User-based CPU Verification Scheme for Public Cloud Computing

Huanyang Zheng, Kangkang Li, Chiu C. Tan and Jie Wu
Department of Computer and Information Sciences, Temple University, USA
Email: {huangyang.zheng,kang.kang.li,cctan,jiewu}@temple.edu

***Abstract*—In this paper, a user-based CPU verification scheme is proposed for cloud cheating detection. In this scheme, a predefined computational task is constructed for the cloud to execute in our cheating detection process. 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 the predefined computational task is guaranteed to be 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. Finally, the feasibility and validity of the proposed scheme is shown in the evaluations.**

***Index Terms*—Cloud computing, cheating, CPU verification, time-lock puzzle.**

I. Introduction

Scalability and metering are two popular features among users of commercial cloud computing services, because they allow users to reduce their operating costs [1–3]. 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 provide this type of service by sharing its hardware between 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 [4–6]. 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 migrate that VM to a separated hardware when more computing resources are required. Improving this migration process is an active area of research [7, 8].

The cloud provider should provide the amount of computing resources that a user has paid for. However, since the cloud provider is both the entity providing the resources as well as 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 15 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, due to errors in the VM migration code or algorithm. In this paper, we consider the problem of allowing users to verify that they actually receive the CPU resources they have purchased.

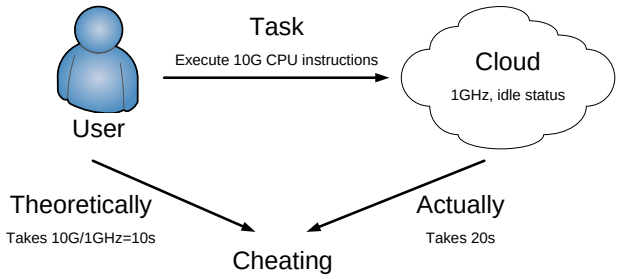


Fig. 1. Example of the cheating detection

Our proposed solution lets the user check the amount of CPU resources by measuring the time it takes for the cloud to complete a predefined computing task (PCT). Fig. 1 illustrates the testing 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, $20s$, for example. 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. This process is called the cheating detection process. This solution requires addressing the following challenges.

- How can we guarantee that the PCT is indeed executed by the cloud? As shown in Fig. 1, the cloud can predict the execution time of 10^{10} CPU instructions, and then report completion after $10s$. Moreover, the cloud may simplify the PCT to speed up the task execution.
- 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, when there is some background programs assigned by the user?
- Obviously, larger PCT resists interference better and thus has higher performance on detection, but in turn, costs more time. If the total detection time is limited, should we use small-scale PCT more times, or large-scale PCT fewer times?

The remainder of the paper is organized as follows: in Section II, we present the cheating problem mathematically, and show the monitoring architecture. Then the cheating detection process is described detailedly in Section III. Section IV shows the analysis for the error control parameter. Evaluation is shown in Section V. Finally, we conclude the paper in Section VI.

II. End User CPU Monitoring

In this section, we firstly 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.

A. Problem Formulation

We assume the committed CPU frequency of the cloud brought by the user is CPU_C, and the real CPU frequency of the cloud is CPU_R. Then the cloud cheating is defined as

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

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 I total CPU instructions to complete, and t_t is the theoretical time of executing the same task. Ideally, t_a=I/CPU_R and t_t=I/CPU_C, if the cloud CPU is idle. δ is the error control parameter (similar to ε in Eq. 1). Considering that Eq. 2 is more closed to user experience than Eq. 1, our detection algorithm uses Eq. 2 as the cheating definition. Naturally, the cheating detection is defined as to detect the abnormal task execution time. Note that, it is meaningless for the cloud provider to slightly cheat the end user, and thus generally $t_a - t_t \gg \delta$ if cheating happens.

Since the user bought the cloud for some computational tasks rather than detections, we assume resources for detection are limited: the whole running time of the detection program should be less than D seconds (denoted as the detection budget). The cloud is running some background programs, the total CPU usage of which is stably $x\%$. In addition, we assume cloud is as smart as humans in doing any possible anti-detection actions. For example, the cloud would present 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 the detection assist. Small interferences such as network transmission delay are neglected.

