On Trustworthiness of CPU Usage Metering and Accounting

Mei Liu
State Key Laboratory of Information Security
Graduate University of Chinese Academy of Sciences
Beijing, China
Email:mliu@is.ac.cn

Xuhua Ding
School of Information Systems
Singapore Management University
Singapore
Email: xhding@smu.edu.sg

Abstract—**In the envisaged utility computing paradigm, a user taps a service provider's computing resources to accomplish her tasks, without deploying the needed hardware and software in her own IT infrastructure. To make the service profitable, the service provider charges the user based on the resources consumed. A commonly billed resource is CPU usage. A key factor to ensure the success of such a business model is the trustworthiness of the resource metering scheme. In this paper, we provide a systematic study on the trustworthiness of CPU usage metering. Our results show that the metering schemes in commodity operating systems should not be used in utility computing. A dishonest server can run various attacks to cheat the users. Many of the attacks are surprisingly simple and do not even require high privileges or sophisticated techniques. To demonstrate that, we experiment with several types of attacks on Linux and show their adversarial effects. We also suggest that source integrity, execution integrity and fine-grained metering are the necessary properties for a trustworthy metering scheme in utility computing.**

Keywords-**CPU time metering; attack; utility computing**

I. INTRODUCTION

Recent years have seen a strong growth of utility computing and cloud computing propelled by IT giants including IBM, HP, Microsoft, Google, Sun, Intel, EMC, Amazon, etc. They model metering and billing service similar to a traditional public utility. In such an environment, a user outsources her computation task either by uploading her own application to the server, or by running the software provided by the server. She is then billed according to the resources consumed by the tasks. Among others, a widely used metric is time [1]–[6]. For instance, Amazon's EC2 [1] charges the users based on execution hours with different platform specifications. This resembles the pricing plans used by power suppliers and the mobile service providers whom a user pays in accord with the reading on the electricity meter or based on the reported talking time. The success of these services relies on the trustworthiness of the underlying metering facility. However, it is more challenging to establish trust on a CPU time metering scheme in utility computing or cloud computing.

First, in the power supply service, the electricity meter is a special hardware device certified by a local authority. In utility computing, it is the operating system that manages all resources, including CPU. Like all other software, the operating system can be misconfigured, compromised or subverted to the adversary's advantage. Secondly, a phone bill can be verified by checking a phone set's local log. Nonetheless, it is extremely challenging to verify a bill on CPU time for a remote process (if not completely infeasible).

In this paper, we highlight that the trustworthiness of a metering scheme is one of the bedrocks for the utility computing. It is clear to observe that the utility computing server can cheat in CPU time metering, as the entire platform is under the server's control. The goal of this paper is to illustrate that the attacks are surprisingly simple and straightforward. They do not require attacks on the OS kernel and some of them even do not require a root privilege. More importantly, our study shows that all attacks are rooted in the discordance between the metering scheme and the program execution mechanism in modern operating systems. Based on our result, we argue that operating systems used in utility computing should have a dedicated and customized CPU time metering scheme.

To this end, we first identify the vulnerabilities in CPU time metering scheme and the program execution process. By taking advantage of these vulnerabilities, we describe and implement several types of attacks on CPU time metering in commodity operating systems. Our systematic study indicates that existing metering schemes are not reliable and therefore cannot be trusted. A malicious service provider can be tempted to overcharge its users by the easiness of the attacks shown in this paper. More sophisticated attacks can also be mounted by real-world service provider to maximize its benefits. Based on the analysis of attacks, we propose three properties a trustworthy metering scheme is expected to possess.

ORGANIZATION. The next section examines related work. Section III explains the CPU metering mechanism and the problem model, and then analyzes the program execution procedure and its vulnerabilities. The attacks are described in Section IV. Section V shows the experiment results of the attacks, followed by a thorough discussion in Section VI. Section VII concludes the paper.

II. Related Work

To the best of our knowledge, there exists no official or de facto standard on CPU time metering for utility computing. Many utility, grid or cloud computing service providers have deployed their own metering and accounting schemes. Amazon Elastic Compute Cloud (EC2) is a web service providing computing infrastructure. The instance type of EC2 specifies the hardware configurations including EC2 Compute Unit, memory and storage size, etc [1]. For each type of instance, a user usually pays for the standard computing resource by hours [1]. Google App Engine [3] offers a virtual platform for the customers to run their web applications. The resources consumed by the application are free as long as they do not exceed the quotas [7]. Otherwise, the user is charged for example by CPU hours [2]. Other time related services include SUN Grid and Microsoft Azure [4], [5]. HP names its pricing unit as *computon* [6] which is a composite of processor time and other resources.

As the predecessor of cloud computing, grid computing has taken resource metering and accounting into consideration at early stage. The Open Grid Service Architecture (OGSA), which is the de facto standard for grids, encloses an accounting component. The main metric for charging a grid user also includes CPU time [8], [9]. In [10], the authors remarked that a resource owner in grid computing is motivated to cheat on the resource consumption of a task. Their viewpoint was established based on the grid architecture: a monitoring agent running on the resource owner reports the consumption. Therefore, their suggested approach is from the architecture and policies perspective. Nonetheless, our finding in this paper is that the attack can be on lower layers. The underlying operating system's resource metering and accounting are not trustworthy. As shown later in the paper, the problem is rooted at the OS kernel.

