Is Your PC Secretly Running Nuclear Simulations?

Dan Tsafrir* Yoav Etsion* Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University of Jerusalem
91904 Jerusalem, Israel

Abstract

Large-scale attacks on networked computers are typically used to create zombie armies for propagating spam or launching DDoS attacks. But they can also be used to harvest immense computing power for use in running nuclear simulations or cryptanalysis attempts. We show that all major operating systems today (possibly except Mac OS X) are vulnerable to such attacks, due to a combination of how they account for CPU usage and how they prioritize competing processes. Specifically, we detail a "cheat" attack, by which a non-privileged process can hijack a large percentage of the CPU cycles. Moreover, in at least some of the systems, listing the active processes will erroneously show that the cheating process is not using any CPU resources at all, making it hard to detect the attack. This can also be a "desirable" feature for spyware designers. We also show that such vulnerabilities are actually quite simple to avoid, and we demonstrate this by implementing a patch for Linux 2.6.

1 Introduction

The schedulers of general-purpose multitasking operating systems typically try to equalize the resources allocated to competing processes. But the success of such attempts depend on knowing how much each process is using. If the accounting for CPU usage can be circumvented, creating a process that uses significant CPU resources but looks as if it does not, the equalization will fail. And in fact the accounting can indeed be circumvented, because schedulers do not measure CPU usage directly but rather by sampling it using periodic clock interrupts.

Periodic clock interrupts (a.k.a. ticks) are a basic design feature in all major operating systems. Ticks go way back, since the dawn of operating systems research in the 1960s. The first paper we are aware of describing this mechanism is the famous paper by Dijkstra from 1968, about the THE system [4], where the motivation for introducing ticks was to implement multitasking and make the system more deterministic and amenable to formal analysis. But their use nowadays is probably more related to the fact that they simplify the design of the system [3, 16, 15, 20, 23].

Operating systems are *reactive* by nature. Most of the time they just wait for an interrupt to happen; when it does, they handle it and return to wait for the next interrupt. But they also have a *proactive* component, where they need to take the initiative. A key example is using time slicing to support multiprogramming: the operating system then has to stop one process and dispatch another in its place. The system design is simplified by unifying the mechanisms for reactive and proactive activities, and making proactive activities contingent on periodic timer interrupts.

While operating systems may have many different types of proactive activities, for our purposes it suffices to focus on three:

- Making scheduling decisions and performing a context switch from one process to another,
- Sampling the running process for accounting purposes,
- Noting the passage to time in order to support a timer service (i.e. waking up a process that requested to sleep for some time).

Importantly, all of these activities are typically tied to the same clock interrupts. This overloading can be exploited by a simple attack that uses the timer to ensure that a process always starts to run just after a clock tick, but stopping it before the next tick. As a result the process

*Both authors contributed equally.

is never billed, because it is never the process that was sampled by a clock tick.

One reason to try and equalize the resources given to different processes is that this is supposed to ensure that interactive processes are responsive. Traditionally, interactive processes have been identified by their relatively low use of the CPU. To equalize resources, processes that use little CPU are prioritized. Avoiding being billed for CPU usage therefore translates directly to a higher priority. This allows the cheating process to monopolize the CPU, as we will show below. Importantly, such a "cheat attack" can be launched by any non-privileged user process: there is no need for any rootkits or other sophisticated exploits. We find that almost all contemporary operating systems are susceptible to such an attack.

An especially alarming consequence of the cheat attack is that the cheating process becomes essentially invisible. The most basic defense one has against malicious programs is seeing them run using a monitoring tool (such as 'top', 'xosview', etc.) when sensing that something is wrong. For example, our departmental system administrators always maintain a screen full of such monitors on the main servers we have. But if the system doesn't account for the CPU usage of the attacking process, it won't show up on the monitors. Even worse, the attack actually leads to *misaccounting*, where another process is billed for CPU time used by the cheating process. As a result, even if the system administrators suspect something, they will suspect the wrong processes. The cheating process can further disguise its tracks by controlling the amount of CPU it uses so as not to have too great an impact on system performance.

The fact that the cheating process is naturally hidden from view makes it a useful vehicle for sophisticated spyware, which does not only record and transmit data but may also process it locally first. It also makes it possible to harvest large amounts of computing resources. The history of Internet worms and viruses is well known and documented. The Code Red worm which infected ˜360,000 hosts in July 2001 was a trigger for several studies on characterizing and preventing such infections [17, 18]. But despite the obvious motivation to protect against such worms and viruses, they continue to plague the Internet [24]: the Slammer worm (Jan 2003), Sasser worm (May 2004), and even the recent Nyxem worm (Jan 2006) prove that the threat is still very real (and this is an extremely partial list). Several researchers have even suggested attacks that can infect over 10,000,000 hosts [22]. Combining such worms with our cheat attack can be used to create an ad-hoc supercomputer, and run a computational payload on massive resources in minimal time. Possible applications include cracking encryptions in a matter of hours or days, running nuclear simulations, and illegally executing a wide range of otherwise regulated computations.

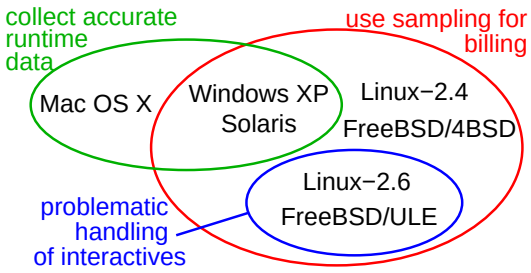

Figure 1: *Classification of major operating systems in terms of features relevant for the cheat attack.*

The principles of the cheat attack are detailed in the next section, and Section 3 then shows how the attack is put into practice for all major operating systems in use today. The security hole is so simple and easy to exploit, it is actually very surprising it hasn't been used (of course, maybe it is but nobody knows about it...) While this attack is very simple, it has never been published as far as we could ascertain; the most explicit mention we found is a short comment in a kernel source file of FreeBSD, and a single paragraph in a 930-page book on Windows (and we managed to compromise both these systems). But more interestingly, the existence of this attack serves to motivate a renewed discussion of the age-old practice of using ticks as a basic design feature in modern operating systems.

The findings derived from our implementations are summarized in Fig. 1. This points out that there are two ways to account for CPU usage: direct measurement or sampling. Amazingly, even some systems that actually perform accurate measurements do not use this information for scheduling (specifically, this is the case in Windows XP and Solaris). In addition, some systems (Linux 2.6 and the ULE scheduler for FreeBSD) suffer from problematic prioritization practices regarding interactive processes, that further increase their vulnerability. In our experience, only Mac OS X was immune from our attack. Luckily a simple solution, which in addition is almost overhead-free, is at hand: we prove this by fully implementing it in Linux. This proposal for a comprehensive solution, which is based on accurate billing, is presented in Section 4, followed by the conclusions in Section 5.

