International Journal of Cloud Computing and Services Science (IJ-CLOSER)
Vol. x, No. x, May 2013, pp. 1 – 14
ISSN: 2089-3337

Cloud CPU Verification Scheme for Individual End Users

Huanyang Zheng[*], Kangkang Li[*], Adam Blaisse[*], Chiu C. Tan[*], and Jie Wu[*]
[*]Department of Computer and Information Sciences, Temple University, USA

Article Info

Article history:
Received Jun 20th, 2014
Revised March 10th, 2014
Accepted March 20th, 2014

Keyword:
Cloud computing
Cheating
CPU verification
Time-lock puzzle

ABSTRACT

In this paper, a user-based CPU verification scheme is proposed for the cloud cheating detection problem, where the cloud service provider offers insufficient CPU resources that are bought by the user. In this scheme, a predefined computational task is constructed for the cloud to execute. Then, we compare the difference of the actual execution time (recorded by the user) and the theoretical execution time, as to determine whether the cloud is cheating or not. A time-lock puzzle is introduced to construct the predefined computational task, so that this task is guaranteed to be fully executed by the cloud. Our cheating detection process has a higher probability of detecting cloud cheating if using a larger predefined computational task, which in turn costs more time. Further analysis shows that, if the total detection time is limited, it is better to detect cloud cheating using small-scale and short-length cheating detecting processes multiple times, as opposed to large-scale and long-length processes a few times. We also discuss the heterogeneity of CPU resources through two simple models. Finally, the feasibility and validity of our scheme is shown in the real system evaluations.

Corresponding Author:
Huanyang Zheng
Department of Computer and Information Sciences, Temple University, USA
Wachman Hall 1015E, 1805 N. Broad Street, Philadelphia, PA 19122-6094
215-204-9259
huanyang.zheng@temple.edu

1. INTRODUCTION

Scalability and metering are two popular features among users of commercial cloud computing services, because they allow users to reduce their operating costs [1, 2, 3]. A user operating a video sharing service based on a commercial cloud provider can, for instance, purchase fewer computing resources during a period of low demand, while rapidly scaling to more computing resources in times of high demand, resulting in higher monetary savings. The cloud computing service provider is able to offer this type of service by sharing its hardware among multiple users. Through virtualization technology, each client's computation jobs are encapsulated within a virtual machine (VM). The cloud provider is able to have multiple VMs share the same hardware, and then migrate the VMs to other physical machines when the current machine is unable to provide the required amount of resources [4, 5, 6, 7]. The user's VM running the video sharing service, from the previous example, could be sharing the same physical hardware with several other users during his period of low demand. This would require that the cloud provider migrates that VM to a separated piece of hardware when more computing resources are required [8, 9, 10].

The cloud provider should provide the amount of computing resources that an individual end user has paid for. However, since the cloud provider is both the entity providing the resources as well as that metering the service and billing the user, this opens up the possibility that the cloud provider may not provide the computing resources that the user has bought. Certain types of resources, such as storage space, can be easily verified by the user. The user can simply attempt to upload a file of a certain size and retrieve it later. However, other computing resources, like CPU, are more difficult to verify. For instance, instead of allocating 10 VMs to a physical machine to ensure sufficient CPU times for all VMs, a malicious cloud allocates 11 VMs to that same machine, saving the operation costs of an additional machine. This type of cheating may even occur when the cloud is *not* malicious, but instead is due to errors in the VM migration code or algorithm, or due to the heterogeneity of the underlying hardware [11]. Actually, according to the service level agreements [12], insufficient CPU resources may also come as a result from some irresistible reasons (e.g., natural disaster and warfare). In this paper, we assume that irresistible reasons for insufficient CPU resources do

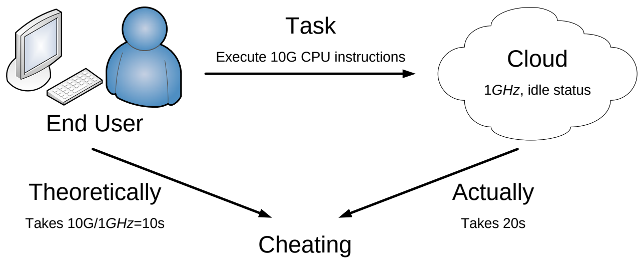


Figure 1. Example of the cheating detection.

not exist, and then consider the problem of allowing users to test and verify their allocated CPU resources [13].

Our proposed solution lets the user check the amount of CPU resources, through measuring the time duration for the cloud to complete a predefined computing task (PCT). Fig. 1 illustrates this checking process. First, the user requires the cloud to execute a PCT, which needs 10^{10} CPU instructions to complete. Then, the user records the actual executing time of the PCT, which is $20s$. Meanwhile, according to the committed CPU frequency, the user can theoretically predict the execution time as $10^{10}/1GHz = 10s$. Since the actual execution time is much longer than the theoretical execution time, the user can then judge that the cloud is cheating (i.e., the user receives insufficient CPU resources). The above process is called the cheating detection process, which addresses the following challenges.

- How can we guarantee that the PCT is indeed executed by the cloud? Instead of actually running the PCT, the cloud can predict the execution time of the PCT, and then report completion at the expected task finishing time. Even if the PCT returns parameters to verify its execution, the cloud may simplify the PCT to speed up the task execution (e.g., repeating $a = a + 1$ one thousand times can be simplified to run $a = a + 1000$ directly).
- Since the user purchased the cloud for computation rather than detection, the cloud may not be idle. How do we deal with the cheating detection process when there are some background programs assigned by the user?
- Generally speaking, a larger PCT resists interferences better, and thus, has a higher performance on cheating detection. However, a larger PCT costs more time as a tradeoff. If the total detection time is limited, should we use small-scale PCT more times, or large-scale PCT fewer times?
- As previously mentioned, the heterogeneity of the underlying hardware [11] leads to fluctuated CPU resources that are allocated to the user. Can we model such a fluctuation? Moreover, how does such a fluctuation influence our cheating detection process in real cloud systems?

The rest of the paper is organized as follows. First, we present the cheating problem, and show the monitoring architecture in Section 2. Then, the detailed cheating detection process is described in Section 3. We analyze the error control parameter in Section 4. Extensions and discussions on the fluctuation of the CPU resources are described in Section 5. Then, real system evaluation is conducted in Section 6. Finally, we conclude this paper in Section 7.

2. END USER CPU MONITORING

In this section, we first present the cloud cheating problems from different perspectives, including the mathematical definitions of cloud cheating and cheating detection, assumptions of the cloud and the user, etc. Then, we show the monitoring architecture: the components involved in the cheating detection process, the interactions between the cloud and the user, and the cheating determination.

2.1. Problem Formulation

Let CPU_C and CPU_R denote the committed CPU frequency of the cloud brought by the user and the real CPU frequency of the cloud, respectively. Here, we first consider a simple case, where CPU_C and CPU_R are stable and fixed values. A more general and practical case, where CPU_R varies with respect to the time due to the hardware heterogeneity [11], is discussed in Section 5. Then, the cloud cheating is defined as the following:

$$CPU_R < CPU_C - \varepsilon \tag{1}$$

where ε is a parameter for the error control. Another representation of Eq. 1 is

$$t_a > t_t + \delta \tag{2}$$

where t_a is the actual time of executing a task which needs in total I CPU instructions to complete, and t_t is the theoretical time of executing the same task. Ideally, if the cloud CPU is idle, we have $t_a = I/CPU_R$ and $t_t = I/CPU_C$. Similar to ε in Eq. 1, δ is also an error control parameter. Considering that Eq. 2 is closer to the *user experience* than Eq. 1, we use Eq. 2 as the cheating definition. The insight behind this definition is that the user is unsatisfied if he waits for an abnormal task execution time that is significantly longer than his prediction.