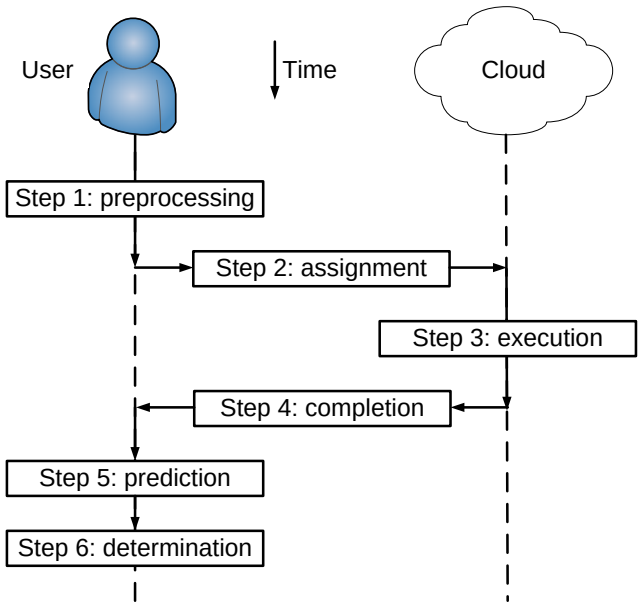


Fig. 2. The cheating detection process

B. Monitoring Architecture

Since the cloud is able to do any anti-detection actions, the cheating detection has to depend on reliable parameters provided by the user. Among all these parameters, time is the most convenient one, which is also convenient for the detection 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 cheating.

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 require 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 successfully in the cloud.

TABLE I
PARAMETERS INVOLVED IN THE ALGORITHMS

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.
p, q	Two large random prime numbers.
n	Calculated by $n = pq$.
$\phi(n)$	Calculated by $\phi(n) = (p-1)(q-1)$.
b	A large random number which is relatively prime to n, $gcd(b, n) = 1$.
S	The number of computing $b = b^2 \, mod \, n$ per second in local PC using 100% CPU.
T	A time parameter in seconds, which is used to control the amount of calculation of the PCT (totally $T{*}CPU_L$ CPU instructions).
M	Calculated by $M = TS$. Computing $b^2 \, mod \, n$ for M times takes $T{*}CPU_L$ CPU instructions, which cost T seconds for execution in local PC using 100% CPU.
a_L	Calculated by $a_L = b^{2^M} \, mod \, n$ in the local PC.
a_C	Calculated by $a_C = b^{2^M} \, mod \, n$ 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_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.
δ	Error control parameter, used for determining cheating.
I	Cheating definition parameter. If the actual time of executing I CPU instructions $>$ the theoretical time $+$ the error control parameter, the cloud is cheating.
D	The 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.

$t_a > t_t + \delta$ determines that the cloud is cheating. In the next section, more details about the detection process are shown, including the construction of the PCT, the calculation of t_t, and the cheating determination.

III. 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 cheating determination and error control. Finally, the algorithm on the whole is shown. All parameters involved are shown in Table I.

A. Predefined Computational Task

The amount of calculation of the PCT is $T{*}CPU_L$ CPU instructions in total (note $T{*}CPU_L{>}I$ 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 (a_C as the answer from the cloud, and a_L as the answer from the local PC). If $a_C{=}a_L$, the PCT is executed successfully 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 $a{=}a{+}1$ $1,000$ times (uses $a{=}0$ for initialization and returns $a{=}1,000$), the cloud could replace this task with $a{=}a{+}1,000$ once, which simplifies the task. 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 [9–11].

Algorithm 1 Constructing The PCT in The Local PC

Input: Parameter T;
Output: Parameter b, M, n for the PCT in the cloud;
Parameter a_L for the cheating determination;

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: $\quad b = b^2 \, mod \, n$;
8: $\quad S = S + 1$;
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: Parameter $x\%$ for calculating t_t;
Parameter a_C for the cheating determination;

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

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 [12–14]. 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 to solve $b^{2^M} \, mod \, n$ is unnecessary. However, if p and q are known, $a = b^{2^M} \, mod \, n$ can be efficiently calculated by

$$\begin{aligned} \phi(n) &= (p-1)(q-1) \\ e &= 2^M mod \; \phi(n) \\ a &= b^e mod \; n \end{aligned} \tag{3}$$

Therefore, calculating $b^{2^M} \, mod \, n$ is employed as the PCT, which takes $T{*}CPU_L$ CPU instructions if p and q are unknown. The process of 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.