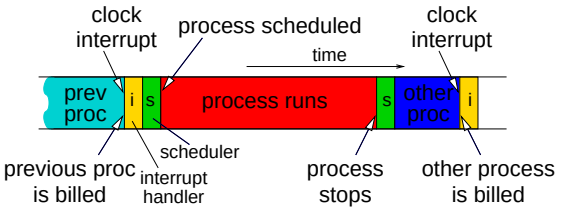

Figure 2: *The cheat attack is based on a scenario where a process starts running immediately after one clock tick, but stops before the next tick, so as not to be billed.*

2 The "Cheat" Attack

The "cheat" attack exploits two operating system mechanisms: the use of periodic sampling to account for CPU usage, and the prioritization of processes that use less of the CPU. The idea is to avoid the accounting, and then enjoy the resulting high priority.

2.1 Using the CPU Without Being Billed

All current major operating systems use a sampling-based approach for billing. When a hardware clock interrupt occurs, the interval since the previous interrupt is billed to the process that ran just before the interrupt occurred. On the long term, this typically provides reasonably accurate billing, due to the random nature of the sampling. A process that runs for a short period each time it is scheduled will typically not be billed, but when it is, it will be billed for a whole tick; in general, processes that use more CPU time have a higher chance of being interrupted and billed.

Assume for the moment that the operating system uses the same clock ticks for all three activities listed above: sampling CPU usage, firing requested timers, and context switching (the exact behavior of real systems is detailed below). In this case, a process will not be billed if the scenario described in Fig. 2 occurs. This scenario has two components:

1. Start running after one billing sample.
2. Stop running before the next billing sample.

Starting to run immediately after a billing sample is relatively easy. Recall that the billing sample is done upon a clock interrupt. The same clock interrupt is also used to fire any pending timer requests. So if a process blocks on a timer, it will be released just after a billing sample (and in particular, setting a very short timer interval will wake you up on the very next tick). If, in addition, it will have a relatively high priority, it will also start running. This is also not a problem, because blocking typically results in a priority boost.

```
cheat_attack( percent ):
    work ← (percent/100) × get_cycles_per_tick()
    sleep( ε ) // wakeup at next tick
    tick_start ← read_cycle_counter()
    while( TRUE ) {
        now ← read_cycle_counter()
        if (now − tick_start ≥ work) {
            sleep( ε )
            // avoid being billed
            tick_start ← read_cycle_counter()
        }
        // do some short work here...
    }
```

Figure 3: *Pseudo code for the cheater process.*

The harder part is to stop running before the next clock interrupt, when the next billing sample will occur. This may happen by chance, if the required amount of processing is simply less than the interval between clock ticks. In fact, we have observed such a situation with the Xine movie player and X server. Xine set a timer to display each subsequent frame, but the X server finished displaying it within about 0.8 of a clock tick, thereby being billed for only about 2% of the CPU time it actually used [7]. The question is how to do this on purpose.

2.2 Crafting a "Cheat" Application

In order to stop running before the next clock tick, the process needs an alternative timing mechanism. Luckily, such a mechanism exists in the form of the cycle counter available on Pentium and other architectures [12, 11]. This counter is readable from user level using special assembler instructions [5]. In the following, we assume that this is encapsulated in the function read_cycle_counter for convenience.

Given access to the cycle counter, an application that uses any desired percentage of the CPU can be written as in shown in Fig. 3. This first finds the number of clock cycles that constitute the desired percentage of the clock tick interval. It then iterates doing its computation, checking whether the desired limit has been reached at each iteration. When the limit is reached, the application goes to sleep for ϵ time, which blocks it till after the next tick. The only assumption is that the computation can be broken into small pieces, which is technically always easy to do.

In order to know how long to run, the cheat application

```
get_cycles_per_tick():
    N ← 1000
    sleep( ε ) // sync with tick
    start ←read_cycle_counter()
    for i = 1, 2, ...N
        sleep( ε )
    now ←read_cycle_counter()
    return 1/N × (now − start)
```

Figure 4: *Pseudo code for finding the tick interval.*

needs to know the interval between clock ticks in cycles. This can be done easily by a short checking routine such as that shown in Fig. 4 (among others). To obtain more precise results, it is possible to tabulate all N timestamps individually, calculate the intervals between them, and exclude outliers that indicate that some other activity interfered with the measurement (which is approximately what we did in the implementations). Note that this code assumes that the same ticks are used for accounting and for timers. This is not necessarily the case. Dealing with the complexities of using different clocks in real systems is described in Section 3.

The above code solves the problem of knowing when to stop to avoid being billed. As a result, this non-privileged application can commandeer any desired percentage of the CPU resources, while looking as if it is using 0 resources.

To demonstrate that this indeed works as described, we implemented such an application and ran it on a Linux 2.6.16 system (the latest currently available). The system was a default installation with the usual daemons, and no other user processes except our tests. The application didn't do any useful work — it just burned cycles. At the same time we also ran another compute-bound application, that also just burned cycles. An equitable scheduler should have given each about 50% of the machine. But the cheat application was set to use 80%, and got them.

During the execution of the two competing applications, we monitored every important event in the system using the Klogger tool [9]. A detailed rendition of precisely what happened is given in Fig. 5. This shows 10 seconds of execution along the X axis, at tick resolution. As the system default tick rate is 250 Hz, each tick represents 4ms. To show what happens during each tick, we spread those 4ms along the Y axis, and use color coding. Evidently, the cheat application is nearly always the first one to run (on rare occasions some system daemon runs initially for a short time). But after 3.2ms (that is, exactly 80% of the tick) it blocks, allowing the honest process or some other process to run.

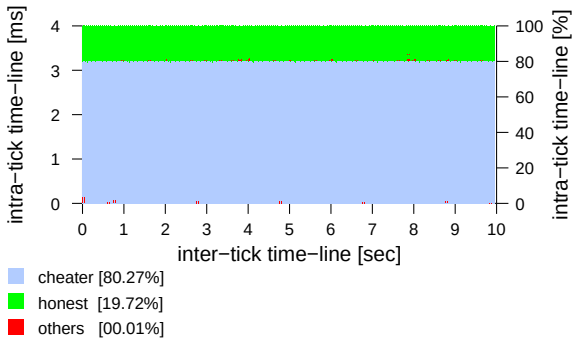

Figure 5: *Timeline of 10 seconds of competition between a cheat process and an honest process. Percents in the legend give the distribution of CPU cycles.*

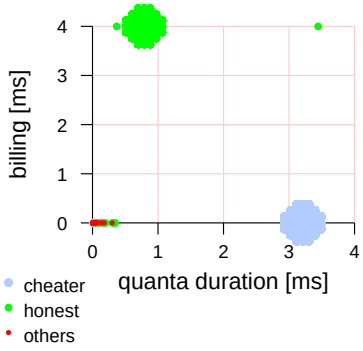

Figure 6: *Billing accuracy achieved during the test shown in Fig. 5.*

Fig. 6 shows the billing accuracy that was achieved during this run. The cheat process runs for just over 3ms each time, but is billed for 0. The honest process, on the other hand, typically runs for less than one ms, but is billed for 4; on rare occasions it runs for nearly a whole tick, probably due to some interference that caused the cheater to miss one tick; the cheater nevertheless recovers on the following tick. the other processes run for a very short time and are not billed.

