# Secretly Monopolizing the CPU Without Superuser Privileges

Dan Tsafrir[◇†] Yoav Etsion[◇] Dror G. Feitelson[◇]

[◇]*School of Computer Science and Engineering*
*The Hebrew University*
*91904 Jerusalem, Israel*
{*dants, etsman, feit*}*@cs.huji.ac.il*

[†]*IBM T. J. Watson Research Center*
*P. O. Box 218*
*Yorktown Heights, NY 10598*
*dants@us.ibm.com*

## Abstract

We describe a "cheat" attack, allowing an ordinary process to hijack any desirable percentage of the CPU cycles without requiring superuser/administrator privileges. Moreover, the nature of the attack is such that, at least in some systems, listing the active processes will erroneously show the cheating process as not using any CPU resources: the "missing" cycles would either be attributed to some other process or not be reported at all (if the machine is otherwise idle). Thus, certain malicious operations generally believed to have required overcoming the hardships of obtaining root access and installing a rootkit, can actually be launched by non-privileged users in a straightforward manner, thereby making the job of a malicious adversary that much easier. We show that most major general-purpose operating systems are vulnerable to the cheat attack, due to a combination of how they account for CPU usage and how they use this information to prioritize competing processes. Furthermore, recent scheduler changes attempting to better support interactive workloads increase the vulnerability to the attack, and naive steps taken by certain systems to reduce the danger are easily circumvented. We show that the attack can nevertheless be defeated, and we demonstreate this by implementing a patch for Linux that eliminates the problem with negligible overhead.

## Prologue

Some of the ideas underlying the cheat attack were implemented by Tsutomu Shimomura circa 1980 at Princeton, but it seems there is no published or detailed essay on the topic, nor *any* mention of it on the web [54]. Related publications deal solely with the fact that general-purpose CPU accounting can be inaccurate, but never conceive this can be somehow maliciously exploited (see Section 2.3). Recent trends in mainstream schedulers render a discussion of the attack especially relevant.

## 1 Introduction

An *attacker* can be defined as one that aspires to perform actions "resulting [in the] violation of the explicit or implicit security policy of a system", which if successful, constitute a *breach* [31]. Under this definition, the said actions may be divided into two classes. One is of *hostile actions*, e.g. unlawful reading of sensitive data, spamming, lunching of DDoS attacks, etc. The other is of *concealment actions*. These are meant to prevent the hostile actions from being discovered, in an effort to prolong the duration in which the compromised machine can be used for hostile purposes. While not hostile, concealment actions fall under the above definitions of "attack" and "breach", as they are in violation of any reasonable security policy.

The "cheat" attack we describe embodies both a hostile and a concealment aspect. In a nutshell, the attack allows to implement a `cheat` utility such that invoking

```
cheat p prog
```

would run the program `prog` in such a way that it is allocated exactly $p$ percent of the CPU cycles. The hostile aspect is that $p$ can be arbitrarily big (e.g. 95%), but `prog` would still get that many cycles, regardless of the presence of competing applications and the fairness policy of the system. The concealment aspect is that `prog` would erroneously appear as consuming 0% CPU in monitoring tools like ps, top, xosview, etc. In other words, the cheat attack allows a program to (**1**) consume CPU cycles in a secretive manner, and (**2**) consume as many of these as it wants. This is similar to the common security breach scenario where an attacker manages to obtain superuser privileges on the compromised machine, and uses these privileges to engage in hostile activities and to conceal them. But in contrast to this common scenario, the cheat attack requires no special privileges. Rather, it can be launched by regular users, deeming this important line of defense (of obtaining root or superuser privileges) as irrelevant, and making the job of the attacker significantly easier.

Concealment actions are typically associated with *rootkits*, consisting of "a set of programs and code that allows a permanent or consistent, undetectable presence on a computer" [25]. After breaking into a computer and obtaining root access, the intruder installs a rootkit to maintain such access, hide traces of it, and exploit it. Thus, ordinarily, the ability to perform concealment actions (the rootkit) is the result of a hostile action (the break-in). In contrast, with the cheat attack, it is exactly the opposite: the concealment action (the ability to appear as consuming 0% CPU) is actually what makes it possible to perform the hostile action (of monopolizing the CPU regardless of the system's fairness policy). We therefore begin by introducing the OS mechanism that allows a non-privileged application to conceal the fact it is using the CPU.