Since the user bought the cloud for some computational tasks rather than detections, we assume that resources for detection are limited: the whole running time of the detection program should be less than D seconds (i.e., the detection budget). The cloud is running some background programs, the total CPU usage of which is stably $x\%$. In addition, we assume that cloud is as smart as humans in doing any possible anti-detection action. For example, the cloud would present the committed CPU frequency, rather than its real CPU frequency, in the OS. We also assume that the user has a reliable local PC with CPU frequency CPU_L and a timer for detection assistance. Small interferences, such as network transmission delay, are neglected.

2.2. Monitoring Architecture

Since the cloud is able to do any given anti-detection action, the cheating detection has to depend on reliable parameters provided by the user. Among all these parameters, time is the most convenient one, which also serves as our objective. Therefore, we decided to construct a special PCT in the user's local PC for the cloud to execute, and to observe the time difference between t_a and t_t in executing the PCT to determine whether the cloud is cheating or not.

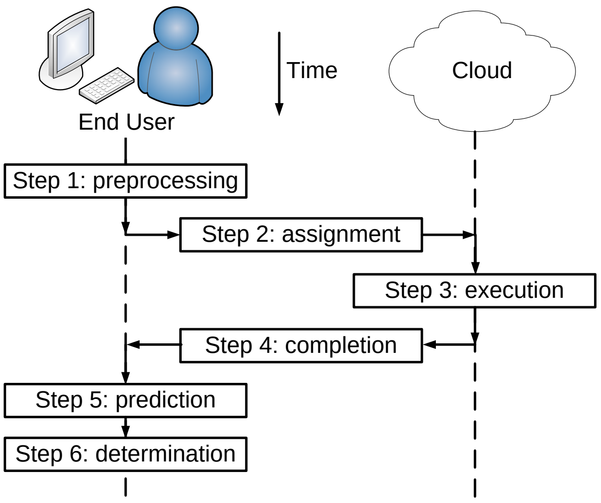


Figure 2. The cheating detection process.

The proposed cheating detection framework is depicted in Fig. 2. In step 1, the user constructs the PCT (totally $T * CPU_L$ CPU instructions) in the local PC. The PCT has some random inputs for the initialization, and returns an output (or say answer, denoted as a_L) at the end of the program. In step 2, the user starts the timer and requires the cloud to execute the PCT with the same inputs in the local PC. In step 3, the cloud executes the PCT. In step 4, when the cloud completes the PCT, it returns the output (denoted as a_C) to verify that the PCT has been executed. Meanwhile, the user stops the timer to obtain the actual task execution time in the cloud (denoted as t_a). In addition, the cloud also returns the parameters of its CPU usage percentage of the background programs before running the PCT (denoted as $x\%$). In step 5, based on the amount of calculations of the PCT ($T * CPU_L$ CPU instructions) and the parameters on the computational resources of the cloud ($x\%$ and CPU_C), the user can predict the theoretical task execution time of the PCT (denoted as t_t). In step 6, the cheating determination is done based on a_L, a_C, t_a, t_t, and the error control parameter δ. $a_L = a_C$ represents that the PCT is executed completely in the cloud. $t_a > t_t + \delta$ determines that the cloud is cheating. In the next section, we will show more details on the cheating detection process, including the construction of the PCT, the calculation of t_t, and so on.

3. MONITORING ALGORITHMS

In this section, we show the details of the monitoring algorithms. First, we introduce the PCT and its characteristics. Second, we present the method for theoretical task execution time calculation. Third, we discuss the cheating determination and error control. Finally, the complete algorithm is shown. All parameters are shown in Table 1.

Table 1. Parameters Involved in this Paper

Variable	Description
CPU_L	The CPU frequency of the local PC.
CPU_C	The committed CPU frequency of the cloud.
CPU_R	The real CPU frequency of the cloud, with its initial value being CPU_0.
p, q, n	p and q are two large random prime numbers, while $n = pq$.
$\phi(n)$	Calculated by $\phi(n) = (p-1)(q-1)$.
b	A large random number which is relatively prime to n, i.e., $gcd(b, n) = 1$.
S	The number of computing $b = b^2\ mod\ n$ per second in local PC using 100% CPU.
T	A time parameter in seconds, while the PCT has in total $T * CPU_L$ CPU instructions.
M	Calculated by $M = TS$. Computing $b^2\ mod\ n$ for M times takes T seconds in local PC.
a_L, a_C	Respectively calculated by $a_L = b^{2^M}\ mod\ n$ in the local PC and $a_C = b^{2^M}\ mod\ n$ in the cloud.
$x\%$	The background program CPU usage in the cloud, before running the cheating detection program.
$y\%$	The background program CPU usage in the cloud, when running the cheating detection program.
$z\%$	The cheating detection program CPU usage in the cloud.
t, t_1, t_2	t is the time. t_1 and t_2 are the start and finish time of the cheating detection process, respectively.
t_a	The actual time duration of running the cheating detection program in the cloud.
t_t	The theoretical time duration of running the cheating detection program in the cloud.
I, δ	I is used to define the cloud cheating in Eq. 2, while δ is the error control parameter.
D	The total time budget of the cheating detection process.
T_{min}	Calculated by $T_{min} = I/CPU_L$, the lower bound of parameter T.
T_{max}	Calculated by $T_{max} = D/(1 + x\%) * CPU_C/CPU_L$, the upper bound of parameter T.
$g(t)$	A Gaussian Process that is used to model CPU_R.
$f(\cdot)$	A monotonic decreasing function that represents the system control mechanism.

3.1. Predefined Computational Task

The PCT includes $T * CPU_L$ CPU instructions in total (we have $T * CPU_L > I$ to satisfy the cheating definition in Eq. 2). This task is executed on both the cloud and the local PC. To ensure that the cloud runs the PCT, the PCT should return an answer. Let a_C and a_L denote the answer from the cloud and the local PC, respectively. If $a_C = a_L$, the PCT is executed completely in the cloud. Since we compare the actual and theoretical time duration of the task execution to determine cheating, the PCT should not be simplified. For example, if the PCT is to repeat $a = a + 1$ one thousand times (uses $a = 0$ for initialization and returns $a = 1000$), the cloud could replace this task with $a = a + 1000$ as a simplification. A time-lock puzzle is introduced to the PCT to solve this problem, which can be viewed as an application of the random-access property of the Blum-Blum-Shub $b^2\ mod\ n$ pseudo-random number generator [14, 15]. The insight is that this time-lock puzzle cannot be further simplified for the computation.

Theorem 1 (Time-Lock Puzzle Theorem) *Assume a large number b is relatively prime to a large composite number n, without factoring n; the quickest method to solve $b^{2^M}\ mod\ n$ (M is an arbitrary natural number) is to loop $b = b^2\ mod\ n$ for M times (returns b as the outcome).*

The Time-Lock Puzzle Theorem is proven in [16, 17]. If factoring n takes too much time, then calculating $b^{2^M}\ mod\ n$ cannot be simplified. If n satisfies $n = pq$, where p and q are two random prime numbers that are large enough, then factoring n is costly. However, if p and q are known, $a = b^{2^M}\ mod\ n$ can be efficiently calculated by

$$a = b^e mod\ n \quad \text{where} \quad e = 2^M mod\ \phi(n) \quad \text{and} \quad \phi(n) = (p-1)(q-1) \tag{3}$$

Therefore, calculating $b^{2^M}\ mod\ n$ is used as the PCT, which takes $T*CPU_L$ CPU instructions if p and q are unknown. Constructing the PCT in the local PC is shown in Algorithm 1 (step 1 in Fig. 2), and the process of executing the PCT in the cloud is shown in Algorithm 2 (step 3 in Fig. 2). Note that steps 2 and 4 are used to measure the execution time of Algorithm 2 in the cloud. In the next subsection, we will discuss how to calculate t_t, which is step 5 in Fig. 2.