Like any monitoring utility, the view presented to the user by the 'top' utility is based on the operating system billing information, and presents a completely distorted picture (Fig. 7). This dump was taken about 8 seconds into the run, and indeed the honest process is billed for 99.3% of the CPU and is reported as having run for 7.79 seconds. The cheater process, on the other

```
Tasks:  70 total,   3 running,  67 sleeping,   0 stopped,   0 zombie
Cpu(s): 99.7% us,  0.3% sy,  0.0% ni,  0.0% id,  0.0% wa,  0.0% hi,  0.0% si
Mem:    513660k total,   306248k used,   207412k free,        0k buffers
Swap:        0k total,        0k used,        0k free,   227256k cached

  PID  USER   PR  NI  VIRT   RES   SHR  S  %CPU  %MEM    TIME+  COMMAND
 5522  dants  21   0  2348   820   728  R  99.3   0.2   0:07.79  honest
 5508  dants  16   0  2232  1168   928  R   0.3   0.2   0:00.04  top
 5246  dants  16   0  3296  1892  1088  S   0.0   0.4   0:00.04  csh
 5259  dants  16   0  3304  1924  1088  S   0.0   0.4   0:00.06  csh
 5509  dants  16   0  3072  1552   964  S   0.0   0.3   0:00.03  bm-no-klog.csh
 5521  dants  15   0  2352   828   732  S   0.0  0.27   0:00.00  cheater
```

Figure 7: *Snippet of the output of the 'top' utility for user dants (the full output includes dozens of processes, and the cheater appears near the end and is hard to notice). The honest process is billed for 99.3% of the CPU, while actually getting only 20%. The cheater looks as if it is not getting any CPU, while it actually consumes 80%.*

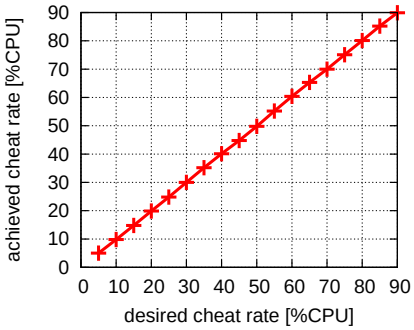

Figure 8: *The attack is very accuracy and the cheater gets exactly the amount of CPU cycles it requested.*

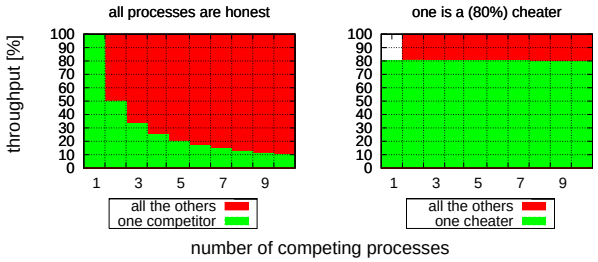

Figure 9: *Cheating is immune to the background load.*

hand, is shown as using 0 time and 0% of the CPU. Also note that the cheater process is viewed as suspended (status S), further throwing off any system administrator that tries to understand what is going on. As a result of the billing errors, the cheater has the highest priority (lowest numerical value: 15), which allows it to continue with its exploits.

The above demonstration used a setting of 80% for the cheat application (this is the percent argument from the code in Fig. 3). Fig. 8 shows that the attack is very accurate, and can achieve precisely the desired level of usage. Thus an attacker that wants to "lay low" can set the cheating to be a relatively small value (e.g. 15%). The chances the user will notice this are very slim, especially in operating systems like Linux where the monitoring utility 'top' reports the cheater as consuming 0%.

Additionally, the attack is successful regardless of the background load, and the cheater gets its way even in a heavily loaded system. This is shown in Fig. 9, which compares the percentage of the CPU that an honest process gets to the percentage that it gets when it employs the cheat attack. When honest, the process gets its equal share, which drops as more competing processes are added. For example, when a total of 4 processes are present, each gets 25%. When cheating, it always gets what it wants (in this case, 80%). The reason of course is that the cheater has very a high priority, as it appears like a process with no CPU consumption whatsoever, which implies an immediate service upon wakeup.

2.3 Running Unmodified Applications

A potential drawback of the above design is that it requires the application to be modified to incorporate the cheat code. But it is also possible to create a cheat client/server architecture that runs unmodified application without ever being billed. Given an application app, this is done by the command

```
cheat 80% app
```

The problem facing a cheat application is that it does not have access to the machine's timing facilities. It

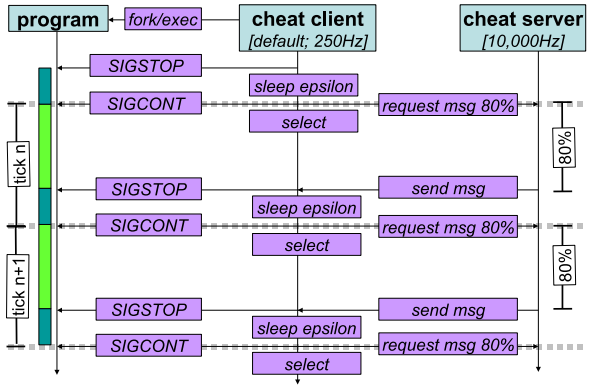

Figure 10: *The cheat server protocol.*

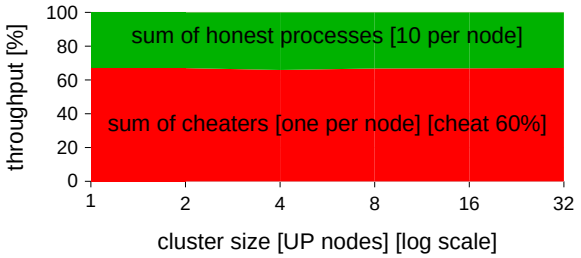

Figure 11: *The combined throughput of honest vs. cheating processes, as a function of the number of cluster nodes used. On each node there are ten honest processes and one cheater running. As indicated by the cheaters' throughput (which is as requested), the cheat-server simultaneously provides good service to all the cheat clients.*

therefore needs an external cheat server, with a tick rate that is significantly higher than that of the target machine. Assuming the cheat client has already found the tick rate of the target machine (e.g. using the above read_cycle_counter routine), it operates as follows (Fig. 10):

1. Initially, the cheat client forks the target application, and sends it a stop signal.
2. The client informs the server about the tick interval on the target machine.
3. The cheat client then goes to sleep till the next tick.
4. Upon being awoken on a tick, the cheat client does the following:
   (a) It sends the external cheat server a request for a timing message that will arrive before the next local clock tick. The request includes the exact percentage of the tick interval to use. This is possible since the timing server provides a higher resolution. Note that the network latency needs to be taken into account too.
   (b) It sends the target application a cont signal to wake it up.
   (c) It blocks waiting for the message from the cheat server to arrive.
5. As the cheat client blocks, the operating system will most probably dispatch the application that was just unblocked (because it looks as if it is always sleeping, and therefore has high priority).
6. At the due time, the cheat server sends its message to the cheat client. This causes a network interrupt, and the scheduling of the cheat client.
7. The cheat client now does two things:
   (a) It sends the target application a stop signal to prevent it from being billed
   (b) It goes to sleep till the next (local) tick

   Upon the next tick, it will resume from step 4.