B. 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 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,

$$y\% = \frac{x\%}{x\% + 100\%} \tag{4}$$

$$z\% = \frac{100\%}{x\% + 100\%} \tag{5}$$

Since the amount of calculation of the PCT is $T{*}CPU_L$ CPU instructions in total, if $x\% = 0\%$, and the user obtained CPU_C, executing $T{*}CPU_L$ CPU instructions in the cloud should take $T{*}CPU_L/CPU_C$ seconds. Thus,

$$t_t = T * \frac{CPU_L}{CPU_C} \tag{6}$$

Taking the influence of the background programs into account ($x\% \neq 0\%$), we have

$$\begin{aligned} t_t &= T * \frac{CPU_L}{CPU_C * z\%} \\ &= T * (1 + x\%) * \frac{CPU_L}{CPU_C} \end{aligned} \tag{7}$$

In Eq. 7, t_t is obtained, which is step 5 in Fig. 2. In the next subsection, how to determine cheating is shown as step 6 in Fig. 2.

C. 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 successfully, 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.

Algorithm 3 Cheating Determination

Input: Parameter a_C, a_L, t_a, t_t, and δ;
Output: Whether the cloud is cheating or not;

```
if a_C ≠ a_L then
    Return cheating;
else
    if t_a > t_t + δ then
        Return cheating;
    else
        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;