## 1.1 Operating System Ticks

A general-purpose operating system (GPOS) typically maintains control by using periodic clock interrupts. This practice started at the 1960s [12] and has continued ever since, such that nowadays it is used by most contemporary GPOSs, including Linux, the BSD family, Solaris, AIX, HPUX, IRIX, and the Windows family. Roughly speaking, the way the mechanism works is that at boot-time the kernel sets a hardware clock to generate periodic interrupts at fixed intervals (every few milliseconds; anywhere between 1ms to 15ms, depending on the OS). The time instance at which the interrupt fires is called a *tick*, and the elapsed time between two consecutive ticks is called a *tick duration*. The interrupt invokes a kernel routine, called the *tick handler* that is responsible for various OS activities, of which the following are relevant for our purposes:

1. **Delivering timing services** and alarm signals. For example, a movie player that wants to wakeup on time to display the next frame, requests the OS (using a system call) to wake it up at the designated time. The kernel places this request in an internal data structure that is checked upon each tick. When the tick handler discovers there's an expired alarm, it wakes the associated player up. The player then displays the frame and the scenario is repeated until the movie ends.
2. **Accounting for CPU usage** by recording that the currently running process $S$ consumed CPU cycles during the last tick. Specifically, on every tick, $S$ is stopped, the tick-handler is started, and the kernel increments $S$'s CPU-consumption tick-counter within its internal data structure.
3. **Initiating involuntary preemption** and thereby implementing multitasking (interleaving of the CPU between several programs to create the illusion they execute concurrently). Specifically, after $S$ is billed for consuming CPU during the last tick, the tick handler checks whether $S$ has exhausted its "quantum", and if so, $S$ is preempted in favor of another process. Otherwise, it is resumed.

## 1.2 The Concealment Component

The fundamental vulnerability of the tick mechanism lies within the second item above: CPU billing is based on periodic sampling. Consequently, if $S$ can somehow manage to arrange things such that it always starts to run just after the clock tick, and always goes to sleep just before the next one, then $S$ will never be billed. One might naively expect this would not be a problem because applications cannot request timing services independent of OS ticks. Indeed, it is *technically impossible* for non-privileged applications to request the OS to deliver alarm signals in between ticks. Nevertheless, we will show that there are several ways to circumvent this difficulty.

To make things even worse, the cheat attack leads to *misaccounting*, where another process is billed for CPU time used by the cheating process. This happens because billing is done in tick units, and so whichever process happens to run while the tick takes place is billed for the *entire* tick duration, even if it only consumed a small fraction of it. As a result, even if the system administrators suspect something, they will suspect the wrong processes. If a cheating process is not visible through system monitoring tools, the only way to notice the attack is by its effect on throughput. The cheater can further disguise its tracks by moderating the amount of CPU it uses so as not to have too great an impact on system performance.

## 1.3 The Hostile Component

The most basic defense one has against malicious programs is knowing what's going on in the system. Thus, a situation in which a non-privileged application can conceal the fact it makes use of the CPU, constitutes a serious security problem in its own right. However, there is significantly more to cheat attacks than concealment, because CPU accounting is not conducted just for the sake of knowing what's going on. Rather, this information has a crucial impact on scheduling decisions.

As exemplified in Section 5, the traditional design principle underlying general-purpose scheduling (as opposed to research or special-purpose schemes) is the same: the more CPU cycles used by a process, the lower its priority becomes [15]. This *negative feedback* (running reduces priority to run more) ensures that (**1**) all processes get a fair share of the CPU, and that (**2**) processes that do not use the CPU very much — such as

I/O bound processes — enjoy a higher priority for those bursts in which they want it. In fact, the latter is largely what makes text editors responsive to our keystrokes in an overloaded system [14].

The practical meaning of this is that by consistently appearing to consume 0% CPU, an application gains a very high priority. As a consequence, when a cheating process wakes up and becomes runnable (following the scenario depicted in the previous subsection) it usually has a higher priority than that of the currently running process, which is therefore immediately preempted in favor of the cheater. Thus, as argued above, unprivileged concealment capabilities indeed allow an application to monopolize the CPU. However, surprisingly, this is not the whole story. It turns out that even without concealment capabilities it is still sometimes possible for an application to dominate the CPU without superuser privileges, as discussed next.