We have implemented this scenario on a cluster of Pentium IV machines to demonstrate that it works. As a timing server we used an old Pentium III machine, with a Linux 2.6 system clocked at 10,000 Hz. While such a high clock rate creates significant overhead [7], this is acceptable since the timing server does not have any other responsibilities. In fact, it could easily generate the required timing messages for the full cluster size, which was 32 cheat clients in our case, as indicated by Fig. 11.

2.4 Potential Exploits

Cheat applications or servers as described above are potentially useful within an organization, enabling a user to monopolize computing resources that should be shared with others. For example, a user could hijack the CPU cycles of a cluster of workstations unbeknown to the cluster's administrators or other users, who would only see CPU utilization drop with no explanation.

A more serious exploit would occur if a cheat application was spread using a computer virus or worm. Modern computer viruses/worms can spread very quickly and infect millions of computers [22]. In many cases these

viruses are found and uprooted because of their very success, as the load the place on the Internet becomes unignorable. But if a virus throttled its infection rate to avoid detection, and then installed an application that was essentially invisible to the host system, it would potentially be able to harvest a huge computing infrastructure, similar to that amassed by projects like SETI@home [1].

This potential development is very worrying, as it foreshadows a new type of exploit for computer viruses. So far computer viruses targeting the whole Internet have been used mainly for launching attacks or spam email [17]. Exploits for direct profit, by contrast, required breaking into specific systems where the desired information was stored, and installing rootkits. But cheat attacks can be used to steal immense computing power, which can be used to crack encryptions, run military codes such as biological and nuclear simulations, etc. I addition, they can be used to run more sophisticated and compute intense spyware without being detected.

3 Implementing the Attack on Major Systems

The cheat attack depends on the confluence of two operating system mechanisms: the use of periodic sampling to account for CPU usage, and the prioritization of processes that use less of the CPU. For the attack, it is enough to circumvent the first. In principle, this is done by devising a setup where the sampling always misses the cheating process, as described above in Fig. 3. However, the specific implementation depends on the platform. In this section we will overview the scheduling and timing mechanism of major general purpose operating systems, and explain the attack can be performed on each system. The main differences between implementations are the manner in which the process waits for the next tick to fire, and whether estimating the next tick's time is performed from within a signal handler or called explicitly when the process regains control after the sleep.

3.1 Linux 2.4

The Linux 2.4 scheduler is very simplistic. All runnable processes are kept in a single queue, regardless of their priority, and every time the scheduler needs to make a dispatch decision the entire run queue is searched for the process with the highest priority that has not yes finished its time quantum. In fact, the remaining time quantum *is* the priority of a process, so the scheduler looks for the process with the longest remaining time quantum. When no such process can be found — i.e. all runnable processes consumed their allocated time quantum — the scheduler replenishes the quanta of *all* processes in the system, including those that are sleeping or blocked on some device. Each process's allocation is the default time quantum plus half of its leftover allocation from the last epoch (if any). This calculation is at the core of the fair-share design as it gives higher priority to sleeping/blocked processes.

The kernel uses a single timer interrupt, driven at 100Hz rate, that is both in charge of billing the running process for the entire finished time tick, as well as executing expired timers.

The pseudo code shown in Fig. 3 is implemented in Linux using standard POSIX signals; The process first request a periodic signal every 10 milliseconds (single clock tick) using the POSIX *setitimer* system call, and waits for the next signal using the POSIX *pause* system call. This implementation enables the cheater process to avoid all clock ticks and billing in general. Since it is never billed, keeping half of its previous time quantum whenever the time quanta are allocated for all process in the system results in its time quantum being twice the normal size — the maximal possible time quantum — and therefore having the highest priority in the system. The result is that the cheater process appears to process monitoring applications such as *top* as utilizing 0% of the CPU.

3.2 Linux 2.6

Monopolizing CPU cycles is especially easy on Linux 2.6 because the scheduler goes out of its way to support interactive processes. Interactive processes are identified by the fact that they yield the CPU voluntarily. This includes processes that perform our cheat attack and sleep in each tick.

The Linux 2.6 scheduler uses a multilevel feedback queue, with lower numeric priorities yielding better process priorities [14]. Technically, Linux uses two queues: the active list and the expired list. When a process consumes its entire time quantum it is replenished and put in the expired queue. When no runnable processes are available on the active list, the lists are switched.

Process priorities are composed as a sum of the static value (nice value) and a dynamic value, which is based on the time the process spent in voluntary sleep. This mechanism is designed so as to prioritize interactive processes which tend to pace themselves by sleeping. The Linux scheduler gives a substantial weight to this interactive factor — processes considered interactive are not

moved to the expired list when their quantum expires, but instead are replenished and returned immediately to the active list.

Linux uses a single timer interrupt at a default rate of 250Hz, which both drives process timers and bills the running process for the entire finished tick.

These characteristics of the Linux kernel makes the implementation of the cheater process a fairly easy task. The pseudo code in Fig. 3 is implemented by requesting a POSIX signal every 4 milliseconds (250Hz) using the standard POSIX *setitimer* system call, and waiting till the next tick is performed by waiting for its accompanying signal using the POSIX *pause* system call. This implementation allows for all ticks to be avoided, both providing the cheater process with ample sleep time to gain an interactive priority, and to avoid being billed for its CPU usage as it always sleeps when the clock interrupt fires. This is also demonstrated by the UNIX *top* utility, which shows the cheater process as consuming 0% of the CPU cycles, as shown in Figure 7.

3.3 Windows XP

The Windows XP scheduler is implemented using a multilevel priority feedback queue, where lower numeric priorities yield lower thread priorities [20]. The event timer and scheduler timer are driven by a single interrupt firing at constant intervals typically around 10ms for uniprocessors and around 15ms for multiprocessors. The default time quantum on a workstation is 2 timer ticks, with the quantum itself having a value of 6, implying that every clock tick decrements the quantum by 3. The reason for this is to allow *partial* accounting when a thread wakes up after sleeping or being blocked, so as to prevent situations in which a thread is never billed because it always sleeps when the billing occurs. Due to the partial accounting, such a thread is only supposed to get 3 times as much runtime as other threads of equal priority. A thread that finishes its time quantum is put at the back of its priority's queue.

