. Eide, "Composable multi-level debugging with Stackdb," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, 2014.
[14] A. Lakhina, M. Crovella, and C. Diot, "Diagnosing network-wide traffic anomalies," in *ACM SIGCOMM Computer Communication Review*, 2004.
[15] W. Li, H. H. Yue, S. Valle-Cervantes, and S. J. Qin, "Recursive pca for adaptive process monitoring," *Journal of process control*, vol. 10, no. 5, 2000.
[16] Y. Liu, L. Zhang, and Y. Guan, "Sketch-based streaming pca algorithm for network-wide traffic anomaly detection," in *ICDCS*, 2010.
[17] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *CCGRID*, 2009.
[18] L. M. Vaquero, L. Rodero-Merino, and D. Morán, "Locking the sky: a survey on iaas cloud security," *Computing*, 2011.
[19] V. Varadarajan, T. Kooburat, B. Farley, T. Ristenpart, and M. M. Swift, "Resource-freeing attacks: improve your cloud performance (at your neighbor's expense)," in *CCS*, 2012.
[20] K. Yi and Q. Zhang, "Multi-dimensional online tracking," in *SODA*, 2009.