3.2. Theoretical Task Execution Time

Assume $x\%$, $y\%$, and $z\%$, respectively, present the background program CPU usage in the cloud before running the cheating detection program, the background program CPU usage in the cloud when running the cheating

Algorithm 1 Constructing The PCT in The Local PC

```
Input:  Parameter $T$;
Output: Parameters $b$, $M$, $n$, and $a_L$;
 1: Close all background programs in the local PC;
 2: Generate two large random prime numbers $p$ and $q$;
 3: Calculate $n = pq$;
 4: Generate a large random number $b$ that $gcd(b, n) = 1$;
 5: Set $S = 0$;
 6: while $loop\ time < 1\ second$ do
 7:     $b = b^2\ mod\ n$;
 8:     $S = S + 1$;
 9: Calculate $M = ST$;
10: Calculate $a_L = b^{2^M}\ mod\ n$ efficiently by Eq. 3;
11: Return $b$, $M$, $n$, and $a_L$;
```

Algorithm 2 Executing The PCT in The Cloud

```
Input:  Parameter $b$, $M$, $n$;
Output: Parameters $x\%$ and $a_C$;
 1: Get background program CPU usage $x\%$;
 2: for $count = 1\ to\ M$ do
 3:     $b = b^2\ mod\ n$;
 4: Set $a_C = b$;
 5: Return $x\%$ and $a_C$;
```

detection program, and the cheating detection program CPU usage in the cloud. We further assume that the background programs and the detection program have stable CPU usage, which does not change over time. Obviously, $y\% + z\% = 100\%$. However, the relationship between $x\%$ and $y\%$ is not simply $x\% = y\%$. If running the background programs alone, it takes $x\%$ CPU usage. If running the detection program alone, it takes 100% CPU usage. As far as we know, in most OS, when running the background programs and the detection program together, they share the CPU proportionally to the CPU usage that they take when running alone. Consequently, we have

$$y\% = \frac{x\%}{x\% + 100\%} \quad \text{and} \quad z\% = \frac{100\%}{x\% + 100\%} \tag{4}$$

Note that the PCT includes $T * CPU_L$ CPU instructions in total. If $x\% = 0\%$ (i.e., no background programs in the cloud), executing $T * CPU_L$ CPU instructions in the cloud should take $T * CPU_L/CPU_C$ seconds. Thus,

$$t_t = T * \frac{CPU_L}{CPU_C} \tag{5}$$

Taking background programs into consideration, then the theoretical task execution time can be calculated as follows:

$$t_t = T * \frac{CPU_L}{CPU_C * z\%} = T * (1 + x\%) * \frac{CPU_L}{CPU_C} \tag{6}$$

Note that, the theoretical task execution time (t_t) obtained in Eq. 6 corresponds to step 5 in Fig. 2. In the next subsection, we show the method of determining cloud cheating, which corresponds to step 6 in Fig. 2.

3.3. Cheating Determination

To determine whether the cloud is cheating or not, the first step is to check that, the PCT is executed correctly in the cloud. If the PCT is executed completely, we should have $a_C = a_L$, since a_C and a_L are the answers to the same PCT. Then we use Eq. 2 to judge cheating: if $t_a > t_t + \delta$, the cloud is judged to be cheating; if $t_a \leq t_t + \delta$, the cloud is not cheating. The cheating determination process has been shown in Algorithm 3.

However, determining the error control parameter δ remains to be a major challenge. Ideally, if there is no interference, δ could be set to 0, and t_t should be strictly equal to t_a. But interferences indeed exist. Since t_t is calculated by T, $x\%$, CPU_L and CPU_C (where δ is the error control parameter), δ may be related to the same four parameters. How to set δ is further discussed in Section 4.

Algorithm 3 Cheating Determination

```
Input:  Parameter $a_C$, $a_L$, $t_a$, $t_t$, and $\delta$;
Output: Whether the cloud is cheating or not;
1: if $a_C \neq a_L$ then
2:    Return cheating;
3: else
4:    if $t_a > t_t + \delta$ then
5:       Return cheating;
6:    else
7:       Return no-cheat;
```

Algorithm 4 The Whole Cheating Detection Process

```
Input:  Parameter $T$, $D$, $x\%$, $CPU_L$, $CPU_C$;
Output: Whether the cloud is cheating or not;
1: Calculate $T_{max} = D/(1 + x\%) * CPU_C/CPU_L$;
2: for $count = 1$ to $\lfloor T_{max}/T \rfloor$ do
3:    Use Algorithm 1 to construct the PCT in the local PC;
4:    Start timer;
5:    Use Algorithm 2 to execute the PCT in the cloud;
6:    Stop timer to get the actual task execution time ($t_a$);
7:    Calculate the theoretical task execution time $t_t$ by Eq. 6;
8:    if Algorithm 3 judges the cloud to be cheating then
9:       Return cheating;
10: Return no-cheat;
```

3.4. The Whole Algorithm

The former three subsections introduce the cheating detection process based on Fig. 2, which is predicted to take t_t seconds. However, as previously mentioned, we have in total D seconds for the cheating detection, which requires $t_t < D$. According to Eq. 6, we have

$$T < \frac{1}{1 + x\%} * \frac{CPU_C}{CPU_L} * D = T_{max} \tag{7}$$

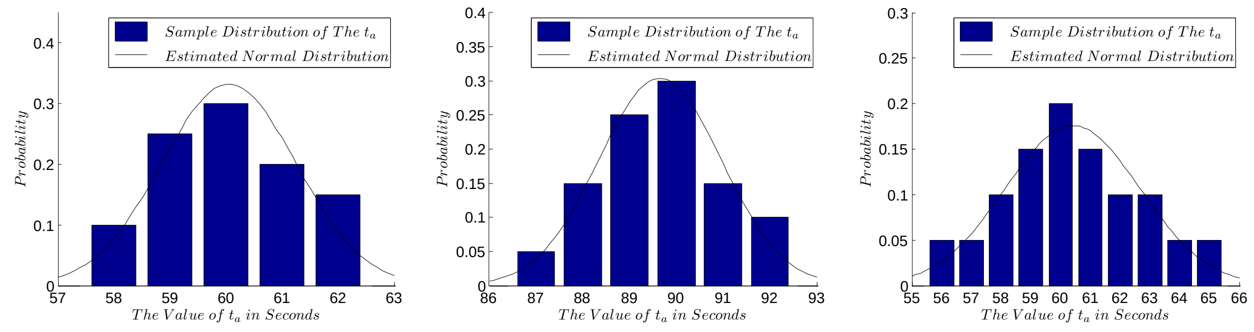
Note that if $T \ll T_{max}$ ($t_t \ll D$), then the remaining time is wasted. A better method is to iteratively run the cheating detection process until all D seconds are used up (totally $\lfloor T_{max}/T \rfloor$ rounds). Once the cloud is detected as cheating, then we judge the cloud to be cheating. For example, if $T = 0.1 * T_{max}$, then we can run the cheating detection process in Fig. 2 for 10 times: only if the cheating detection processes return no-cheat all 10 times is the cloud judged to be no-cheat. This method is called *multi-round cheating detection*, the whole algorithm of which is presented in Algorithm 4. Meanwhile, the lower bound of the T is given by the definition of cheating in Eq. 2:

$$T > \frac{I}{CPU_L} = T_{min} \tag{8}$$

Obviously, the value of T has a great influence on the performance of the cheating detection algorithm, since $\lfloor T_{max}/T \rfloor$ determines the loop times. Intuitively, a larger T brings fewer loops for the cheating detection process, while each loop has higher accuracy. The value of T, along with the error control parameter, is further discussed in the next section.