Dispatching decisions are made based on the threads' dynamic priorities, which are the sum of the static priority and a priority boost. The default user static priority is 8. The scheduler does not reduce the priority of CPU hogs, but rather uses five heuristics that can give threads a positive priority boost, which is subsequently reduced by 1 at the end of each time quantum. These five heuristics try to favor I/O-bound threads or threads that experience CPU starvation. One of the heuristics favors threads waiting for system events.

The cheating pseudo code depicted in Fig. 3 is implemented in Windows XP using Win32 messages. The cheater thread requests a periodic timer message every 10 milliseconds (on a system driven by a 10 milliseconds interrupt), and uses the Win32 GetMessage to block until a timer message is dispatched — immediately after the system clock interrupt fires. As soon as the message is received by the thread, it computes when the next tick is due, and goes on to the computation phase until the next tick approaches.

Because the thread is never the running thread when the tick fires it is never billed for the time it consumed. Moreover, since the thread blocks waiting for a system event it even gets a priority boost of 2, yielding higher dynamic priority than other normal threads in the system. This allows it to evade the protection that was supposed to be provided by the partial accounting: as the other threads stay at the original priority, they do not run until the cheater thread blocks itself.

When using Windows' *TaskManager* to view the CPU consumption rates of the various threads, the cheater thread is shown to consume the accurate percentage of cycles it actually does, indicating that Windows does save accurate runtime statistics as part of its *Thread Performance Counters*. This information however is not used by the scheduler, which instead uses sampling to bill processes — a dissonance that opens the gate for the cheat attack.

3.4 Solaris 10

The Solaris 10 scheduler is also based on a multilevel priority feedback queue where lower numeric priorities represent lower thread priorities [15]. Its default scheduling class is the *timeshare* class which inhabits the lower 60 priorities in the system. This class is based on a table which computes a thread's new priority and time quantum based on its previous priority and the reason it is inserted into the queue — either because its time quantum expired or because it just woke up after blocking on some resource (a new thread inherits its parent's priority). The table is designed such that processes that consume their entire allocated time quantum receives lower priorities and longer time quanta, while threads that block or sleep receive higher priorities and shorter time quanta.

Solaris maintains a clock interrupt which is invoked every 10 milliseconds (100Hz rate). The interrupt handler is in charge of waking up sleeping threads whose timeout expired, as well as invoking the scheduler's tick callback function, which bills the current running process for the entire finished tick.

The pseudo code in Fig. 3 is implemented in Solaris

using standard POSIX constructs: the thread requests a periodic 10 milliseconds timer using the setitimer system call and blocks until the next tick using the pause system call. Calculating when the next timer is due is performed inside the *SIGALRM* signal handler called when each timer expires — an operation that immediately follows the operating system's periodic timer interrupt.

By avoiding being the running thread when the periodic interrupt timer fires the cheater thread is considered by the scheduler to be a chronic sleeper that never runs (since it is never billed for any tick), thus — according to the *timeshare* scheduler class — causing its priority to increase until it reaches the topmost priority available to *timeshare* processes.

Even though the scheduler never bills the cheater for the cycles it consumes, running the UNIX *top* utility still shows the cheater's accurate CPU consumption, as Solaris actually gathers cycle-accurate runtime statistics by saving the time a thread spends in each of the thread states (running, waiting to run, blocked, etc.) described in the well known thread state diagram [21]. This again demonstrate the dissonance caused by an operating system having accurate billing information but not revealing it to the scheduler.

3.5 FreeBSD

Compromising FreeBSD is especially meaningful because, as far as we know, this is the only systems that was designed to prevent potential cheat attacks [16]. This awareness is expressed in the design of the kernel timing mechanism which uses two timers with relatively prime frequencies — one for interrupts in charge of driving regular kernel timing events (with frequency *HZ*), and one for gathering scheduling statistics (with frequency *STATHZ*). For example, the test system that was available to us runs FreeBSD 6.1 with a HZ frequency of 1000Hz and STATHZ frequency of ˜133Hz (the common default values are lower). The running thread's time quantum is decremented by 1 every STATHZ tick.

FreeBSD has two optional schedulers: the default *4BSD* scheduler which is derived from the classic UNIX scheduler [3], and the alternative *ULE* scheduler aimed at favoring interactive applications [19]. Both schedulers use a multilevel priority feedback queue where lower numeric priorities yield *higher* thread priorities, and differ in the manner in which they dispense priorities.

The *4BSD* scheduler uses a static priority (nice value) adjusted with a dynamic component based on the thread's CPU consumption estimator. This estimator accumulates linearly as the process runs (every STATHZ tick) and is exponentially decayed in a periodic manner, where the exponent depends on the system load. This scheme yields worsening priorities as threads consume CPU cycles, but threads that avoid the STATHZ tick and are not billed for their consumption retain their priority.

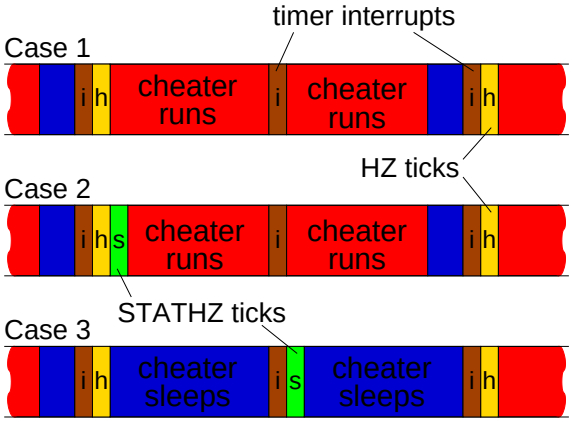

Figure 12: *The three possible alignments of the various FreeBSD clocks: no STATHZ tick between consecutive HZ ticks (case 1), STATHZ ticks falls on an even timer interrupt alongside a HZ tick (case 2), and a STATHZ tick falling on an odd clock interrupt between HZ ticks (case 3).*

The *ULE* scheduler uses two run queues — *current*, which contains waiting threads that have not consumed their entire time quantum, and *next* which contains threads with an expired time quantum (similar to the active and expired lists in Linux 2.6). Whenever the *current* queue empties the scheduler switches between the two queues. To better support interactive applications it uses an *interactive factor* for prioritization. The interactive factor is based on the ratio between the time a process voluntarily slept and the time it ran. This factor is added to the priority calculation, effectively reducing the priority of threads that sleep less compared to those that sleep more. Furthermore, if a thread's interactive factor is below a certain value (lower factor values indicate the thread sleeps more) it is considered interactive, in which case even if its time quantum expires it is inserted into the *current* run queue again, and is immediately available for re-scheduling (again as in Linux 2.6). Based on the predefined kernel parameters for calculating the interactive factor, a thread is considered interactive if it sleep more than ˜63% of the time. The interactive factor relies on a thread's runtime and sleep time that are measured in HZ ticks (rather than STATHZ ticks), and are both exponentially decayed. If both are 0 (when ticks are avoided)

the interactive factor is set to 0. However, if STATHZ ticks are avoided as well the thread's quantum is never decremented so it is always ready to run in the *current* queue.