```
Calculate T_max = D/(1 + x%) * CPU_C/CPU_L;
for count = 1 to ⌊T_max/T⌋ do
    Use Algorithm 1 to construct the PCT in the local PC;
    Start timer;
    Use Algorithm 2 to execute the PCT in the cloud;
    Stop timer to get the t_a;
    Calculate t_t = T * (1 + x%) * CPU_L/CPU_C;
    if Algorithm 3 judges the cloud to be cheating then
        Return cheating;
Return no-cheat;
```

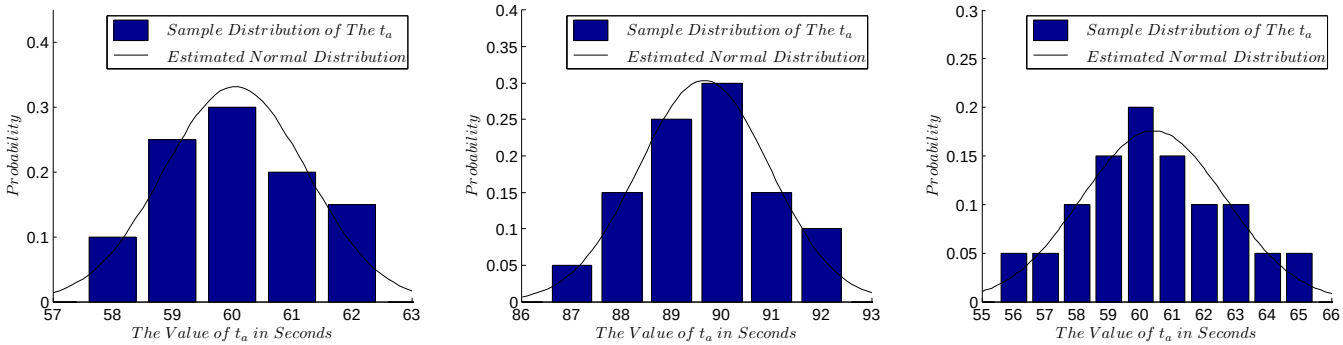
However, how to determine the error control parameter δ remains to be a major challenge. Ideally, if there is no interference, δ could be set to 0, and t_t should be strictly equal to t_a. But the interferences indeed exist. Since t_t is calculated by T, $x\%$, CPU_L and CPU_C (where δ is the error control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

D. 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 stated in Section II subsection A, we have at most D seconds for a cheating detection processs, which requires $t_t < D$. According to Eq. 7,

$$T < \frac{1}{1 + x\%} * \frac{CPU_C}{CPU_L} * D = T_{max} \tag{8}$$

Note that if $T \ll T_{max}$ ($t_t \ll D$), then the remaining time is wasted. A better method is to recursively run the cheating detection process until all D seconds are used up (totally $\lfloor T_{max}/T \rfloor$ times). 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 run the cheating detection process in Fig. 2 10 times: only if the cheating detection processes return no-cheat all 10 times, the cloud is judged to be no-cheat. In summary, the whole algorithm is presented in Algorithm 4. In



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

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

addition, the lower bound of the T is given by the definition of cheating in Eq. 2:

$$T > \frac{I}{CPU_L} = T_{min} \tag{9}$$

Obviously, the value of T has 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 less 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.

IV. 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.

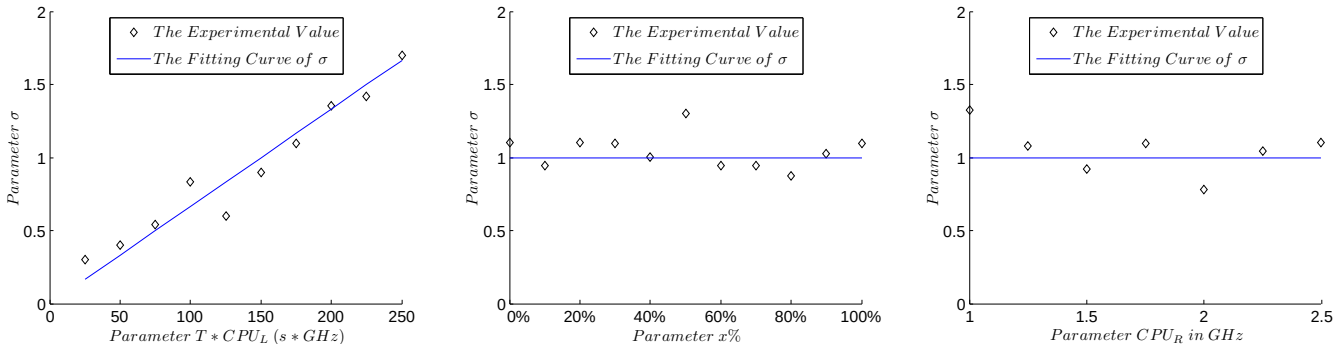
A. Model

Due to the interference, the t_a randomly varies, which is modeled to submit normal distribution (μ, σ). μ is the expectation of t_a, and σ is the standard deviation of t_a. 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). It is reasonable to assume that t_a submits to normal distribution, as shown in Fig. 3. In the experiments shown in Fig. 3, there are $T * CPU_L = 1.5 * 10^{11}$ CPU instructions in total assigned as the the amount of calculations of the PCT. We collected 20 sampling points of t_a in each test, the distribution of which is called, sampling distribution (for convenience, t_a is recorded in seconds). The maximum likelihood estimate (MLE) is employed to estimate the distribution of these sampling points, the distribution of which is called, estimated normal distribution. The experiments in Fig. 3 proves the feasibility of estimating t_a to submit normal distribution. In addition, the detailed test environment is described in Section V subsection A.

Then we need to estimate the value of μ and σ, if no cheating happens. Obviously, μ can be estimated to be t_t (replace CPU_C in Eq. 7 with CPU_R if the cloud is cheating), and the next step is to estimate σ. Intuitively, σ should be related to parameter M, $x\%$ and CPU_R: M is the input of Algorithm 2, which determines the extent of the PCT's calculations to be computing $b^2 \, mod \, n$ for M times; $x\%$ and CPU_R describes the computational capabilities of the cloud. Therefore, σ can be written as $\sigma(M, x\%, CPU_R)$. Note that, computing $b^2 \, mod \, n$ for M times needs $T{*}CPU_L$ CPU instructions in total, thus σ is rewritten to be $\sigma(T{*}CPU_L, x\%, CPU_R)$. 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

$$\sigma = \frac{T}{60} * \frac{CPU_L}{2.5} = \frac{T * CPU_L}{150} \tag{10}$$