## 1.4 The Interactivity Component and the Spectrum of Vulnerability to Cheating

Not all GPOSs are vulnerable to cheat attacks to the same degree. To demonstrate, let us first compare between Linux-2.4 and Linux-2.6. One of the radical differences between the two is the scheduling subsystem, which has been redesigned from scratch and undergone a complete rewrite. A major design goal of the new scheduler was to improve users' experience by attempting to better identify and service interactive processes. In fact, the lead developer of this subsystem argued that "the improvement in the way interactive tasks are handled is actually the change that should be the most noticeable for ordinary users" [3]. Unfortunately, with this improvement also came increased vulnerability to cheat attacks.

In Linux-2.6, a process need not conceal the fact it is using the CPU in order to monopolize it. Instead, it can masquerade as being "interactive", a concept that is tied within Linux-2.6 to the number of times the process voluntarily sleeps [32]. Full details are given in Section 6, but in a nutshell, to our surprise, even after we introduced cycle-accurate CPU accounting to the Linux-2.6 kernel and made the cheating process fully "visible" at all times, the cheater still managed to monopolize the CPU. The reason turned out to be the cheater's many short voluntary sleep-periods while clock ticks take place (as specified in Section 1.2). This, along with Linux-2.6's aggressive preference of "interactive" processes yielded the new weakness.

In contrast, the interactivity weakness is not present in Linux-2.4, because priorities do not reflect any considerations that undermine the aforementioned negative feedback. Specifically, the time remaining until a process exhausts its allocated quantum also serves as its priority, and so the negative feedback is strictly enforced [36]. Indeed, having Linux-2.4 use accurate accounting information defeats the cheat attack.

The case of Linux 2.4 and 2.6 is not an isolated incident. It is analogous to the case of FreeBSD and the two schedulers it makes available to its users. The default "4BSD" scheduler [5] is vulnerable to cheat attacks due to the sampling nature of CPU accounting, like Linux-2.4. The newer "ULE" scheduler [42] (designated to replace 4BSD) attempts to improve the service provided to interactive processes, and likewise introduces an additional weakness that is similar to that of Linux-2.6. We conclude that there's a genuine (and much needed) intent to make GPOSs do a better job in adequately supporting newer workloads consisting of modern interactive applications such as movie players and games, but that this issue is quite subtle and prone to errors compromising the system (see Section 5.2 for further discussion of why this is the case).

Continuing to survey the OS spectrum, Solaris represents a different kind of vulnerability to cheat attacks. This OS maintains completely accurate CPU accounting (which is not based on sampling) and does *not* suffer from the interactivity weakness that is present in Linux-2.6 and FreeBSD/ULE. Surprisingly, despite this configuration, it is still vulnerable to the hostile component of cheating. The reason is that, while accurate information is maintained by the kernel, the scheduling subsystem does not make use of it (!). Instead, it utilizes the sampling-based information gathered by the periodic tick handler [35]. This would have been acceptable if all applications "played by the rules" (in which case periodic sampling works quite well), but such an assumption is of course not justified. The fact that the developers of the scheduling subsystems did not replace the sampled information with the accurate one, despite its availability, serves as a testament of their lack of awareness to the possibility of cheat attacks.

Similarly to Solaris, Windows XP maintains accurate accounting that is unused by the scheduler, which maintains its own sample-based statistics. But in contrast to Solaris, XP also suffers from the interactivity weakness of Linux 2.6 and ULE. Thus, utilizing the accurate information would have had virtually no effect.

From the seven OS/scheduler pairs we have examined, only Mac OS X was found to be immune from the cheat attack. The reason for this exception, however, is not a better design of the tick mechanism so as to avoid the attack. Rather, it is because Mac OS X uses a different timing mechanism altogether. Similarly to several realtime OSs, Mac OS X uses *one-shot timers* to drive its timing and alarm events [29, 47, 20]. These are hardware interrupts that are set to go off only for specific needs, rather than periodically. With this design, the OS maintains an