Since most architectures support only a single timing source, both the HZ and STATHZ timers are derived from a single timer interrupt driven at a higher frequency — in our case $2 \times HZ = 2000Hz$. During each timer interrupt the handler checks whether the HZ and/or STATHZ tick handlers should be called — the first is called every 2 interrupts, whereas the second is called every 15–16 interrupts. The possible alignments of the two ticks are shown in Figure 12. This figure shows that the HZ ticks are executed on each even timer interrupt, usually as a single tick (case 1). Occasionally both the HZ and STATHZ ticks align on a an even timer interrupt (case 2), and sometimes STATHZ is executed on an odd timer interrupt (case 3). By avoiding HZ ticks we automatically also avoid STATHZ ticks in case 2. But to completely avoid being billed for the CPU time it consumes, the cheater thread must identify when case 3 occurs and sleep between the two consecutive HZ tick surrounding the STATHZ tick.

The kernel's timer interrupt handler calculates when to call the HZ and STATHZ ticks in a manner which realigns the two clocks every second. Based on this, we slightly modified the pseudo code described in Fig. 3 to pre-compute a $2 \times HZ$ sized STATHZ bitmap, in which each bit corresponds to a specific timer interrupt in a one second interval, and setting the bit for those interrupts which drive a STATHZ tick. Furthermore, the code reads the number of timer interrupts that occurred since the system was started, data that is available using a special *sysctl* call. The cheater thread then requests the system for signals at a constant HZ rate using the standard POSIX *setitimer* system call (and waits for each signal using the POSIX *pause* system call). The *SIGALRM* signal handler in turn accesses the STATHZ bitmap with a value of $(interrupt\_index + 1) \bmod (2 \times HZ)$ to check whether the *next* timer interrupt will trigger a STATHZ tick (besides calculating when the next HZ tick is due). This mechanism allows the cheater thread to identify case 3 described in Figure 12, and simply sleep until the next HZ tick fires and a *SIGALRM* is delivered — thus never getting billed for the CPU time it consumes.

3.6 Mac OS X

Mac OS X was the only system we were not able to cheat (the version we checked is Mac OS X Darwin 8.6.0, *xnu* version 792.6.70). The system uses the multi-level priority feedback queue, with lower numeric priorities yielding lower thread priority [2]. Threads in the default timeshare class have their dynamic priority calculated based on their recent CPU consumption. Threads' priority is reduced by a factor which is a linear function of their cpu consumption history, which in turn is exponentially decayed (this history include the CPU consumption in the preceding ˜3 seconds). The priority is recalculated whenever a thread is scheduled out, and the thread is put at the back of the new priority's run queue. A periodic system thread recomputes priorities of all threads waiting for a processor, decaying their CPU consumption history thus elevating their priorities.

The main reason we could not "cheat" the Max OS X scheduler is that while its priority mechanism resembles that of other systems, Max OS X accounts for threads' CPU usage in an almost perfect manner. Whenever a processor traps from user-level to system-level the running thread is billed for the time since the last transition at nanosecond resolution (using the underlying architecture's cycle counter facilities). Moreover, scheduling decisions actually use these accurate CPU consumption patterns — based on a much finer time granularity than we could exploit.

Besides avoiding CPU consumption sampling, Max OS X uses one-shot timers to drive its timing mechanism. Each processor maintains a queue of future events, including process timer expirations and the current thread's time quantum, with each expired event causing the processor to reprogram the hardware's timing mechanism to trigger. Thus there are no periodic timer events that can be anticipated in advance.

3.7 Cheater Results on Major Systems

To test the effect of the cheater on system throughput we constructed a simple cycle burning application that simply increments an integer for a predefined period, and then prints the integer value reached. The program was first run alone on each system to get a reference value, and was then executed alongside the cheater to examine the cheater's effect on the counting application's throughput. These measurements were executed on the following versions of the operating systems reviewed: Linux 2.4.32, Linux 2.6.16, Window XP SP2, Solaris 10 (SunOS 5.11 for i386), and FreeBSD 6.1.

Figure 13 shows the effect of running the load application for a 10 seconds duration with the cheater process set to steal 80% of the CPU cycles running in the background. It is clear that the cheater is pretty accurate in the percentage of cycles it consumes, leaving the "le-

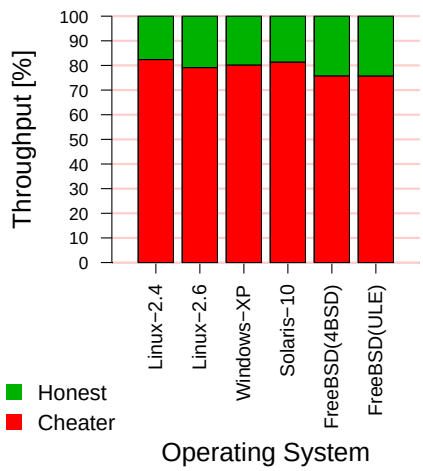

Figure 13: *Throughput of the cheater thread vs. simple CPU load on major operating systems.*

gal" load process with only ˜20 percent of the machine cycles. The biggest deviation occurs on FreeBSD systems, where the "legal" process gets almost 25% of the CPU. This is due to the need to evade FreeBSD's multiple clocks. Since the cheater can only use the HZ clock to evade the STATHZ clock it occasionally has to sleep for two full clock ticks, as depicted in case 3 of Figure 12.

4 Protecting Against the "Cheat" Attack

As explained above, the cheat attack exploits two common operating system mechanisms: accounting based on sampling, and prioritization of processes that use little CPU power. Here we present solutions for each of these in turn, and then address other potential solutions. Our solution turns out to be very similar to how the Mac OS X system works.

4.1 Accurate Billing

Billing based on periodic timers is problematic due to aliasing problems, as demonstrated by the X server and Xine process which escaped being billed without any such intent [7]. As noted in the previous section, some systems nowadays use clocks with different resolutions for timers and for profiling. This makes a cheat attack harder, but does not prevent it, as explained above. A better solution is to use accurate billing based on the underlying hardware cycle counter. And of course, the obtained information has to be used by the scheduler. Amazingly, several systems (including Windows XP and Solaris) already have accurate billing information, but fail to use it in the scheduler.

process	*default resource use*	*nice +19 resource use*
cheat 80%	80.47%	46.13%
honest	19.53%	53.87%

Table 1: *Computational resources obtained by a cheat process competing with an honest process on a Linux 2.6 system with the perfect billing patch.*

We implemented a patch for perfect billing in the Linux 2.6 system. The patch is only a few dozen lines long. The main modification is to replace the field time_slice in the process structure with two fields: ns_time_slice, which measures the allocated time slice in nano-seconds instead of in jiffies (the Linux term for clock ticks), and ns_last_update, which records when ns_time_slice was last updated. The value of ns_time_slice is decremented by the elapsed time since ns_last_update in two places: on each clock tick (to account for the last process run during the tick, as in the original system, but with the improvement of only accounting for the time really used by this process), and from the scheduler, to account for processes that used earlier parts of a tick. The rest of the kernel is unmodified, and still works in a resolution of jiffies. This was done by replacing accesses to time_slice by an inlined function that uses ns_time_slice to calculate and return the corresponding value in jiffies. Due to the use of 64-bit values rather than 32-bit values for keeping track of time, the overhead of a clock tick is slightly higher with this patch: 1636 ± 182 cycles on average instead of 1557 ± 159 cycles without the patch (the $\pm$ denotes the standard deviation). But the overhead of a scheduler tick is slightly reduced, from 8439 ± 9323 to 6971 ± 9506. This may have to do with the fact that after the patch, the cheater runs much less, and therefore there are a lot less timers to handle in the tick handler, and therefore on average, it's shorter. As the high standard deviations indicate, the distribution of scheduler ticks has a long tail, with maximal values around 150,000.