In the security literature, several schemes have been proposed to address the cheating problem for CPU consumption. The earliest work is [11], which presents a mechanism to detect any cheats in benchmark program execution. A high-end computer vendor cannot have a shortcut to run the benchmark program, since the buyer can verify it using a weaker machine. Golle et. al designed schemes to detect cheating in distributed computing [12], [13]. A distributed computing contributor must honestly run the assigned jobs in order to get rewarded. The proof-of-work scheme proposed in [14] allows a verifier to check whether certain amount of computation is accomplished by a prover. All these schemes are built on top of one-way functions. They are only suitable for a special type of computation, instead of generic applications.

III. CPU Time Metering

A. Overview of CPU Time Metering

The operating system manages all resources, including CPU time. To ensure the fairness and effectiveness of resource utilization as well as user billing in some applications, OS keeps track of each process's resource consumption. Typically, a process's CPU time in Linux consists of two portions: stime and utime. The former denotes the system time, i.e. the total amount of time spent in the *kernel mode* when the kernel runs on behalf of the process. The latter denotes the user time, namely the total amount of time spent when the process runs in *user mode*.

OS updates the CPU time utilization of a process at every *timer interrupt*, which is generated at a fixed time interval. The interval is also called jiffy or clock tick. When the timer interrupt handler is triggered to take control, it checks whether the current process is in user mode or kernel mode. Then, it adds one clock tick (or jiffy) to the process's stime or utime accordingly[1]. Other processes not running on CPU are not updated. Therefore, the resolution of CPU time accounting is the timer interrupt interval for clock tick.

Note that in the recent version of Linux kernel (2.6.23), a process's CPU time has a different composition to cater to the *Completely Fair Scheduler* (CFS). Nonetheless, it is still updated based on the timer interrupt. Thus, system and user time can be extended to the new time composition used by CFS.

B. Problem and Threat Model

To facilitate our discussion, we consider a model similar to the one used in [10]. We consider one user submitting a program to the server for execution. When the task is accomplished, the user is charged according to the amount of consumed CPU time. We remark that CPU time is different from *response time* or *turnaround time* which refers to the elapsed time between user's job submission and the result output. Compared with CPU time consumption, the turnaround time is less stable, since it significantly depends on many other factors, such as system load or the scheduling policy. Therefore, the turnaround time does not truly reflect the amount of resource consumed by a user's job.

A CPU time metering scheme is *trustworthy* if and only if the measured time equals to the outcome from the same job execution in the user's own platform with the same hardware/software specification. A dishonest service provider has a strong motivation to inflate the CPU time of a user job by various means. Without undermining our analysis, we make a few conservative assumptions to weaken the server's attack capability. First, we assume that the server does not attack on the OS kernel and the hardware. Therefore, the kernel's resource metering and accounting service is intact. Recent advances in trusted computing [15], [16] can help to safeguard the OS kernel. Second, we assume that the server does not tamper with the integrity of user's programs. In other words, the server does not modify the executables submitted by users. Third, the server does not

[1]One tick is usually 1 to 10 milliseconds.

risk the correctness of program execution if the user can verify the outputs. In certain type of applications, a user may have methods to verify the program integrity and execution correctness as shown in [17]. In the subsequent discussions, we will demonstrate that such a less aggressive adversary can still mount many types of attacks on CPU time metering. A server with stronger capability can perform more attacks, and attacks discussed in this paper still work.

We do not consider a trivial attack whereby the server simply bypasses the kernel's metering function and reports a spurious result to the user. As the kernel is trusted in our threat model, a countermeasure is that the measurement result is signed by the TPM on the kernel's request and the signature is then verified by the user.

CAVEAT. Note that the main goal of this paper is to show that the normal CPU time accounting scheme is unsuitable for the utility computing model. By studying the attacks from a weak adversary with the aforementioned assumptions, we avoid any exaggeration of our conclusion, since a real-world adversary can mount more sophisticated attacks with greater consequences.

C. A Closer Look At Program Execution and Vulnerabilities

A user program's CPU time seems to be only determined by its binary instructions and data, since it is the instructions that are executed by CPU. Nonetheless, recall that CPU time consists of both user time and system time. The latter component involves many objects (e.g. resources and events) in the running environment. The adversary can manipulate those objects to her advantage.

To fully understand the security of CPU time metering, it is desirable to examine the details of program execution. A process's life cycle consists of three consecutive phases: *process launch, runtime execution* and *exit*. They are elaborated as below.

- *Process Launch:* In this phase, the operating system prepares for the program execution. First, the OS shell creates a new process for the target program. As a child process of the shell process, the new process invokes execve system call to load the executable's image into virtual address space. execve then loads the dynamic linker and relinquishes control to it. The dynamic linker links the needed shared libraries and passes the control to the main function of the program. This completes the process launch phase and kicks off the real runtime execution.
  Note that the accounting for the program's CPU time usage starts immediately after the process is created, despite that the program's code is not executed until it enters into main(). Therefore, all these auxiliary subroutines, like the dynamic linking, are billed to the process's account.
- *Runtime Execution:* Once main() is entered, the process runs based on its own control flow logic. During the execution, it may interact with OS in several ways.
  – *shared library loading/unloading.* Similar to the library loaded at launch time, a shared library can be loaded upon the program's request.
  – *system calls.* The user program can invoke system calls to access the services offered by the operating system, e.g. file management and device management. A system call results in a state switch from the user mode to the system mode.
  – *signal, interrupt and exception handling.* Signals, interrupts and exceptions are system notification events which cause the corresponding handler to deal with them.
  – *process schedule.* Processes are subject to the operating system's scheduling. A process schedule usually occurs when the running process yields the CPU control, after an interrupt, etc.