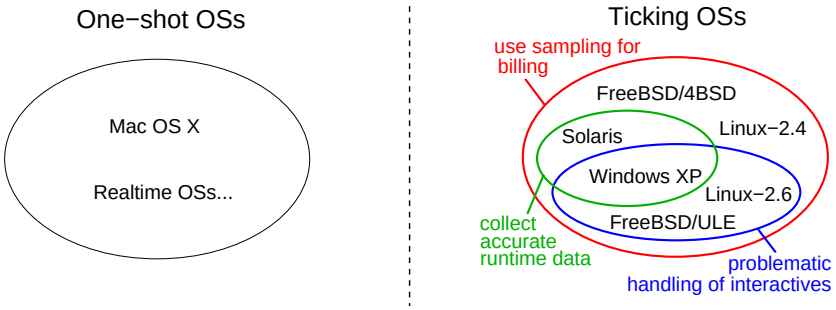



Figure 1: *Classification of major operating systems in terms of features relevant for the cheat attack. General-purpose OSs are typically tick-based, in which case they invariably use sampling and are therefore vulnerable to cheat attacks to various degrees.*

ascending list of outstanding timers and sets a one-shot event to fire only when it is time to process the event at the head of the list; when this occurs, the head is poped, and a new one-shot event is set according to the new head. This design is motivated by various benefits, such as reduced power consumption in mobile devices [40], better alarm resolution [15], and less OS "noise" [53]. However, it causes the time period between two consecutive timer events to be variable and unbounded. As a consequence, CPU-accounting based on sampling is no longer a viable option, and the Mac OS X immunity to cheat attacks is merely a side effect of this. Our findings regarding the spectrum of vulnerabilities to cheat attacks are summarized in Fig. 1.

While it is possible to rewrite a tick-based OS to be one-shot, this is a non-trivial task requiring a radical change in the kernel (e.g. the Linux-2.6.16 kernel source tree contains 8,997 occurrences of the tick frequency `HZ` macro, spanning 3,199 files). Worse, ticks have been around for so long, that some user code came to directly rely on them [52]. Luckily, eliminating the threat of cheat attacks does not necessitate a radical change: there exists a much simpler solution (Section 6). Regardless, the root cause of the problem is not implementation difficulties, but rather, lack of awareness.

### 1.5 Roadmap

This paper is structured as follows. Section 2 places the cheat attack within the related context and discusses the potential exploits. Section 3 describes in detail how to implement a cheating process and experimentally evaluates this design. Section 4 further shows how to apply the cheating technique to arbitrary applications, turning them into "cheaters" without changing their source code. Section 5 provides more details on contemporary schedulers and highlights their weaknesses in relation to the cheat attack on an individual basis. Section 6 describes and evaluates our solution to the problem, and Section 7 concludes.

## 2 Potential Exploits and Related Work

### 2.1 The Privileges-Conflict Axis

The conflict between attackers and defenders often revolves around privileges of using resources, notably network, storage, and the CPU. The most aggressive and general manifestation of this conflict is attackers that aspire to have all privileges and avoid all restrictions by obtaining root/administrator access. Once obtained, attackers can make use of all the resources of the compromised machine in an uncontrolled manner. Furthermore, using rootkits, they can do so secretly in order to avoid detection and lengthen the period in which the resources can be exploited. Initially, rootkits simply replaced various system programs, such as `netstat` to conceal network activity, `ls` to conceal files, and `ps`/`top` to conceal processes and CPU usage [55]. But later rootkits migrated into the kernel [9, 46] and underneath it [27], reflecting the rapid escalation of the concealment/detection battle.

At the other end of the privileges conflict one can find attacks that are more subtle and limited in nature. For example, in order to take control over a single JVM instance running on a machine to which an attacker has no physical access, Govindavajhala and Appel suggest the attacker should "convince it [the machine] to run the [Java] program and then wait for a cosmic ray (or other natural source) to induce a memory error"; they then show that "a single bit error in the Java program's data space can be exploited to execute arbitrary code with a probability of about 70%" within the JVM instance [21]. When successful, this would provide the attacker with the privileges of the user that spawned the JVM.

When positioning the general vs. limited attacks at opposite ends of the privileges-conflict "axis", the cheat attack is located somewhere in between. It is certainly not as powerful as having root access and a rootkit, e.g. the attacker cannot manipulate and hide network activity or file usage. On the other hand, the attack is not limited to only one user application, written in a specific language,

on the condition of a low probability event such as a cosmic ray flipping an appropriate bit. Instead, at its fullest, the cheat attack offers non-privileged users one generic functionality of a rootkit: A ubiquitous way to control, manipulate, and exploit one computer resource — CPU cycles — in a fairly secretive manner. In this respect, cheating is analogous to attacks like the one suggested by Borisov et al. that have shown how to circumvent the restrictions imposed by file permissions in a fairly robust way [8]. As with cheating, non-privileged users are offered a generic functionality of rootkits, only this time concerning files. An important difference, however, is that Borisov's attack necessitates the presence of a root setuid program that uses the access/open idiom (a widely discouraged practice [11][1]), whereas our attack has no requirements but running under a ticking OS.