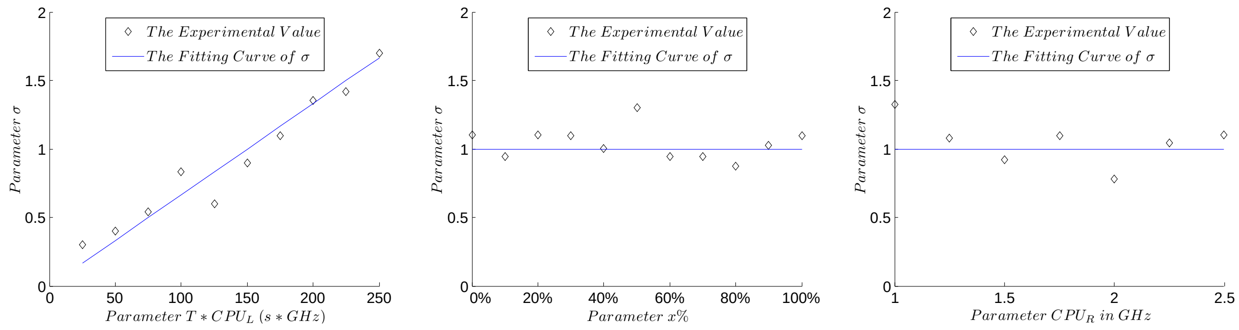
4. ERROR CONTROL PARAMETER

In this section, we discuss the error control parameter δ in Eq. 2, where we use $t_a > t_t + \delta$ to determine cheating. Due to some small interferences (for example, some OS burst processes), the actual task execution time t_a varies. Though ideally $t_a = t_t$ if the user obtains the committed CPU in the cloud, in practice, t_a is only approximated to t_t. Therefore, it is necessary to model the distribution of t_a to determine the value of error control parameter δ, which is presented in the first subsection. Based on the value of δ, we then discuss the value of T to see which scheme is better (since the total detection time is limited to budget D): a small-scale and short-length cheating detecting process for more times, or a large-scale and long-length cheating detecting process for fewer times.



(a) $CPU_R = 2.5GHz$ with $x = 00\%$ (b) $CPU_R = 2.5GHz$ with $x = 50\%$ (c) $CPU_R = 1.0GHz$ with $x = 00\%$

Figure 3. Estimating t_a to follow normal distribution ($T = 60$ and $CPU_L = CPU_R$).



(a) $CPU_R = 2.5GHz, x = 00\%$ (b) $T = 60s, CPU_L = CPU_R = 2.5GHz$ (c) $T = 60s, CPU_L = 2.5GHz, x = 00\%$

Figure 4. Analysis of parameter σ.

4.1. The Variance of The Actual Task Execution Time

Due to the interference, the t_a randomly varies, which is modeled to follow the normal distribution (μ, σ). Here, μ and σ are the expectation and standard deviation of t_a, respectively. Since the 99% *confidence interval* of the normal distribution is $[\mu - 3\sigma, \mu + 3\sigma]$, we employ t_t to estimate μ and $\delta = 3\sigma$ as the error control parameter (if $t_a > t_t + \delta$, the cloud is judged to be cheating). Experiments are conducted to check whether the estimation of normal distribution is feasible or not, where we test the actual task execution time of a PCT that includes $T * CPU_L$ instructions. The detailed environment setup of this is shown later in Section 6. Then, we collected 20 sampling points of t_a in each test, the distribution of which is called the sample distribution. The maximum likelihood estimate (MLE) is employed to estimate the distribution of these sampling points, i.e., the estimated normal distribution. The results in Fig. 3 shows the feasibility of estimating t_a to follow the normal distribution.

Now, we need to estimate the value of μ and σ, if no cheating happens. Obviously, μ should be equal to t_t if the cloud is not cheating. Intuitively, σ can be estimated by parameters M, $x\%$ and CPU_R: M is the input of Algorithm 2, i.e., the PCT computes $b^2 \; mod \; n$ for M times ($T * CPU_L$ instructions in total); $x\%$ and CPU_R describes the computational capabilities of the cloud. Fig. 4 shows the relationship between σ and these parameters (the σ is calculated through MLE). It can be seen that σ is almost linearly proportional to $T * CPU_L$, while σ is almost uncorrelated to parameter $x\%$ and CPU_R. Thus, the fitting curve of σ is

$$\sigma = \frac{T}{60} * \frac{CPU_L}{2.5} = \frac{T * CPU_L}{150} \tag{9}$$

where the unit of T is second, and the unit of CPU_L is GHz. Note that, Eq. 9 is only an empirical estimation that may be sensitive to the cloud environment, i.e., it may not be the same for different cloud systems. Based on Eq. 9, we further discuss how to set parameter T in the next subsection.

4.2. Analysis on The Cheating Detection Probability

In the former subsection, we modeled t_a to follow normal distribution (μ, σ) due to the interferences. For a single-round cheating detection process, we employ condition $t_a > t_t + 3\sigma$ to judge cloud cheating. If $CPU_R \neq$

CPU_C, the probability to successfully detect cheating is

$$P = \int_{t_t+3\sigma}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt = \frac{1}{2} erfc(\frac{t_t + 3\sigma - \mu}{\sqrt{2}\sigma}) \tag{10}$$

where we have $\mu = T * (1 + x\%) * \frac{CPU_L}{CPU_R}$ and $\sigma = \frac{T*CPU_L}{150}$. Then, Eq. 10 can be simplified to be

$$P = \frac{1}{2} erfc[\frac{3 + 150(1 + x\%)(1/CPU_C - 1/CPU_R)}{\sqrt{2}}] \tag{11}$$

The cheating detection probability of a single-round cheating detection process has been shown in Eq. 11. Assume $x\% = 0\%$ and $CPU_C = 2.5GHz$, we have $P = 100\%$ if $CPU_R = 2.2GHz$, $P = 98.7\%$ if $CPU_R = 2.3GHz$, and $P = 30.9\%$ if $CPU_R = 2.4GHz$. If the cloud is not cheating ($CPU_R = CPU_C = 2.5GHz$), the probability of erroneous judgement is $P = 0.1\%$. Then, the probability of successful cheating detection using time budget D is

$$P_d = 1 - (1 - P)^{\lfloor \frac{T_{max}}{T} \rfloor} = 1 - (1 - P)^{\lfloor \frac{1}{1+x\%} * \frac{CPU_C}{CPU_L} * \frac{D}{T} \rfloor} \tag{12}$$

Eq. 12 shows the final cheating detection probability using $\lfloor \frac{T_{max}}{T} \rfloor$ rounds, which has been previously mentioned in Eq. 7. Note that P_d is increased with decreasing T. Considering that the PCT includes $T * CPU_L$ instructions in total and CPU_L is a fixed value, we have the following position:

Position 1 (Time Assignment Position) *Based on our error control model, it has a higher probability of detecting cloud cheating using small-scale and short-length cheating detecting processes multiple times, as opposed to using large-scale and long-length cheating detecting processes a few times.*

However, our model assumes that there are no other interferences (for example, network transmission delay), which would lead to errors when T is too small. Another point is that, the background programs generally occupy CPU erratically; because of this, we suggest running the cheating detection process when the cloud is idle ($x\% = 0\%$).