  All the four elements above allow a malicious server to overcharge a user process's CPU time. Attacks on the first three causes unnecessary system subroutines to run on behalf of the target process, whereas the attack on process schedule exploits the flaw of the kernel's accounting mechanism. Recall that process accounting is based on timer interrupt, and however, sometimes context switch may occur between them. Therefore, an adversary can leverage this measurement inaccuracy to charge more time on the process.
- *Exit:* At exit time, the shared libraries loaded at the launch time are unloaded.

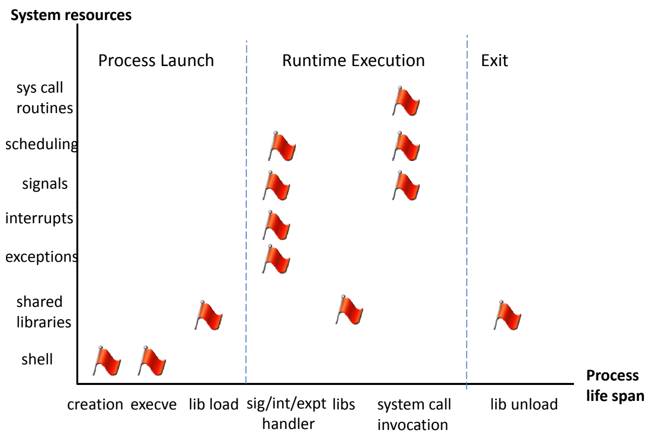

Figure 1. An overview of potential attacks on CPU time metering

To summarize, we visualize the "time-space" relation between system resources and process life span in Fig. 1 below. Each red flag at an intersection indicates a vulnerability which, as explained in the following sections, leads to an attack by exploiting the corresponding system resource object at a particular process execution phase.

IV. Attacks on CPU Time Metering

In this section, we show several types of attacks at different stages of the process life based on the previous analysis. Without loss of generality, we hereafter denote a user program by T, which is the victim of all attacks explained in the rest of the paper. T initially resides in the server's disk and waits for execution. The process running T is denoted by P_T. Hereafter, we use *server* to denote collectively the hardware/software platform and the administrator at the computation service provider whose objective is to inflate the CPU time of P_T.

A. Launch-time Attacks

During the process launch, the server prepares for T's execution. It manipulates the shell and shared library utilities to attack CPU usage metering for T.

1) Shell Attack: To execute T, the shell forks a new process P_T, which inherits from its parent process (i.e. the shell) a logical copy of the address space and context. On one hand, although P_T is created for T, it does not execute T's instructions until it runs execve() to load T; on the other hand, the kernel starts the CPU time metering for P_T once it comes into being. The time gap between the process inception and the actual program load is a good opportunity for the server to inflate the CPU time. The server tampers with the shell's code by injecting arbitrary instructions between fork() and execve(). Those instructions are executed right after P_T takes control for the first time. The CPU time utilized by those attacking instructions will be charged to P_T's account.

2) Shared Library Attack: Typically, T's execution involves not only its own executable files, but also other subroutines offered by the platform. Those subroutines are in the form of shared libraries, which can be loaded during T's startup or at runtime in an on-demand fashion [2]. In order to smoothly integrate the needed shared libraries with T, the operating system has to perform additional functions for initialization and clean-up, e.g. the constructor routines and the destructor routines. The former are executed before dlopen() returns in dynamic loading or before main() is started in startup loading. The latter are executed before dlclose() returns in dynamic loading, or after exit() or completion of main() in the cases of startup loading. Fig. 2 depicts the mixture of the original code from T, shared libraries and auxiliary functions from the server.

The utilization of shared libraries invites attacks, because the alien instructions from either the shared libraries or the library loading/unloading routines (the colored blocks in Fig. 2) are executed and credited to P_T's CPU time. Therefore, the server can achieve its goal simply by injecting arbitrary attacking code into the standard libraries or the constructor/destructor routines.

[2]The latter type is also called dynamic loading.

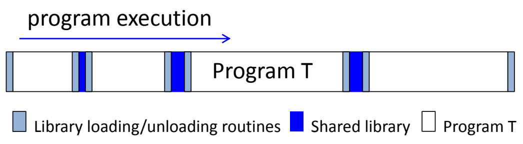

Figure 2. An illustration of shared library usage

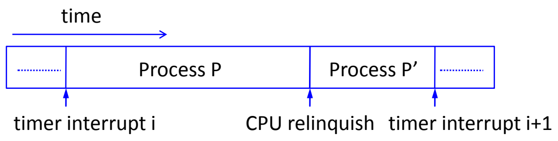

Figure 3. An illustration of scheduling attack

B. Runtime Attacks

During runtime, a program usually interacts with OS and other processes through signals, interrupts, and exceptions. The server can exploit the interaction between P_T and its execution environment to increase its CPU time. Specifically, the server inflates P_T's system time by causing the kernel to serve for P_T more than needed (as in execution thrashing, interrupt or exception flooding attacks). In addition, the server can also exploit the inaccuracy in CPU time accounting to overcharge P_T (as in process scheduling attack).