## 2.2 Denying or Using the Hijacked Cycles

Cheating can obviously be used for launching DoS attacks. Since attackers can hijack any amount of CPU cycles, they can run a program that uselessly consumes e.g. 25%, 50%, or 75% of each tick's cycles, depending on the extent to which they want to degrade the effective throughput of the system; and with concealment capabilities, users may feel that things work slower, but would be unable to say why. This is similar to "shrew" and "RoQ" (Reduction of Quality) attacks that take advantage of the fact that TCP interprets packet loss as an indication of congestion and halves a connection's transmission rate in response. With well-timed low-rate DoS traffic patterns, these attacks can throttle TCP flows to a small fraction of their ideal rate while eluding detection [28, 23, 50].

Another related concept is "parasitic computing", with which one machine forces another to solve a piece of a complex computational problem merely by sending to it malformed IP packets and observing the response [6]. Likewise, instead of just denying the hijacked cycles from other applications, a cheating process can leverage them to engage in actual computation (but in contrast, it can do so effectively, whereas parasitic computing is extremely inefficient). Indeed, Section 4 demonstrates how we secretly monopolized an entire departmental *shared* cluster for our own computational needs, without "doing anything wrong".

A serious exploit would occur if a cheat application was spread using a computer virus or worm. 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 DDoS attacks or spam email [34]. In many cases these viruses and worms were found and uprooted because of their very success, as the load they place on the Internet become unignorable [38]. But Staniford et al. described a "surreptitious" attack by which a worm that requires no special privileges can spread in a much harder to detect contagion fashion, without exhibiting peculiar communication pattens, potentially infecting upwards of 10,000,000 hosts [49]. Combining such worms with our cheat attack can be used to create a concealed ad-hoc supercomputer and run a computational payload on massive resources in minimal time, harvesting a huge infrastructure similar to that amassed by projects like SETI@home [2]. 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. While this can be done with real rootkits, the fact it can also potentially be done without ever requiring superuser privileges on the subverted machines is further alarming. Indeed, with methods like Borisov's (circumvent file permissions [8]), Staniford's (networked undetected contagion [49]), and ours, one can envision a kind of "rootkit without root privileges".

## 2.3 The Novelty of Cheating

While the cheat attack is simple, to our knowledge, there is no published record of it, nor any mention of it on the web. Related publications point out that general-purpose CPU accounting might be inaccurate, but never raise the possibility that this can be maliciously exploited. Our first encounter with the attack was, interestingly, when it occurred by chance. While investigating the effect of different tick frequencies [15], we observed that an X server servicing a Xine movie player was only billed for 2% of the cycles it actually consumed, a result of (**1**) X starting to run just after a tick (following Xine's repetitive alarm signals to display each subsequent frame, which are delivered by the tick handler), and (**2**) X finishing the work within 0.8 of a tick duration. This pathology in fact outlined the cheat attack principles. But at the time, we did not realize that this can be maliciously done on purpose.

We were not alone: There have been others that were aware of the accounting problem, but failed to realize the consequences. Liedtke argued that the system/user time-used statistics, as e.g. provided by the getrusage system call, might be inaccurate "when short active intervals are timer-scheduled, i.e. start always directly after a clock interrupt and terminate before the next one" [30] (exactly

[1] The access system call was designed to be used by setuid root programs to check whether the invoking user has appropriate permissions, before opening a respective file. This induces a time-of-check-to-time-of-use (TOCTTOU) race condition whereby an adversary can make a name refer to a different file after the access and before the open. Thus, its manual page states that "the access system call is a potential security hole due to race conditions and should never be used" [1].

describing the behavior we observed, but stopping short from recognizing this can be exploited).