5. EXTENSIONS AND DISCUSSIONS

In former sections, we have considered a case, in which the real CPU frequency of the cloud (i.e., CPU_R) is a stable and fixed value. Meanwhile, the detection has a certain probability, since the actual task execution time (i.e., t_a) may vary, as shown in Fig 3. In this section, we will consider a more general and practical case, where CPU_R varies with respect to the time (i.e., not fixed). Actually, most of the current cloud systems cannot offer completely stable CPU resources, due to the heterogeneity of the underlying hardware [11]. Therefore, to fulfill the user's payment, a very common method for the cloud service provider is to offer *bounded* CPU resources. In other words, instead of offering a fixed CPU_R, the cloud service provider offers an unstable CPU resource that is always in the range of $[CPU_C^L, CPU_C^U]$. Here, CPU_C^L and CPU_C^U stand for the lower and upper bound of the committed CPU resource, respectively ($CPU_C^L \leq CPU_C \leq CPU_C^U$). At this time, if we only want to check whether the could is cheating or not, then the previously described detection scheme can still be used; the only difference is to focus on verifying that the cloud provides a CPU resource of at least CPU_C^L, rather than verifying CPU_C. However, a more interesting problem comes out as follows. Since our detection scheme actually checks the *average* CPU resource, it does not necessarily mean that the provided CPU resource is always at least CPU_C^L *over the time duration.* As shown in Fig. 5, although the average CPU_R over the cheating detection time duration is larger than its promised lower bound, cheating may still happens during some time slots.

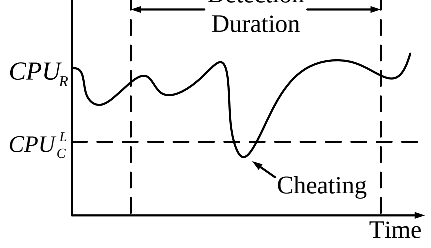


Figure 5. CPU_R varies with respect to time.

In this section, we will give some assumptions on the functional relationship between CPU_R and the time, i.e., assumptions on how CPU_R changes with respect to time. Then, we will *quantitatively* discuss the detection probability that can be obtained by our scheme. Here, we focus on a theoretical case where the system noise is ignored, i.e., $\sigma = 0$ and $t_a = \mu$. However, various starting times of the cheating detection process may lead to different actual task execution time (i.e., different t_a), since the actual CPU resource varies with respect to time. For simplicity, we also assume that the cloud is idle ($x\% = y\% = 0\%$) and is fully utilized by the detection program ($z\% = 100\%$). In other words, we assume t_a to be equal to the average CPU_R over the cheating detection time duration. In addition, in this section, t_t is calculated through CPU_C^L rather than CPU_C. Then, we will start with a Gaussian process to model the relationship between CPU_R and the time.

5.1. Gaussian Process

First, we discuss the Gaussian process model, where the change of CPU_R follows a Gaussian process. The reason for using this model is based on previous experiments shown in Fig. 3: the actual execution time of a certain task follows normal distribution; therefore, the CPU resource may be approximated to follow the same distribution. Let t denote the time, then we have

$$\frac{\mathrm{d}CPU_R}{\mathrm{d}t} = g(t) \tag{13}$$

where $g(t)$ is a Gaussian process with a mean of zero. Eq. 13 means that the change of CPU_R in the next time step is proportional to a Gaussian random variable. Due to the integral in Eq. 13, we have

$$CPU_R = CPU_0 + \int_0^t g(\tau)\mathrm{d}\tau \tag{14}$$

where we denote the initial value of CPU_R as CPU_0. In addition, it is well-known that the integral of a Gaussian process (i.e., $\int_0^t g(\tau)\mathrm{d}t\tau$) is also a Gaussian process. Let t_1 and t_2 denote the start and finish time of the cheating detection process, respectively. Then, the average CPU over the cheating detection time duration is

$$\frac{T * CPU_L}{t_2 - t_1} = \frac{1}{t_2 - t_1}\int_{t_1}^{t_2} CPU_R \mathrm{d}t = CPU_0 + \frac{1}{t_2 - t_1}\int_{t_1}^{t_2}\int_0^t g(\tau)\mathrm{d}\tau\mathrm{d}t \tag{15}$$

where $T * CPU_L$ is the total number of instructions in the PCT, and $t_2 - t_1 = t_a$ is the actual task execution time. In Eq. 15, we no longer consider the system noise as previously mentioned ($\sigma = 0$ and $t_a = \mu$). Meanwhile, t_t can be calculated as $\frac{T*CPU_L}{CPU_C^L}$. Then, comparing t_a and t_t can be converted to compare $CPU_0 + \frac{1}{t_2-t_1}\int_{t_1}^{t_2}\int_0^t g(\tau)\mathrm{d}\tau\mathrm{d}t$ and CPU_C^L. If the former one is smaller, the cloud is judged to be cheating. Note that CPU_0 and CPU_C^L are fixed values, while $\frac{1}{t_2-t_1}\int_{t_1}^{t_2}\int_0^t g(\tau)\mathrm{d}\tau\mathrm{d}t$ is a Gaussian random variable. Now, the cheating detection probability can be calculated as the probability that Gaussian random variable ($\frac{1}{t_2-t_1}\int_{t_1}^{t_2}\int_0^t g(\tau)\mathrm{d}\tau\mathrm{d}t$ with mean value zero) is smaller than a threshold ($CPU_C^L - CPU_0$). Clearly, a larger initial value (larger CPU_0) will bring a lower detection probability.

5.2. A General Model

In the previous subsection, we have discussed a Gaussian process to model CPU_R. However, this model may not be general enough [18], since it does not consider the control mechanism of the cloud system [19]. The system control mechanism will adjust the CPU resource allocated to the user, e.g., more CPU resources are allocated when CPU_R is much smaller than CPU_C (the committed CPU). Therefore, we modify Eq. 13 to be

$$\frac{\mathrm{d}CPU_R}{\mathrm{d}t} = f(CPU_R - CPU_C) + g(t) \tag{16}$$

where $f(\cdot)$ is a monotonic decreasing function with $f(0) = 0$. Here, $f(\cdot)$ represents the system control mechanism that gives feedback to the CPU change. When $CPU_R > CPU_C$, the control mechanism turns down the CPU in the following time to reduce unnecessary resource consumption for the user; when $CPU_R < CPU_C$, the control mechanism turns up the CPU in the following time to guarantee the user's payment. The detailed functional form of $f(\cdot)$ depends on the cloud system design, which varies on different clouds.

Now, Eq. 16 posts a higher requirement to obtain statistic characteristics of CPU_R. One method for solving this problem is to use both the discrete state of CPU_R and the discrete time steps, instead of continuous values, as shown in Fig. 6. The state transfer probability (from the current CPU state to a CPU state of the next time step) can be determined by Eq. 16. At this time, this problem turns to be a Markov chain [20], the transition matrix of which can be constructed by the state transfer probabilities. Then, the stationary distribution of CPU_R can be solved as the eigenvectors of the transition matrix. We do not further discuss the properties of the Markov chain, since it is out of the scope of this paper. Here, we only present a general model to solve Eq. 16, while we intend to look into the real cloud systems that cannot be fully described by theories. In the next section, real system evaluations are shown to check the effectiveness of the proposed cheating detection scheme.