1) Process Scheduling Attack: Note that process context switch may occur in the middle of a jiffy. Therefore, the coarse granularity time accounting sometimes does not truly reflect the process's CPU time. Fig. 3 illustrates such a scenario. Process P relinquishes CPU before the arrival of the $(i+1)$-th timer interrupt. The kernel then schedules another process P' to run. Therefore, when the $(i+1)$-th timer interrupt occurs, process P' is the current process and its CPU time is increased by one tick, even though P' only occupies CPU for a portion of a tick.

The server can exploit this inaccuracy to increase P_T's CPU time. An example of this crafty attack is as follows. The server runs a malicious process P concurrently with P_T. When in running, P forks a child process and waits the child process to exit. The child process performs no operation but exits. After waking up, P repeats this cycle. Both P and its child processes relinquish CPU frequently during their execution. As a result, P_T frequently uses only a fraction of a jiffy and its CPU time consumption is expanded compared with in normal execution.

2) Execution Thrashing Attack: Many commodity operating systems offer a process trace facility which is widely used in debugging programs or monitoring processes. The trace facility allows one process to stop and resume P_T's execution by issuing "red light" signals and "green light" signals. As a result, P_T's process switches between *running* and *sleeping* states back and forth following the trace process's commands. Every state transition entails non-negligible computation for additional instructions, signal/exception handling and context switches. Therefore, the CPU time for P_T is increased. We emphasize that the objective of the thrashing attack is to expand the CPU time, instead of the turnaround time though the latter is increased more significantly.

In Linux platforms, the ptrace() system call is the function used for process tracing. Other operating systems have similar functions. ptrace() can be used to trace a process's signal handling and system calls. It can also be used to set up breakpoints of the traced process or to make a process running in the single-step mode. Without loss of generality, we use the breakpoint as an example to explain the details of the thrashing attack.

After P_T starts running, the server creates another process P. P attaches itself to P_T using ptrace() system call and becomes the parent of P_T. The attach operation stops P_T by using the SIGSTOP signal. When P is informed that P_T stops, it wakes up from wait(). It then picks an address A from P_T's virtual address space and initializes two debug registers DR0 and DR7 with A and other control information. Next, it makes another ptrace() system call to command P_T to resume its execution, and then wait again for P_T's stop. When P_T's execution accesses address A, a debug exception is generated, which stops P_T and wakes up P. Then, P repeats this attack.

3) Interrupt Flooding Attack: Most operating system takes a process-independent approach in dealing with interrupts and accounting. An interrupt causes the current running process (e.g. P) to pause since the kernel has to run the interrupt handler. The CPU time consumed by the handler is then credited to P, no matter whether the interrupt has any relevance to P or not.

Therefore, the server can take advantage of this implausible accounting mechanism to increase P_T's CPU usage. It pesters P_T by generating a flood of interrupts, e.g. I/O events. Consequently, P_T's system time is inflated. Note that in utility computing settings, a user process usually does not share the (virtual) platform with other users. Therefore, although interrupt flooding is an attack on the system, P_T is the main victim.

4) Exception Flooding Attack: Another way to increase P_T's system time is to introduce more exceptions. Similar to interrupts, an exception triggers the kernel to to save the context of the present process and run the exception handler. Since many exceptions terminate or stop the process, the server can attack T by using "soft" exceptions, which allow P_T to resume. The page fault exception due to page miss in the physical memory is a suitable choice. When P_T is running, the server may introduce frequent page faults by exhausting available memory space.

V. Experiment Results

A. Experiment Setting

We implement the attacks in the previous section to show how the server can easily attack a user process's CPU time. All experiments are run on a DELL OptiPlex 755 with Intel Duo core CPU E7200 @2.53G HZ. One core is disabled. The operating system in use is Ubuntu 8.10 with kernel v2.6.29.1. Our test programs are listed below.

1) `Our programs`: A family of programs written by us to highlight the effect in some attacks.
2) `Pi`: An open source C program to calculate the value of π [18].
3) `Whetstone`: A benchmark program to test computer performance [19].
4) `Brute`: A program to crack MD5, SHA256 and SHA512 by brute force attacks [20]. We run it to attack MD5 with the *brutefile* offered by the program author. A main feature of Brute is that it spawns many threads to search for a hash collision.

To avoid verbosity, we hereafter use **O**, **P**, **W** and **B** to denote these four programs, respectively. We implemented the attacks described in Section IV and ran them on each of the testing programs. In our experiment setting, there are no other user processes except those testing ones and the attack processes. We also shut down all unnecessary system services in order to minimize the system load. This is to simulate the dedicated platform utilization in utility computing to the largest extent. In the exit of each testing program, the getrusage is called to log the system time and user time usage.

All the figures in this section (except Fig. 7 and Fig. 8) depict the attack effects on all four programs. The Y axis shows the CPU time in seconds, while the X axis indicates different programs. For each program, there are a pair of bars. The left bar represents its CPU time in normal execution while the right one represents its CPU time with attacks.