Somewhat surprisingly, using this patch didn't solve the cheat problem: a cheat process that was trying to obtain 80% of the cycles still managed to get them, despite the fact that the scheduler now had full information about this (Tab. 1 and Fig. 14). It turns out that this happened because of the extra support for interactive processes introduced in the 2.6 kernel.

The 2.6 kernel identifies processes that block voluntarily as interactive, provided their nice level is not too

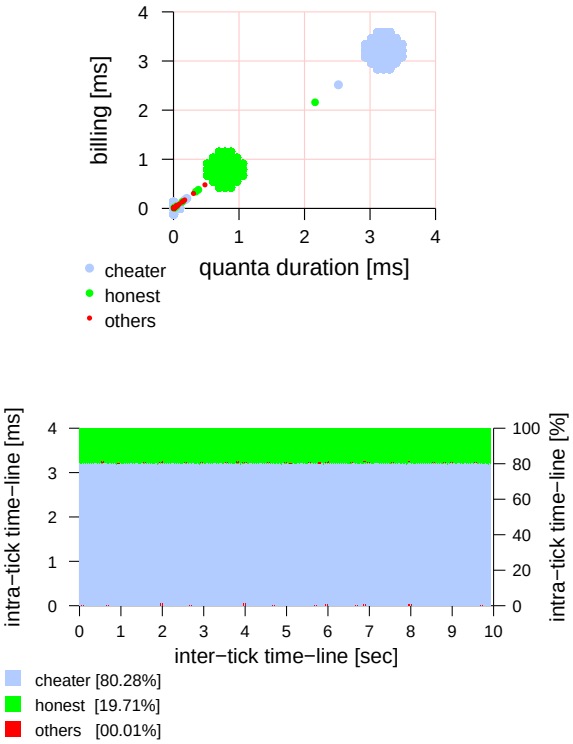

Figure 14: *In Linux 2.6, cheating is still possible even with perfect billing. Compare with Figs. 5 and 6.*

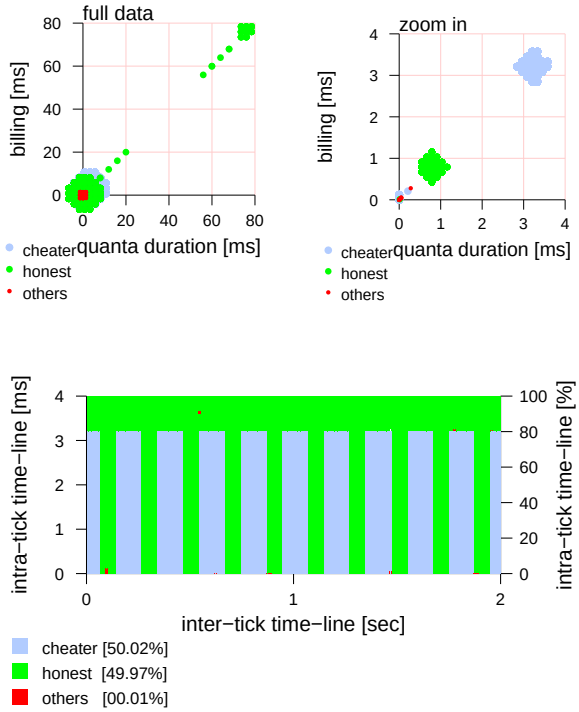

Figure 15: *Results with the fixed Linux scheduler, in which expired processes are not re-inserted into the active list. Compare with Fig. 14.*

high. When such an interactive process exhausts its allocation and should be removed from the active list and placed in the expired list, it is instead allowed to remain on the active list. Our cheat process, which blocks before each clock tick, fits the definition and therefore receives this preferential treatment. In fact, our process is even over-sophisticated: any process that displays enough interactive behavior would also be able to monopolize the system. In effect, the scheduler is over-riding the original allocations it gave the different processes, and allowing the interactive ones to starve the others. This is a good demonstration of the two sides of the solution: it is not enough to have good information — it is also necessary to use it effectively.

To show that this is indeed what is happening, we commented out the line in the scheduler that re-inserts expired interactive processes to the active list (a similar effect can be achieved by setting the nice level of both processes to +19, because with this nice rating processes are not considered interactive, so the "optimization" that allows interactive processes to starve the others is effectively disabled). We then re-ran the experiment. And indeed the cheating process failed to obtain the extra resources it was trying to get (Tab. 1).

The details of what happens in the corrected system are shown in Fig. 15. The timeline is effectively divided into segments of about 200ms, corresponding to renewed allocations to the two competing processes. In each such segment, the two processes share the CPU equitably. However, this is not the case in each tick. Initially, the cheat process has a higher priority, because it blocks a lot. It is therefore the first to run on each tick, until it blocks in anticipation of the next tick. But when its allocation runs out, it is not allowed to continue to run and starve the honest process. Instead, the honest process is allowed to catch up, until its allocation also runs out, and the whole thing is repeated. When catching up, the honest process is largely undisturbed, to it runs for many ticks, totaling up to 76ms.

4.2 Prioritizing Interactive Processes

Implementing accurate billing is relatively straightforward, as we demonstrated in the previous subsection; and indeed Solaris and Windows XP also keep track of resource usage accurately. However, they fail to use this information in the scheduler to prevent the cheat attack.

Part of the problem may be related to the issue of supporting interactive work. Identifying interactive processes nowadays is not as easy as it used to be. Modern interactive application such as multimedia players and role-playing games may be very CPU intensive, and have the same CPU usage profiles as background tasks like a kernel make or even a compute intensive application [6]. Thus the harder part of the solution is how to identify and prioritize *real* interactive processes, but at the same time avoid giving resources to cheating processes.

A possible solution is to track user interactions directly. This requires a collaboration between the windowing system and the scheduler. The window manager (the X server in a Unix/Linux environment) needs to collect data about the level of interactive input and output activity by each process, and pass it to the scheduler. The system also needs to track the interactions among processes, to see if one mediates user I/O for another. This data is then used by the scheduler to obtain a better prioritization of interactive processes [8].