The designers of the FreeBSD kernel were also aware this might occur, contending that "since processes tend to synchronize to 'tick', the statistics clock needs to be independent to ensure that CPU utilization is correctly accounted" [26]. Indeed, FreeBSD performs the billing activity (second item in Section 1.1) independently of the other tick activities (notably timing), at different times and in a different frequency. But while this design alleviates some of the concerns raised by Liedtke [30] and largely eliminates the behavior we observed [15], it is nonetheless helpless against a cheat attack that factors this design in (Section 5) and only highlights the lack of awareness to the possibility of systematic cheating.

Solaris designers noticed that "CPU usage measurements aren't as accurate as you may think ... especially at low usage levels", namely, a process that consumes little CPU "could be sneaking a bite of CPU time whenever the clock interrupt isn't looking" and thus "appear to use 1% of the system but in fact use 5%" [10]. The billing error was shown to match the inverse of the CPU utilization (which is obviously not the case when cheating, as CPU utilization and the billing error are in fact equal).

Windows XP employs a "partial decay" mechanism, proclaiming that without it "it would be possible for threads never to have their quantums reduced; for example, a thread ran, entered a wait state, ran again, and entered another wait state but was never the currently running thread when the clock interval timer fired" [44]. Like in the FreeBSD case, partial decay is useless against a cheat attack (Section 5), but in contrast, it doesn't even need to be specifically addressed, reemphasizing the elusiveness of the problem.

We contend, however, that all the above can be considered as anecdotal evidence of the absence of awareness to cheat attacks, considering the bottom line, which is that *all widely used ticking operating systems are susceptible to the attack, and have been that way for years.*[2]

# 3 Implementation and Evaluation

As outlined above, the cheat attack exploits the combination of two operating system mechanisms: periodic sampling to account for CPU usage, and prioritization of processes that use less of the CPU. The idea is to avoid the accounting and then enjoy the resulting high priority. We next detail how the former is achieved.

[2]We conceived the attack a few years after [15], as a result of a dispute between PhD students regarding who gets to use the departmental compute clusters for simulations before some approaching deadlines. We eventually did not exercise the attack to resolve the dispute, except for the experiment described in Section 4.1, which was properly authorized by the system personnel.

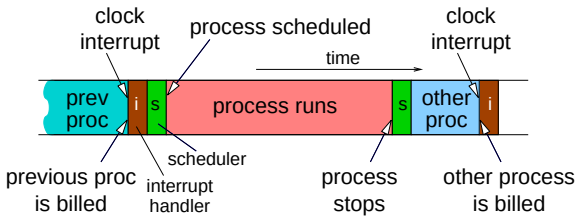



Figure 2: *The cheat attack is based on a scenario where a process starts running immediately after one clock tick, but stops before the next tick, so as not to be billed.*

## 3.1 Using the CPU Without Being Billed

When a tick (= periodic hardware clock interrupt) occurs, the entire interval since the previous tick is billed to the application that ran just before the current tick occurred. This mechanism usually provides reasonably accurate billing, despite the coarse tick granularity of a few milliseconds and the fact that nowadays the typical quanta is much shorter, for many applications [15].[3] This is a result of the probabilistic nature of the sampling: Since a large portion of the quanta are shorter than one clock tick, and the scheduler can only count in complete tick units, many of the quanta are not billed at all. But when a short quantum does happen to include a clock interrupt, the associated application is overbilled and charged a full tick. Hence, on average, these two effects tend to cancel out, because the probability that a quantum includes a tick is proportional to its duration.

Fig. 2 outlines how this rationale is circumvented. The depicted scenario has two components: (**1**) start running after a given billing sample, and (**2**) stop before the next. Implementing the first component is relatively easy, as both the billing and the firing of pending alarm timers are done upon a tick (first and second items in Section 1.1; handling the situation where the two items are independent, as in FreeBSD, is deferred to Section 5). Consequently, 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 (e.g. zero nanoseconds) will wake a process up immediately after the very next tick. If in addition it will have high priority, as is the case when the OS believes it is consistently sleeping, it will also start to run.