B. Attack Effects

1) Shell Attack: We perform the shell attack by modifying the function execute_disk_command() of the bash source code. Specifically, we inject a segment of code between make_child() and shell_execve() within this function. The attacking code comprises about 2^{34} times of loops and therefore is CPU bound. Fig. 4 shows the CPU time growth of all four testing programs. Note that their system time usage are not affected by the attack. (The system time used by the first three programs are too little to be shown.) The figure also shows that the user time of the four programs are increased by almost the same amount,i.e. roughly 34 seconds. This is because the time growth is all due to the same piece of code.

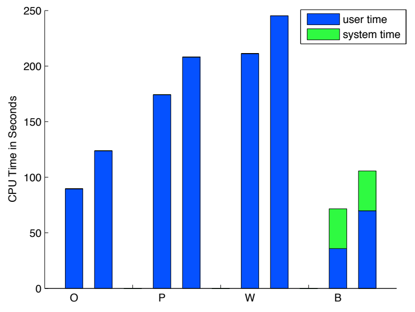

Figure 4. Shell Attack

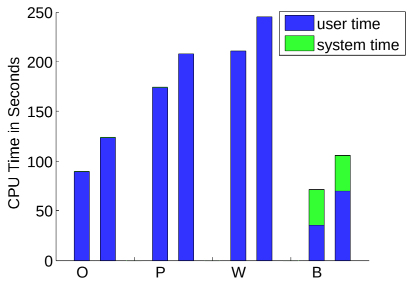

Figure 5. Shared library constructor attack

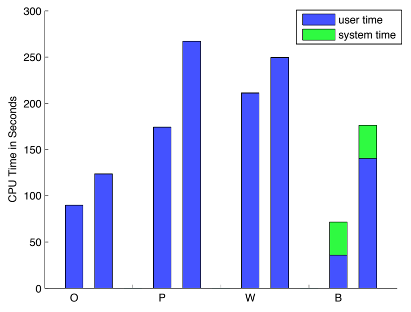

Figure 6. Shared library function substitution attack

2) Shared Library Attack: One type of shared library attacks is to tamper with the constructor and destructor functions. The two functions are compiled to an attack shared library. The constructor and destructor functions are declared as follows:

- __attribute((constructor)) void test_init_t(void)
- __attribute((destructor)) void test_fini_t(void)

We only implement the constructor and the destructor is similar. We set the LD_PRELOAD environment variable to the malicious shared library, so that it will be loaded when the test programs are launched. The results are shown in Fig. 5. Not surprisingly, they are almost identical to Fig. 4. In essence, the same attacking code is executed at different locations.

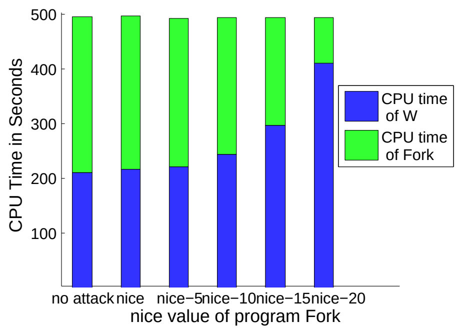

Figure 7. Process scheduling attack on Whetstone

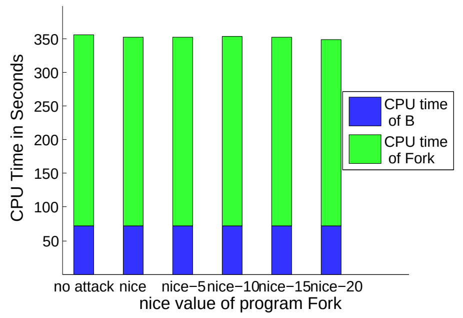

Figure 8. Process scheduling attack on Brute

The server can also inject the attacking code into a shared library function. We fake a malicious malloc() function, which first executes the attacking code and then call the genuine malloc() function. We set the LD_PRELOAD environment variable to overwrite the path to the genuine library and use the faked one. The same trick is played for the sqrt() function. Fig. 6 shows the attack effect. Similar to previous experiments, the system time usage is not affected. The only difference from previous experiment is that the effects are amplified, since the faked function containing the attacking code is invoked multiple times.

3) Process Scheduling Attack: We implement an attack program described in Section IV-B1, which runs a cycle of snatching CPU control and then relinquishing it. The attack program quits after forking 2^{21} child processes. We applied this attack on **O, P** and **W**, and observed the similar result. Fig. 7 shows the effect of attacking **W**.