where the unit of T is s, and the unit of CPU_L is GHz. The reason why σ is uncorrelated to parameters $x\%$ and CPU_R is that, these two parameters lead to very small fluctuations of t_a, which can be relatively neglected. The amount of calculations by the PCT is the main point. The fluctuation of t_a brought by $T{*}CPU_L$ derives from the inaccurate parameter S in Algorithm 2. Fortunately, the parameter S could be pre-measured by the cheating detection software developer to further improve the accuracy. Note that, the coefficient $1/150$ in Eq. 10 is not changeless among different machines. However, the tendency keeps that σ is linearly proportional to $T{*}CPU_L$, and is uncorrelated to $x\%$ and CPU_R. Based on Eq. 10, we further discuss how to set parameter T in the next subsection.



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

Fig. 4. Analysis of parameter σ

B. Analysis

In the former subsection, we modeled t_a to submit normal distribution (μ, σ) due to the interferences. μ equals t_t if no cheating happens, and σ is fitted as Eq. 4. In a single cheating detection process, we employ condition $t_a > t_t + 3\sigma$ to judge cloud cheating. If the cloud is cheating, $CPU_R \neq CPU_C$, the probability to successfully detect cheating is

$$P = \int_{t_t+3\sigma}^{\infty} \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{(t-\mu)^2}{2\sigma^2}} dt = \frac{1}{2} erfc(\frac{t_t + 3\sigma - \mu}{\sqrt{2}\sigma}) \quad (11)$$

where

$$\mu = T * (1 + x\%) * \frac{CPU_L}{CPU_R} \quad and \quad \sigma = \frac{T * CPU_L}{150} \quad (12)$$

Simplifying Eq. 11, we have

$$P = \frac{1}{2} erfc[\frac{3+150(1+x\%)(1/CPU_C - 1/CPU_R)}{\sqrt{2}}] \quad (13)$$

The theoretical successful detection rate of a single cheating detection process has been shown in Eq. 13. If $x\% = 0\%$ and $CPU_C = 2.5GHz$, the theoretical successful cheating detection probability would be $P = 100\%$ 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 $P = 0.1\%$. Note that P is uncorrelated to parameter T and CPU_L. Based on Eq. 13, the total probability of successful cheating detection using time budget D is

$$P_d = 1 - (1 - P)^n$$
$$n = \lfloor \frac{T_{max}}{T} \rfloor = \lfloor \frac{1}{1 + x\%} * \frac{CPU_C}{CPU_L} * \frac{D}{T} \rfloor \quad (14)$$

Note that P_d is increased with decreasing T, and thus 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 is no other interferences (for example, network transmission delay), which would lead to errors when T is too small. Another point is that T has its lower bound given by Eq. 9. So, we suggest to set $T = T_{min}$. In the consideration of that, the background programs generally occupy CPU erratically; we suggest to run the cheating detection process when the cloud is idle ($x = 0\%$).

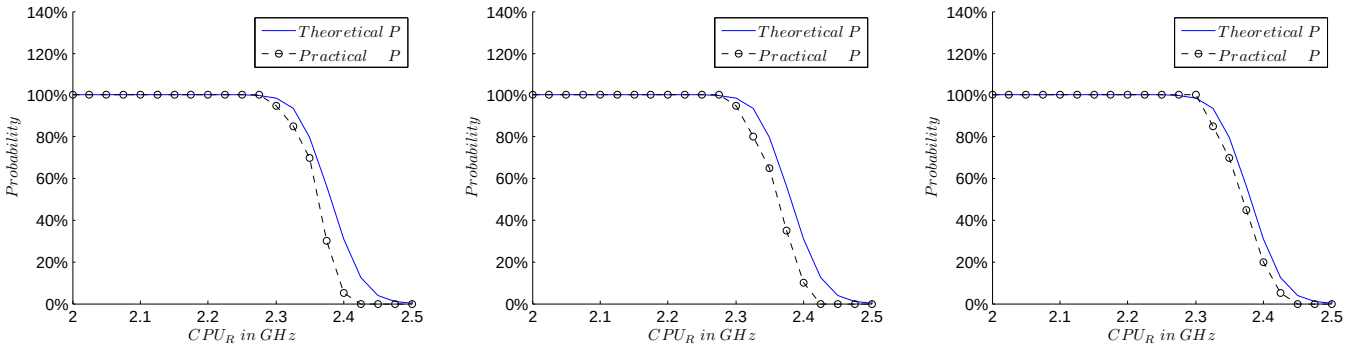
V. Evaluation

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.

A. System Setup

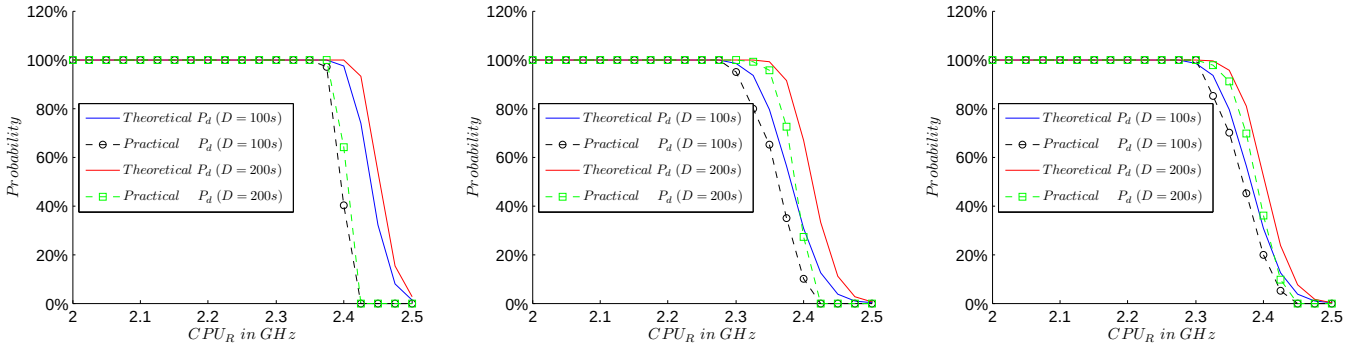
Our evaluation environment is based on Oracle VM VirtualBox, version 4.1.22. The local PC is a laptop with CPU frequency $2.5GHz$ ($CPU_L = 2.5GHz$). The cloud is a virtual machine assigned by the VirtualBox, which is running on another laptop with 4 core of CPU frequency $2.5GHz$ ($CPU_C = 2.5GHz$ for convenience). The CPU frequency of the virtual machine 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 usage of the GNU multiple precision arithmetic library (gmp library version 5.1.0). The detection program is executed in the OS of Ubuntu, version 12.04.

It is reasonable to test our algorithms on the VirtualBox software [15], 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, since this is more similar to the real situation. The OS has some burst processes, for example, checking for updates through the network. In addition, the background programs are made by alternating the endless addition loops and the thread sleep function.