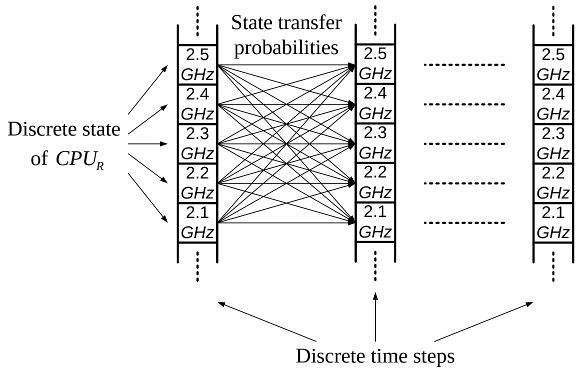


Figure 6. Discrete method to solve Eq. 16.

6. EVALUATION

In this section, the evaluation tests are conducted to check the feasibility and accuracy of the proposed cheating detection method. First, the evaluation system setup is introduced. Then the memory-intensive test is conducted to check whether or not the cheating detection process occupies lots of memory. Finally, the evaluation results are shown to verify the competitive performance of the proposed scheme.

6.1. System Setup

In this paper, we have three evaluation environments, as to show the proposed cheating detection scheme from different perspectives. The first evaluation environment is based on Oracle VM VirtualBox, version 4.1.22. The local PC is a laptop with CPU frequency $2.5GHz$ ($CPU_L = 2.5GHz$). The cloud is simulated by a VM of the VirtualBox, which is running on another laptop with 4 core CPU of $2.5GHz$ ($CPU_C = 2.5GHz$ for convenience). The CPU frequency of the VM can be adjusted through settings in the VirtualBox, by limiting the percentage of time that the virtual CPU is allowed to use of the real CPU, from 40% to 100%. Thus, CPU_R can be set from $1.0GHz$ to $2.5GHz$. The program is written in C++ with the use of the GNU multiple precision arithmetic library (gmp library version 5.1.0). The background programs are made by alternating the endless addition loops and the thread sleep function. The detection program is executed in the OS of Ubuntu, version 12.04. It is reasonable to test our algorithms on the VirtualBox software [21], since it presents the virtualization technology, which is operating in the cloud system. However, there might be some differences between the VirtualBox-based VMs and the real cloud VMs. During the test, the network service is not closed. The OS has some burst processes, for example, checking for updates through the network. In addition, experiments shown in Figs. 3 and 4 are conducted in this environment.

The other two environments are real cloud systems. One is the Temple Elastic HPC Cloud [22] (also called TCloud). TCloud is currently based on the eucalyptus open-source cloud toolkit and is composed of two main components: VMs and elastic block storage. The current TCloud configuration is hosted on a 12 (specs) R614 cloud servers for a total of 96 conventional CPU cores. Each VM is interconnected with a redundant 4-way 10Gb Ethernet and a redundant 2-way infiniband (specs). Two configurations of VMs are used in our test: single core CPU of $2.8GHz$ and dual core CPU of $2.8GHz$. The other cloud system is Temple Cluster (also called TCluster), which is constructed by 32 Dell PowerEdge R210 II Servers. Each server has a dual core Intel Celeron G530 processor at 2.4GHz. Each server also has $4GB$ RAM. They all have 500 GB SATA hard drives. For networking, they each have 4 Gigabit ports on an expansion card with 2 additional service Gigabit ports. Currently, 5 Cisco Small Business 300 Series Managed Switches are used in this system. These switches have 8 Gigabit ports for servers and 2 Gigabit ports to connect them to other switches. All of this is housed inside of 3 42U server racks made by Rack Solutions. The TCluster system is newly constructed and it only provides VMs that have a dual core CPU of $2.8GHz$. In addition, the OS and cheating detection program tested inside the VMs of these two cloud systems are the same as those of the VirtualBox system.

6.2. Memory-Intensive Test on VirtualBox Environment

Before further evaluations, it is necessary to test whether the proposed method is memory-intensive or not [23]. Here, memory-intensive means that the cheating detection process frequently occupies RAM, and insufficient free RAM would lead to aborted detection. This test is conducted in the VirtualBox environment (the first evaluation environment), in which the memory configuration of a VM can be modified. The cheating detection program is tested in three different VMs which have memories of $512MB$, $640MB$, and $2GB$, respectively. Meanwhile, we have $T = 60s$, $x\% = 0\%$, $CPU_R = CPU_C = CPU_L = 2.5GHz$. A memory of $512MB$ is not enough for supporting the OS, leading to slow reactions of the programs, since they need to use the memory in turn. A memory of $640MB$ is enough for the OS, while less than $128MB$ memory is free. A memory of $2GB$ is enough for all programs.

Table 2. Memory-Intensive Test Results

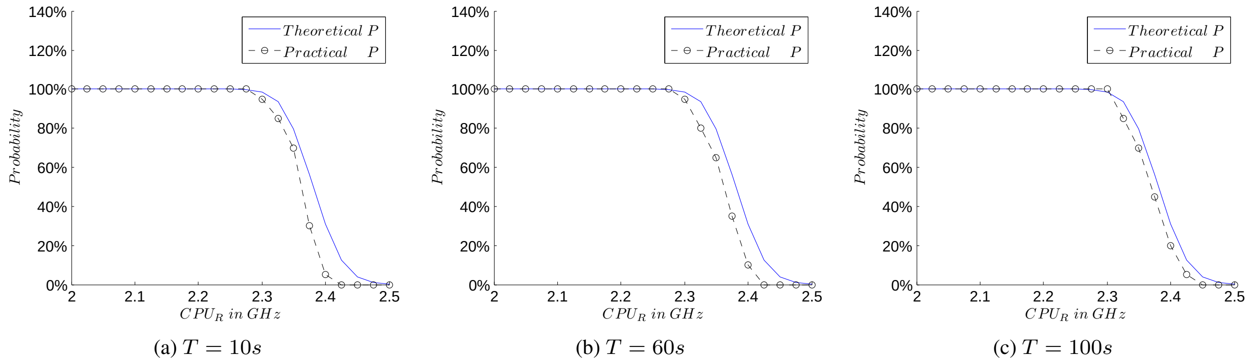
$Memory$	t_t	μ of t_a	σ of t_a
$512MB$	60	62.4	8.27
$640MB$	60	60.5	1.29
$2GB$	60	60.2	1.20

The test results are shown in Table 2, in which μ and σ are calculated by MLE through 10 samples. It can be seen that, even though memory is very limited ($512MB$), the cheating detection process works well. The t_a does not vary significantly away from t_t, though σ is abnormal. When there is only a little free memory ($640MB$), the cheating detection process works as fine as when there is enough free memory ($2GB$). Therefore, the proposed cheating detection process is not memory-intensive. Even a little free memory can ensure its normal operation.