The green bars (the upper one) in the figure represent the CPU time used by our attack program, whereas the blue bars (the lower one) represent CPU time used by Whetstone. The X axis shows different priorities(different nice values) of the attacking program Fork. The leftmost bar pair show the CPU time used by **W** and the attack program **Fork** when they are executed independently, i.e. no attacks. The second bar pair show both program's time consumption when running concurrently, i.e. the attack is in place. The CPU time change becomes more noticeable when the attacking program's priority becomes higher. Although

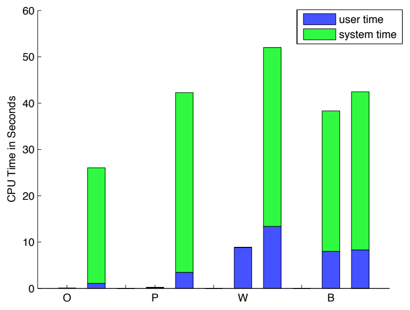

Figure 9. Execution thrashing attack

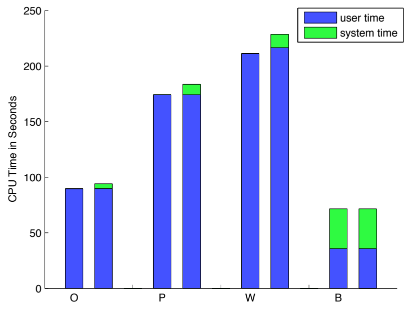

Figure 10. Interrupt flooding attack

both programs start and exit normally, the CPU time for the attacking program decreases while the time for Whetstone increases. More importantly, the sum of them almost remains the same. It implies that a portion of time used by the attacking program is mistakenly accounted to **W**. Note that the higher the privilege of the attack program, the CPU time of **W** is increased more. The possible reason is that program Fork has longer quantum within a scheduling epoch and scheduling happens more frequently for higher priority Fork program. Therefore, the higher privilege of the attack program implies more chances for CPU time miscounting.

This attack is not effective for Brute program, as shown in Fig. 8 because it forks many threads which are scheduled as processes in Linux. Therefore, the accounting error does not affect the overall time significantly.

4) Execution Thrashing Attack: We implement the thrashing attack by setting breakpoints. Program **O** is similar to it in the shell attack. Breakpoint is set at the loop control variable frequently accessed. For **B**, the breakpoint is set at a variable count in crack_len() function. To maximize the effect, we set PER_THREAD_TRIES to 50 so that this breakpoint is accessed for about 895 thousand times. For **P** and **W**, we choose variable y and T1 which are accessed about 10^7 times and 2×10^5 times respectively. The result is shown in Fig. 9. Clearly, the attack mostly increases the system time of the test process due to the exception, signal handling and context switches.

5) Interrupt Flooding Attack: We create an interrupt flood by sending junk IP packets to the testing computer from another PC. An interrupt is raised whenever the testing computer's network adapter receives a packet. Since none of the target programs needs network communications, the flood of packets will not be received and therefore has no consequences to their execution results. We applied the attack on four target programs one by one. The results are shown in Fig. 10. Note that their system time are slightly increased, due to the extra interrupt handler.

6) Exception Flooding Attack: Following discussion in Section IV-B4, we create a flood of page fault exceptions by running an attack program which exhausts all available physical memory. Specifically it requests more than 2 gigabytes memory which is beyond the capacity of the physical memory in the testing platform. To force kernel to allocate memory, or reallocate memory if it is reclaimed, the attack program continuously writes data and reads them later. The results are shown in Fig. 11. When memory is scarce, page swap from disk is more frequent, resulting in more exception handling and context switches. Another factor is the disk accesses which also contribute to the CPU time increase.

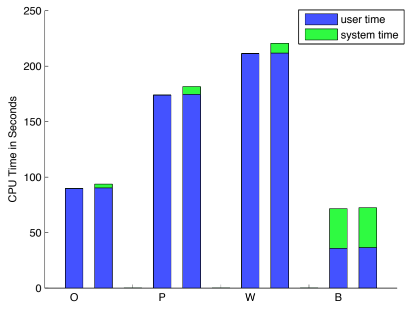

Figure 11. Exception flooding attack

C. Comparison

We make a comparison among the attacks discussed in the previous section. The comparison is based on the following aspects: vulnerabilities exploited, attack strength and side effect.

Vulnerabilities Exploited. The shell attack and the shared library attacks aim at P_T's user time. They exploit the fact that P_T encloses not only the program T, but also additional code segments from the system. The execution thrashing attack, and the interrupt and exception flooding attacks aim at P_T's system time. These three attacks trigger off unsolicited system events which cause the kernel to use extra CPU time to make proper responses. If the spamming event happens during P_T's execution, the kernel's service cost is accounted to P_T. The process scheduling attack mainly exploits the imprecision of CPU time metering, which is caused by the coarse granularity of the kernel's time metering scheme.

Attack Strength. The shell and shared library attacks are the most aggressive one because the inflated CPU time can be arbitrarily long. In other attacks, the server cannot predict precisely the amount of cheating. The interrupt flooding and the exception flooding are the weakest, since they do not impact CPU time significantly because of the following two reasons. First, the kernel's execution of interrupt/exception handlers and context switch are much less costly than the execution of user processes. Secondly, the number of interrupts and exceptions generated cannot be very huge. Moreover, for interrupt attack, the probability for P_T to get hit by them depends on whether it is running on CPU at that moment. Therefore, P_T's extra CPU time is only a share of the overall overhead. For exception, if the physical memory is very heavily used, a process will be killed by the kernel due to lack of physical memory resource. Thus, the amount of issued page fault is capped.

The execution thrashing attack can be very effective as the attack directly impacts P_T's execution. The CPU time inflation can be adjusted by the number of breakpoints. The effectiveness of the process scheduling attack is comparatively less predictable, because it relies on many runtime factors, e.g. the scheduling algorithm, the system load, etc.