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

Fig. 5. The theoretical and practical successful cheating detection rate of a cheating detection process



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

Fig. 6. The theoretical and practical successful cheating detection rate of cyclic cheating detection processes with the same total detection budget D

TABLE II
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

B. Memory-Intensive Test

Before conducting further tests, it is necessary to test whether the proposed method is memory-intensive or not [16]. Here memory intensive means that the cheating detection process frequently occupies RAM, and insufficient free RAM would lead to aborted detection. The memory configuration is able to be modified in the VirtualBox. Three tests are conducted for $RAM = 512MB$, $RAM = 640MB$ and $RAM = 2GB$, with the parameter $T = 60s$, $x = 0\%$, $CPU_R = CPU_C = CPU_L = 2.5GHz$. When $RAM = 512MB$, the memory is not even enough for supporting the OS, and the programs react slowly, since they need to wait for the usage of the insufficient memory. When $RAM = 640MB$, the memory is enough for the OS, and there is very limited free memory. When $RAM = 2GB$, there are enough memory for all the programs. μ and σ are calculated by MLE through 10 samples.

The test results are shown in Table II. It can be seen that, even though memory is very limited ($RAM = 512MB$), the cheating detection process works. The t_a does not vary significantly away from t_t, though σ is abnormal. When there is only a little free memory ($RAM = 640MB$), the cheating detection process works as fine as when there is enough free memory ($RAM = 2GB$). Therefore, the proposed cheating detection process is not memory-intensive. Even a little free memory can ensure the operation of the cheating detection process.

C. Cheating Detection Rate

In this subsection, we observe the probability of the successful cheating detection of the proposed method. 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) and $T_{min} = 10s$.

The theoretical (derived in Eq. 13) and practical successful cheating detection rate of a cheating detection process has been shown in Fig. 5. 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 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 if $CPU_R < 2.4GHz$. In the consideration of $CPU_C = 2.5GHz$, it can be concluded that the proposed method works well. In addition, note that the theoretical successful cheating detection rate is not correlated to the amount of PCT's total calculations ($T*CPU_L$ CPU instructions). But in practice, a larger PCT indeed brings slightly better detection probability. This gap results from the model of σ.

The theoretical (derived in Eq. 14) and practical successful cheating detection rates of cyclic cheating detection processes, with the same total detection budget D, has been shown in Fig. 6. A higher detection probability means that the cheating is more likely to be detected in the detection budget D (the higher the better). Compared to the single cheating detection process in Fig. 5, the whole cheating detection rate is improved by using a larger detection time budget D. When $D = 200s$, the cheating of $CPU_R < 2.3GHz$ is surely detected. Here, a large amount of calculation of of PCT's total calculations ($T*CPU_L$ CPU instructions), has a negative influence on the cheating detection rate. As presented in Position 1, both actually and theoretically, it has has a higher probability of detecting cloud cheating using 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 to 100% if the corresponding $P > 0$. Theoretically, the cheating can be surely detected if $CPU_R < CPU_C$. However, in practice, slight differences between CPU_R and CPU_C leads to unsuccessful detection ($P_d = 0$). Fortunately, it is meaningless to point out slight differences between CPU_R and CPU_C, since $CPU_R \ll CPU_C$ if cheating happens.