6.3. Cheating Detection Probability on VirtualBox Environment

In this subsection, we focus on the cheating detection probability of the proposed method, based on the VirtualBox environment (the first evaluation environment). The cloud cheating is simulated by the CPU frequency modification of the VirtualBox-based VM. For example, if we want to simulate a dishonest cloud with the real CPU frequency $CPU_R = 1.0GHz$, we set the CPU frequency of the VM on the VirtualBox to be $1.0GHz$. The CPU frequency of the VirtualBox-based VM is modified through limiting the percentage of time that the virtual CPU is allowed to use of the real CPU, from 40% to 100%. Therefore, we use $CPU_C = CPU_L = 2.5GHz$ in this test, and modify the CPU frequency of the VirtualBox-based VM from $1.0GHz$ to $2.5GHz$, as to simulate cloud cheating (CPU_R varies from $1.0GHz$ to $2.5GHz$). In addition, we set $x\% = 0\%$ (the simulated cloud is idle).

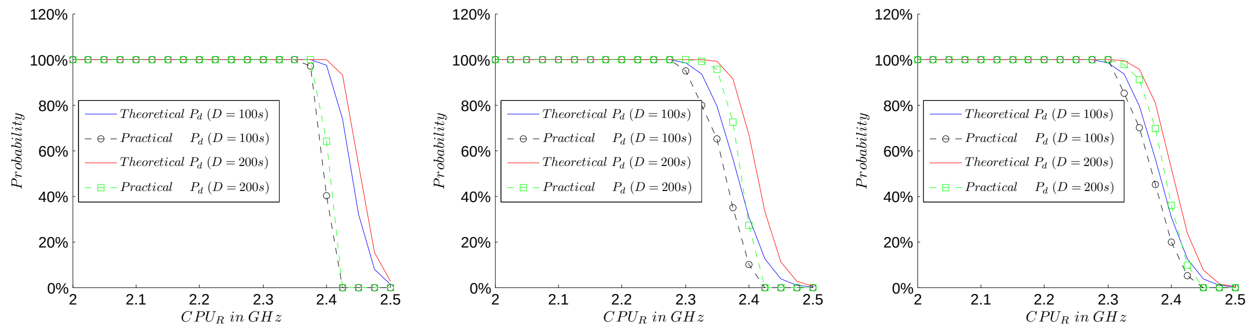
The theoretical (derived in Eq. 11) and practical cheating detection probabilities of a single-round cheating detection process have been shown in Fig. 7. A higher detection probability means that the cheating is more likely to be detected (the higher the better). It can be seen that, a single-round cheating detection process has 100% probability of discovering the cheating if $CPU_R < 90\% * CPU_C = 2.25GHz$, and it is able to detect cheating when $CPU_R < 2.4GHz$. Since $CPU_C = 2.5GHz$, it can be concluded that the proposed method works well. Note that a larger PCT (i.e., a larger T) brings a slightly better cheating detection probability, which goes against our analysis in Eq. 11. This is due to the existence of interferences that are not considered in our model. Larger PCTs should have better performances, since they are more interference-resistant.



(a) $T = 10s$ (b) $T = 60s$ (c) $T = 100s$

Figure 7. The theoretical and practical cheating detection probability of a single-round cheating detection process.

The theoretical (derived in Eq. 12) and practical cheating detection probabilities of multi-round cheating detection processes have been shown in Fig. 8. A higher detection probability also means that the cheating is more likely to be detected (the higher the better). Compared to the single-round cheating detection process in Fig. 7, the whole cheating detection probability is improved by using a larger detection time budget D. When $D = 200s$, the case $CPU_R < 2.3GHz$ can be detected for sure. As presented in Position 1, both actually and theoretically, it has a higher probability of detecting cloud cheating using a small-scale and short-length cheating detecting process many times, as opposed to a few uses of large-scale and long-length processes. Another point is that, through enlarging the total time budget of the cheating detection process (parameter D), P_d can be improved. However, this improvement is necessary, since our scheme has achieved a good detection probability.



(a) $T = 10s$ (b) $T = 60s$ (c) $T = 100s$

Figure 8. The theoretical and practical cheating detection probability of a multi-round cheating detection processes.

6.4. Real System Test on TCloud and TCluster

In this subsection, we study the feasibility of the proposed method in real cloud systems (TCloud and TCluster). As previously mentioned, TCloud can provide two types of VMs (single core CPU of $2.8GHz$ and dual core CPU of $2.8GHz$), while TCluster can only provide one type (dual core CPU of $2.4GHz$). The CPU frequencies of VMs in these two clouds cannot be modified, since the cloud environments do not provide the programming interface. Therefore, we focus on the gap between the actual task execution time (t_a) and theoretical task execution time (t_t) of some given PCTs (note that the PCT includes $T * CPU_L$ instructions). The CPU frequency of the local PC is still set to be $CPU_L = 2.5GHz$, while the parameter T is respectively set to be $10s$, $60s$, and $100s$. For each setting, five different CPU_C are used to calculate t_t for comparisons with t_a. Note that, since the cloud is idle ($x\% = 0\%$), we have $t_t = T * CPU_L / CPU_C$. Meanwhile, for each setting, we run the cheating detection program 10 times to get the statistics of the actual task execution time (i.e., 10 samples of t_a).

The evaluation results are shown in Table 3. The t_t, which correspond to $CPU_R = CPU_C$, are in bold font. For the TCloud platform, the number of CPU cores (single core or dual core) do not bring a significant difference on the actual task execution times of the PCT, since the PCT cannot be executed in parallel. If TCloud declares its CPU to be $3.0GHz$ ($CPU_R = 2.8GHz$), this cheating can be easily detected since there is a huge time gap (more than 10%) between t_a and t_t. However, there is still a considerable performance gap when TCloud honestly declares its CPU to be $2.8GHz$. The *overhead* (e.g., burst programs from the OS) is not ignorable. The t_a of a VM with $2.8GHz$ CPU is closed to the t_t of a VM with $2.6GHz$ CPU. Therefore, a CPU resource of $0.2GHz$ may be considered to be the overhead. As for the TCluster, the actual task execution time is very closed to its theoretical one. Remarkable gaps exist between the t_a and the t_t (for both $CPU_C = 2.2GHz$ and $CPU_C = 2.6GHz$). Therefore, cheating can be easily detected in TCluster. The evaluation results show the feasibility and validity of the proposed scheme.

7. CONCLUSION

In this paper, we propose a method to detect whether the cloud is providing the correct amounts of CPU resources that the client has paid for. Our solution is based on task execution time comparison: a PCT is constructed for the cloud to execute; since the amount of calculation of the task is known, the gap between the actual and theoretical task execution time can be used to judge whether the cloud is cheating or not. The PCT is based on time-lock puzzles, so that the task cannot be simplified by the cloud to reduce the amount of calculations. Further analysis shows that, in our model, it has a higher probability of detecting cloud cheating using small-scale and short-length cheating detection processes more frequently than using large-scale and long-length cheating detection processes less frequently. We also build a primary model for the fluctuation of the cloud CPU with respect to time. Our evaluation shows that the cheating detection scheme is not memory-intensive, and is applicable to real-world cloud cheating detection.

Table 3. Evaluations in TCloud and TCluster.