Side Effects and Limitations. All of the attacks have side effects. The shell attack increases the CPU time for all programs started from the same attacked shell. The shared library attack inflates the time for all programs calling the library functions. Nonetheless, these side effects can be mitigated by customizing the settings for the target user with a designated shell and local environment variables.

The interrupt flooding and the exception flooding are essentially a denial-of-service attack on the whole system. Therefore, all system processes suffer from it. The execution thrashing attack has the least side effect, since it exactly aims at P_T. Nonetheless, the execution of ptrace (or utrace in newer kernel versions [21]) requires privileges controlled by the Linux Security Modules [22], [23]. The process scheduling attack also requires the root privilege since the attack program has to elevate its own priority. These privilege prerequisite may not be satisfied in all utility computing applications.

VI. Discussion

A. Trustworthiness in CPU Time Accounting

The ultimate goal of utility computing is for users to tap the computing service as a utility. Our discussion has shown that unlike other utilities, it is not straightforward to provide trustworthiness of CPU time metering. Without tampering with the operating system kernel or the executed program, the server can still easily mount various attacks to inflate the program's CPU usage. This has grave negative impact to the success of utility computing. No matter whether CPU hours or CPU seconds are used as the pricing unit, the important caveat is that the existing metering scheme is not qualified as the foundation of trust to run business applications. On the one hand, the weak trustworthiness of the metering scheme allows a server to cheat. On the other hand, it allows the customers to deny their bills by questioning the trueness of the usage. Therefore, we remark that a secure CPU time metering is one of the bedrocks for trust establishment between customers and service providers and therefore is pivotal to utility computing.

B. Desirable Properties

Based on the attack analysis, we define several desirable security properties for a platform to fairly and securely measure a process's CPU time. We also suggest promising approaches to achieve the properties.

Source Integrity. This property refers to that only the expected code should be executed in the context of a user process. The expected code includes the programs submitted by the user and other standard subroutines expected to be executed. The subroutines can be code at application level, OS level and kernel level.

The shell attack and the library attacks in Section IV are essentially attacks on source integrity. Other attacks include malicious modifications on the user program or other system libraries in order to increase the execution time.

One possible solution is the TPM based remote attestation [15], [16], [24] proposed by the trusted computing community. The scheme in [24] allows the server to attest the closure of a program (i.e. all instructions possibly to be run) to a verifier. The security of the attestation scheme is built upon the TPM chip [16]. Unfortunately, all existing remote attestation schemes including [24] suffer from the gap between the time-of-measure and time-of-use.

Execution Integrity. Note that the attacks described in Section IV are rather weak in the sense that they do not tamper with the program execution. A more sophisticated attack would be that the server tamper with the control flow of a program. For example the server may modify the registers which controls the flow so that the process takes a much longer execution path to finish. Control-data attack and non-control-data attack [25] can all be used to corrupt program execution. Some possible attacks would be performed by changing the input from terminal, disk and other I/O device, Inter-Process Communication, etc. Execution integrity is to ensure that such attacks on the program control flow can be detected by the users. This task is more challenging than source integrity attack because the program outputs are not always verifiable. The proof-of-work schemes are proposed in [11], [13], [14], [26]. Most of them only deal with well-structured mathematical functions. Since they are not suitable for generic software, it is not applicable to utility computing. It is an open problem of verifying the integrity of the control flow.

Fine-grained Metering. Existing commodity operating systems use coarse-grained CPU time metering scheme. A fine-grained metering scheme has twofold implications. First, the code being measured is fine-grained, in the sense that only those instructions executed on behalf of a process should be added to its CPU time. In existing metering scheme, the execution of an I/O interrupt handler is accredited to the current running process. This is unfair if the running process is not the one issuing I/O request or receiving the data. The interrupt flooding attack introduced before actually exploits this defect. In fact, this problem has been addressed in the real-time system research, e.g. in [27], where the system timely response is of paramount importance. Unfortunately, the general purpose operating systems have not adopted the proposed process-specific scheduling and accounting.

The other implication of fine granularity is the time metric in use. Many operating systems only count the number of system ticks used by the process, which is usually at the granularity of milliseconds. The process scheduling attack takes advantage of this defect. Although a process is only executed in a fraction of a tick, its CPU time will be increased with one tick if it is running when the timer interrupt occurs. In fact, most modern processors has a built-in *time stamp counter* (TSC) which can be accessed by mainstream operating systems like Windows and Linux. For the utility computing, a more accurate CPU time metering scheme can be built on TSC.

C. Metering Other Resources

Utility computing may include other services besides CPU power. For instance, a service provider may bill a user based on the number of database transactions, the number of bytes her task communicates, or the amount of storage her data occupies. For these types of services, the charged resource consumption is possibly easier to verify, because they are transaction oriented. For example, a user uploads a file to the server and keeps a log on the file size. Therefore, the user can verify the claimed resource utilization by comparing it with her local transaction log.