VI. Conclusion

In this paper, a cheating detection method is proposed to detect whether the cloud is cheating or not. The method is based on task execution time comparison: a predefined computational task is constructed for the cloud to execute; since the amount of calculation of the task is known, the theoretical execution time of the task can be computed as t_t; meanwhile, the actual execution time of the task is t_a; if $t_a > t_t + \delta$, the cloud is judged to be cheating, where δ is an error control parameter. The predefined computational task is based on time-lock puzzles, so that the task cannot be simplified by the cloud to reduce the amount of calculations. Then, further analysis shows that, based on our model, it has a higher probability of detecting cloud cheating using small-scale and short-length cheating detecting processes more frequently than using large-scale and long-lenth cheating detecting processes less frequently. Finally, the evaluation shows that the cheating detection process is not memory intensive, and is applicable to real-world cloud cheating detection.

References

[1] I. Foster, Y. Zhao, I. Raicu, and S. Lu, "Cloud computing and grid computing 360-degree compared," in *Grid Computing Environments Workshop, 2008. GCE '08*, nov. 2008, pp. 1 –10.

[2] P. Mell and T. Grance, "The NIST Definition of Cloud Computing (Draft)," *National Institute of Standards and Technology*, p. 7, Jan. 2010.

[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, Nov. 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 *High Performance Computing and Communications, 2008. HPCC '08. 10th IEEE International Conference on*, sept. 2008, pp. 5 –13.

[5] N. Regola and J.-C. Ducom, "Recommendations for virtualization technologies in high performance computing," in *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, ser. CLOUDCOM '10. Washington, DC, USA: IEEE Computer Society, 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] 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.

[8] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "Snowflock: rapid virtual machine cloning for cloud computing," in *Proceedings of the 4th ACM European conference on Computer systems*, ser. EuroSys '09. New York, NY, USA: ACM, 2009, pp. 1–12.

[9] L. Blum, M. Blum, and M. Shub, "A simple unpredictable pseudo random number generator," *SIAM J. Comput.*, vol. 15, no. 2, pp. 364–383, May 1986.

[10] D. Hofheinz and E. Kiltz, "Practical chosen ciphertext secure encryption from factoring," in *Proceedings of the 28th Annual International Conference on Advances in Cryptology: the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT '09. Berlin, Heidelberg: Springer-Verlag, 2009, pp. 313–332.

[11] D. Nowak, "Information security and cryptology — icisc 2008," P. J. Lee and J. H. Cheon, Eds. Berlin, Heidelberg: Springer-Verlag, 2009, ch. On Formal Verification of Arithmetic-Based Cryptographic Primitives, pp. 368–382.

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

[13] Y. Jerschow and M. Mauve, "Offline submission with rsa time-lock puzzles," in *Computer and Information Technology (CIT), 2010 IEEE 10th International Conference on*, July 2010, pp. 1058 –1064.

[14] M. Mahmoody, T. Moran, and S. Vadhan, "Time-lock puzzles in the random oracle model," in *Proceedings of the 31st annual conference on Advances in cryptology*, ser. CRYPTO'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 39–50.

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

[16] J. E. Gottschlich, M. Vachharajani, and J. G. Siek, "An efficient software transactional memory using commit-time invalidation," in *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, ser. CGO '10. New York, NY, USA: ACM, 2010, pp. 101–110.