Configurations		Actual task execution time (t_a) (statistics of 10 samples)			Theoretical task execution time (t_t) (each sub-column corresponds to a different CPU_C)				
		min	mean	max	$2.2GHz$	$2.4GHz$	$2.6GHz$	$2.8GHz$	$3.0GHz$
TCloud single core @$2.8GHz$	T=10s	9.64s	9.67s	9.78s	11.36s	10.42s	9.61s	**8.93s**	8.33s
	T=60s	57.90s	57.97s	58.06s	68.18s	62.50s	57.69s	**53.57s**	50.00s
	T=100s	96.43s	96.53s	96.66s	113.64s	104.17s	96.15s	**89.29s**	83.33s
TCloud dual core @$2.8GHz$	T=10s	9.64s	9.66s	9.68s	11.36s	10.42s	9.61s	**8.93s**	8.33s
	T=60s	57.87s	58.00s	58.65s	68.18s	62.50s	57.69s	**53.57s**	50.00s
	T=100s	96.46s	96.66s	97.96s	113.64s	104.17s	96.15s	**89.29s**	83.33s
TCluster dual core @$2.4GHz$	T=10s	10.16s	10.23s	10.68s	11.36s	**10.42s**	9.61s	8.93s	8.33s
	T=60s	61.00s	61.09s	61.48s	68.18s	**62.50s**	57.69s	53.57s	50.00s
	T=100s	101.55s	101.71s	101.81s	113.64s	**104.17s**	96.15s	89.29s	83.33s

REFERENCES

[1] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Proceedings of GCE Workshop 2008*, pp. 1–10.

[2] P. Mell and T. Grance, "The NIST Definition of Cloud Computing (Draft)," *NIST*, vol. 800, no. 145, p. 7, 2011.

[3] P. Hofmann and D. Woods, "Cloud computing: The limits of public clouds for business applications," *IEEE Internet Computing*, vol. 14, no. 6, pp. 90–93, 2010.

[4] R. Buyya, C. S. Yeo, and S. Venugopal, "Market-oriented cloud computing: Vision, hype, and reality for delivering it services as computing utilities," in *Proceedings of HPCC 2008*, pp. 5–13.

[5] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Proceedings of CLOUDCOM 2010*, pp. 409–416.

[6] K. L. Kroeker, "The evolution of virtualization," *Communications of the ACM*, vol. 52, no. 3, pp. 18–20, 2009.

[7] K. Li, J. Wu, and A. Blaisse, "Elasticity-aware virtual machine placement for cloud datacenters," in *Proceedings of CloudNet 2013*, pp. 99–107.

[8] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of Internet Services and Applications*, vol. 1, no. 1, pp. 7–18, 2010.

[9] K. Li, H. Zheng, and J. Wu, "Migration-based virtual machine placement in cloud systems," in *Proceedings of CloudNet 2013*, pp. 83–90.

[10] Y. Zhao and J. Wu, "Dache: A data aware caching for big-data applications using the mapreduce framework," in *Proceedings of INFOCOM 2013*, pp. 35–39.

[11] B. Farley, A. Juels, V. Varadarajan, T. Ristenpart, K. D. Bowers, and M. M. Swift, "More for your money: exploiting performance heterogeneity in public clouds," in *Proceedings of SoCC 2012*, pp. 1–14.

[12] F. Zhu, H. Li, and J. Lu, "A service level agreement framework of cloud computing based on the cloud bank model," in *Proceedings of CSAE 2012*, pp. 255–259.

[13] H. Zheng, K. Li, C. C. Tan, and J. Wu, "User-based cpu verification scheme for public cloud computing," in *Proceedings of CLOUD 2013*, pp. 732–739.

[14] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo random number generator," *SIAM Journal on Computing*, vol. 15, no. 2, pp. 364–383, 1986.

[15] D. Nowak, "Information Security and Cryptology," P. J. Lee and J. H. Cheon, Eds., 2009, ch. On Formal Verification of Arithmetic-Based Cryptographic Primitives.

[16] R. L. Rivest, A. Shamir, and D. A. Wagner, "Time-lock puzzles and timed-release crypto," Tech. Rep., 1996.

[17] M. Mahmoody, T. Moran, and S. Vadhan, "Time-lock puzzles in the random oracle model," in *Proceedings of CRYPTO 2011*, pp. 39–50.

[18] Y. Zhao and J. Wu, "Socially-aware publish/subscribe system for human networks," in *Proceedings of WCNC 2010*, pp. 1–6.

[19] H. C. Lim, S. Babu, J. S. Chase, and S. S. Parekh, "Automated control in cloud computing: Challenges and opportunities," in *Proceedings of ACDC 2009*, pp. 13–18.

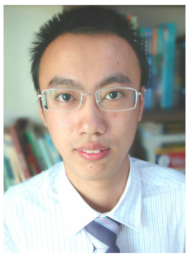
[20] S. Brooks, A. Gelman, G. Jones, and X.-L. Meng, *Handbook of Markov Chain Monte Carlo*, 2011.

[21] J. Watson, "Virtualbox: bits and bytes masquerading as machines," *Linux Journal*, vol. 2008, no. 166, 2008.

[22] *https://sites.google.com/a/temple.edu/tcloud/home*.

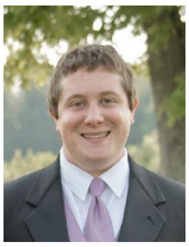
[23] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proceedings of CGO 2010*, pp. 101–110.

BIOGRAPHY OF AUTHORS



Huanyang Zheng received his B.Eng. in Telecommunication Engineering from Beijing University of Posts and Telecommunications, Beijing, China, in 2012. He is currently a Ph.D. student in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, US. His current research focuses on the opportunistic networks, social networks, and cloud systems. Further info on his homepage: http://astro.temple.edu/%7Etue86241/

Kangkang Li is a graduate student in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, US. He received his B.Eng. in Communication Engineering from Nanjing University of Posts and Telecommunications, Nanjing, China. His research interests include data center networks, resource management in distributed and cloud computing, and wireless networks.



Adam Blaisse received his B.S. in Mathematics and Computer Science from Temple University, Philadelphia, PA, US, in 2013. He is currently a M.S. Student in the Department of Computer and Information Sciences, Temple University, Philadelphia, PA, US. His current research focuses on mobile computing, and cloud systems. Further info on his homepage: http://astro.temple.edu/%7Etuc47904/

Chiu C. Tan is an assistant professor in the Department of Computer and Information Sciences at Temple University. He received his PhD from the College of William and Mary in 2010. His research is in the area of cyber security, and his current research interests are in cloud computing security, smarthealth systems, and wireless network security (mainly 802.11, RFID, and sensor networks). He is also the director for the NSF/DoD REU Site program at Temple University. Further info on his homepage: http://www.cis.temple.edu/%7Ecctan/

Jie Wu is the chair and a Laura H. Carnell Professor in the Department of Computer and Information Sciences at Temple University. Prior to joining Temple University, he was a program director at the National Science Foundation and Distinguished Professor at Florida Atlantic University. His current research interests include mobile computing and wireless networks, routing protocols, cloud and green computing, network trust and security, and social network applications. Dr. Wu regularly publishes in scholarly journals, conference proceedings, and books. He serves on several editorial boards, including IEEE Transactions on Computers, IEEE Transactions on Service Computing, and Journal of Parallel and Distributed Computing. Dr. Wu was general co-chair or chair for IEEE MASS 2006 and IEEE IPDPS 2008 and program co-chair for IEEE INFOCOM 2011. Currently, he is serving as general chair for IEEE ICDCS 2013 and ACM MobiHoc 2014, and program chair for CCF CNCC 2013. He was an IEEE Computer Society Distinguished Visitor, ACM Distinguished Speaker, and chair for the IEEE Technical Committee on Distributed Processing (TCDP). Dr. Wu is a CCF Distinguished Speaker and a Fellow of the IEEE. He is the recipient of the 2011 China Computer Federation (CCF) Overseas Outstanding Achievement Award. Further info on his homepage: http://www.cis.temple.edu/%7Ejiewu/index.html