VII. Conclusion

In this paper, we study the trustworthiness of CPU time metering and accounting in utility computing settings. We remark that the existing metering and accounting scheme provided by the general-purpose operating system does not satisfy the security requirement of utility computing. We have described and implemented several attacks whereby the server can inflate a user process's system/user time without subverting the program or the kernel. More sophisticated attacks without these restrictions can also be mounted by the real-world server. In the future, more attacks on virtual machine model will be studied. Based on our analysis on these attacks, we suggested source integrity, execution integrity and fine-grained metering as three properties an CPU time metering scheme should necessarily have to fairly and securely measure a process's CPU time for utility computing or cloud computing.

Acknowledgment

This research was partly supported by the Office of Research, Singapore Management University and by National Science Foundation of China under grant 70890084/G021102.

References

[1] (2010, Feb.) Amazon Elastic Compute Cloud (Amazon EC2). [Online]. Available: http://aws.amazon.com/ec2/

[2] (2010, Feb.) Billing and budgeting resources. [Online]. Available: http://code.google.com/intl/en/appengine/docs/billing.html

[3] (2010, Feb.) What is Google App Engine? [Online]. Available: http://code.google.com/intl/en/appengine/docs/whatisgoogleappengine.html

[4] (2010, Feb.) Windows Azure platform pricing. [Online]. Available: http://www.microsoft.com/windowsazure/pricing/

[5] *Sun Grid Compute Utility User's Guide*. Sun Microsystems, 2007.

[6] T. Hoffman. (2010, Feb.) HP takes new pricing path for utility-based computing. [Online]. Available: http://www.computerworld.com/s/article/81522/HP_takes_new_pricing_path_for_utility_based_computing?taxonomyId=074

[7] (2010, Feb.) Quotas. [Online]. Available: http://code.google.com/intl/en/appengine/docs/quotas.html

[8] (2009, Aug.) The distributed grid accounting system (DGAS). [Online]. Available: http://www.to.infn.it/grid/accounting/main.html

[9] A. Barmouta and R. Buyya, "Gridbank: A grid accounting services architecture (GASA) for distributed systems sharing and integration," in *Proceedings of IPDPS*, 2003.

[10] L. Catuogno, P. Faruolo, U. F. Petrillo, and I. Visconti, "Reliable accounting in grid economic transactions," in *Proceedings of GCC Workshops*, 2004, pp. 514–521.

[11] J. Cai, R. Lipton, R. Sedgewick, and A. Yao, "Towards uncheatable benchmarks," in *Proceedings of IEEE Structure in Complexity Theory Conference*, 1993, pp. 2–11.

[12] P. Golle and I. Mironov, "Uncheatable distributed computations," in *Proceedings of RSA Conference, Cryptographer's Track*, 2001, pp. 425–440.

[13] P. Golle and S. Stubblebine, "Secure distributed computing in a commercial environment," in *Proceedings of Financial Cryptography*, 2002, pp. 289–304.

[14] M. Jakobsson and A. Juels, "Proofs of work and bread pudding protocols (extended abstract)," in *Proceedings of Communications and Multimedia Security*, 1999, pp. 258–272.

[15] R. Sailer, X. Zhang, T. Jaeger, and L. van Doorn, "Design and implementation of a TCG-based integrity measurement architecture," in *Proceedings of USENIX Security Symposium*, 2004, pp. 223–238.

[16] Trusted Computing Group, "TPM main specification," Trusted Computing Group, Main Specification Version 1.2 rev. 103, 2007. [Online]. Available: http://www.trustedcomputinggroup.org/resources/tpm_main_specification

[17] C. S. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation - tools for software protection," *IEEE Trans. Softw. Eng.*, vol. 28, no. 8, pp. 735–746, 2002.

[18] (2010, Feb.) Pi. [Online]. Available: http://sourceforge.net/projects/projectpi/

[19] (2010, Feb.) Whetstone. [Online]. Available: http://www.netlib.org/benchmark/whetstonec

[20] (2010, Feb.) Brute. [Online]. Available: http://sourceforge.net/projects/brute2/

[21] J. Keniston, A. Mavinakayanahalli, P. Panchamukhi, and V. Prasad, "Ptrace, utrace, uprobes: Lightweight, dynamic tracing of user apps," in *Proceedings of the Linux Symposium*, 2007, pp. 215–224.

[22] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman, "Linux security modules: General security support for the Linux kernel," in *Proceedings of USENIX Security Symposium*, 2002, pp. 17–31.

[23] P. Loscocco and S. Smalley, "Integrating flexible support for security policies into the Linux operating system," in *Proceedings of USENIX Annual Technical Conference, FREENIX Track*, 2001, pp. 29–42.

[24] L. Gu, X. Ding, R. H. Deng, B. Xie, and H. Mei, "Remote attestation on program execution," in *Proceedings of the Workshop on Scalable Trusted Computing, STC*, 2008, pp. 11–20.

[25] S. Chen, J. Xu, E. C. Sezer, P. Gauriar, and R. K. Iyer, "Non-control-data attacks are realistic threats," in *Proceedings of USENIX Security Symposium*, 2005, pp. 177–192.

[26] F. Monrose, P. Wyckoff, and A. Rubin, "Distributed execution with remote audit," in *Proceedings of Network ad Distributed System Security Symposium (NDSS)*, 1999, pp. 103–113.

[27] Y. Zhang and R. West, "Process-aware interrupt scheduling and accounting," in *Proceedings of the International Real-Time Systems Symposium*, 2006, pp. 191–201.