4.3 Other Potential Solutions

Several other solutions may be used to prevent cheating applications from obtaining excessive CPU resources. Here we details some of them, and explain why they are inferior to the accurate billing we suggested above.

Perhaps the simplest solution is to charge for CPU usage up-front, when a process is scheduled to run, rather than relying on sampling of the running process. However, this will over-charge interactive processes that in fact do not use much CPU time.

Another potential solution is to use two clocks, but have the billing clock operate at a finer resolution than the timer clock. This leads to two problems. One is that it requires a very high tick rate, which leads to excessive overhead. The other is that it does not completely eliminate the cheat attack. An attack is still possible using an extension of the cheat server approach described in Section 2.3. The extension is that the server is used not only to stop execution, but also to start it.

A variant of this is to randomize the clock in order to make it impossible for an attacker to predict when ticks will occur [13]. This might work, but at the cost of overheads and complexity. However, note that true randomness is hard to come by, and a system's random number generator could be reverse-engineered in order to beat the randomness [10].

A third possible approach is to block access to the cycle counter from user level (this is possible at least on the Intel machines). This again suffers from two problems. First, it withdraws a service that may have good and legitimate uses. Second, it too does not eliminate the cheat attack, only make it somewhat less accurate. A cheat application can still be written without access to a cycle counter by finding approximately how much application work can be done between ticks, and using this directly to decide when to stop running.

5 Conclusions

The "cheat" attack is an extremely simple way to exploit computer systems. It allows an unprivileged user-level application to seize whatever fraction of the CPU cycles it wants. We have demonstrated this attack on all major general purpose operating systems in common use today, except for Mac OS X. In some of the systems, the attacking process is also "invisible" on system monitors, meaning that they assign the CPU time it uses to some other innocent process. This naturally makes the attack much harder to detect.

The attack is based on two features of the attacked systems: that CPU accounting and timer servicing are both tied to the same periodic hardware clock interrupts, and that the scheduler favors processes that exhibit low CPU usage. The accounting problem can be fixed easily, and Windows XP, Solaris, and Mac OS X indeed already perform accurate accounting. Fixing the scheduler is harder. In attempting to prioritize interactive processes, Windows raises the priority of threads that have slept, thus undermining the partial accounting that was supposed to protect it from the cheat attack. Linux 2.6 and the ULE scheduler for FreeBSD explicitly ignore accounting data, and allow processees identified as interactive to monopolize the CPU. Thus it seems that the scheduling algorithm is more critical the we're used to think, both in design and implementation. As we showed in [6], it seems that using CPU usage to identify interactive processes is doomed to failure. The alternative is to explicitly track user interactions [8].

Our patch to fix the Linux 2.6 scheduler doesn't handle the identification of interactive processes, but just fixes the loophole allowing the cheat attack. It is also based on accurate billing using the hardware cycle counter. As a byproduct, this removes one of the many functions of the clock interrupt handler. This can be considered a first

step in the direction of a completely tick-less operating system, that will solve various other problems that result from using ticks as a basic design feature [23].

References

[1] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer, "*SETIhome: an experiment in public-resource computing*". *Comm. ACM* **45(11)**, pp. 56–61, Nov 2002.

[2] "*Apple Mac OS X/Darwin kernel source code*". http://www.opensource.apple.com/darwinsource.

[3] M. J. Bach, *The Design of the UNIX Operating System*. Prentice-Hall, 1986.

[4] E. W. Dijkstra, "*The structure of the "THE"-multiprogramming system*". *Comm. ACM* **11(5)**, pp. 341–346, May 1968.

[5] Y. Etsion and D. G. Feitelson, *Time Stamp Counters Library - Measurements with Nano Seconds Resolution*. Technical Report 2000-36, Inst. Computer Science, The Hebrew University of Jerusalem, Aug 2000.

[6] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Desktop scheduling: how can we know what the user wants?*". In 14th *Network & Operating Syst. Support for Digital Audio & Video*, pp. 110–115, Jun 2004.

[7] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Effects of clock resolution on the scheduling of interactive and soft real-time processes*". In *SIGMETRICS Conf. Measurement & Modeling of Comput. Syst.*, pp. 172–183, Jun 2003.

[8] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. Multimedia Comput., Commun. & Appl.*, 2006. to appear.

[9] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. Feitelson, *Fine Grained Kernel Logging with KLogger: Experience and Insights*. Technical Report 2005–35, The Hebrew University of Jerusalem, Jun 2005.

[10] Z. Gutterman and D. Malkhi, "*Hold your sessions: an attack on java servlet session-id generation*". In *Topics in Cryptology — CT-RSA 2005*, A. Menezes (ed.), pp. 44–57, Springer-Verlag, Feb 2005. Lect. Notes Comput. Sci. vol. 3376.

[11] IBM Corp., *PowerPC Microprocessor Family: The Programming Environments Manual for 64-bit Microprocessors*. Jul 2005. Ver. 3.0.

[12] Intel Corp., *IA-32 Intel Achitecture Software Developr's Manual, Vol. 1: Basic Architecture*.

[13] J. Liedtke, "*A short note on cheap fine-grained time measurement*". *Operating Systems Review (OSR)* **30(2)**, pp. 92–94, April 1996.

[14] R. Love, *Linux Kernel Development*. Novell Press, 2nd ed., 2005.

[15] J. Mauro and R. McDougall, *Solaris Internals*. Prentice Hall, 2001.

[16] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*. Addison Wesley, 1996.

[17] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "*Inside the slammer worm*". *IEEE Security & Privacy* **1(4)**, pp. 33–38, Jul/Aug 2003.

[18] D. Moore, C. Shannon, and K. claffy, "*Code-red: a case study on the spread and victims of an internet worm*". In *Proc. ACM Workshop on Internet measurment*, pp. 273–284, ACM Press, New York, NY, USA, 2002.

[19] J. Roberson, "*ULE: a modern scheduler for FreeBSD*". In *BSDCon*, pp. 17–28, The Usenix Association, Berkeley, CA, USA, Sep 2003.

[20] M. E. Russinovich and D. A. Solomon, *Microsoft Windows Internals*. Microsoft Press, 4th ed., Dec 2004.

[21] A. Silberschatz, P. B. Galvin, and G. Gagne, *Operating System Concepts*. John Wiley & Sons, 7th ed., 2005.

[22] S. Staniford, V. Paxson, and N. Weaver, "*How to own the Internet in your spare time*". In 11th *USENIX Security Symp.*, pp. 149–167, Aug 2002.

[23] D. Tsafrir, Y. Etsion, and D. G. Feitelson, *General-Purpose Timing: The Failure of Periodic Timers*. Technical Report 2005-6, School of Computer Science and Engineering, The Hebrew University of Jerusalem, Feb 2005.

[24] "*Timeline of notable computer viruses and worms*". Wikipedia Entry. http://en.wikipedia.org.