The harder part is to stop running before the next tick, when the next billing sample occurs. This may happen by chance as described above in relation to Xine and X. The question is how to do this on purpose. Since the OS does not provide intra-tick timing services, the process needs some sort of a finer-grained alternative timing

[3]In this context, quantum is defined to be the duration between the time an application was allocated the CPU and the time in which it relinquished the CPU, either voluntary or due to preemption.

```
inline cycle_t get_cycles()
{
    cycle_t ret;
    asm volatile("rdtsc" : "=A" (ret));
    return ret;
}


cycle_t cycles_per_tick()
{
    nanosleep(&zero,0); // sync with tick
    cycle_t start = get_cycles();

    for(int i=0 ; i<1000 ; i++)
        nanosleep(&zero,0);

    return (get_cycles() - start)/1000;
}
```

```
void cheat_attack( double fraction )
{
    cycle_t work, tick_start, now;

    work = fraction * cycles_per_tick();

    nanosleep(&zero,0); // sync with tick
    tick_start = get_cycles();

    while( 1 ) {
        now = get_cycles();
        if( now - tick_start >= work ) {
            nanosleep(&zero,0); // avoid bill
            tick_start = get_cycles();
        }
        // do some short work here...
    }
}
```

Figure 3: *The complete code for the cheater process (cycle_t is typedef-ed to be an unsigned 64-bit integer).*

mechanism. This can be constructed with the help of the *cycle counter*, available in all contemporary architectures. The counter is readable from user level using an appropriate assembly instruction, as in the get_cycles function (Fig. 3, top/left) for the Pentium architecture.

The next step is figuring out the interval between each two consecutive clock ticks, in cycles. This can be done by a routine such as cycles_per_tick (Fig. 3 bottom/left), correctly assuming a zero sleep would wake it up at the next clock interrupt, and averaging the duration of a thousand ticks. While this was sufficient for our purposes, a more precise method would be to tabulate all thousand timestamps individually, calculate the intervals between them, and exclude outliers that indicate some activity interfered with the measurement. Alternatively, the data can be deduced from various OS-specific information sources, e.g. by observing Linux's /proc/interrupts file (reveals the OS tick frequency) and /proc/cpuinfo (processor frequency).

It is now possible to write an application that uses any desired fraction of the available CPU cycles, as in the cheat_attack function (Fig. 3, right). This first calculates the number of clock cycles that constitute the desired percentage of the clock tick interval. It then iterates doing its computation, while checking whether the desired limit has been reached at each iteration. When the limit is reached, the application goes to sleep for zero time, blocking till after the next tick. The only assumption is that the computation can be broken into small pieces, which is technically always possible to do (though in Section 4 we further show how to cheat without this assumption). This 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 zero resources.

## 3.2 Experimental Results

To demonstrate that this indeed works as described, we implemented such an application and ran it on a 2.8GHz Pentium-IV, running a standard Linux-2.6.16 system 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 (such as interrupts and context switches) using the Klogger tool [17]. A detailed rendition of precisely what happened is given in Fig. 4. 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.

Fig. 5 scatter-plots the billing accuracy, where each point represents one quantum. With accurate accounting we would have seen a diagonal, but this is not the case. While the cheat process runs for just over 3ms each time, it is billed for 0 (bottom right disk). The honest process, on the other hand, typically runs for less than 1ms, but is billed for 4 (top left); on rare occasions it runs for nearly a whole tick, due to some interference that caused the cheater to miss one tick (top right); the cheater nevertheless recovers at the following tick. The other processes run for a very short time and are never billed.

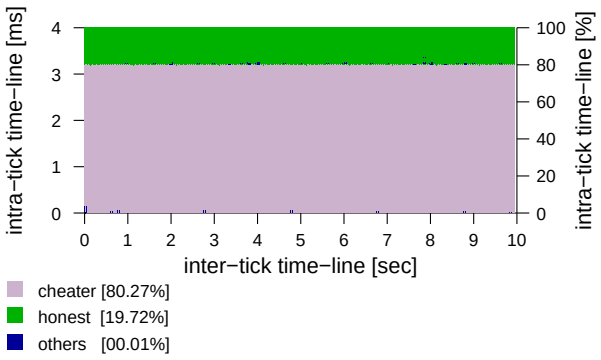



Figure 4: *Timeline of 10 seconds of competition between a cheat and honest processes. Legends give the distribution of CPU cycles.*

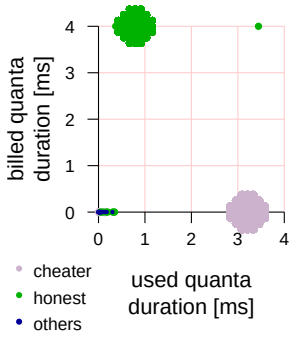



Figure 5: *Billing accuracy achieved during the test shown in Fig. 4.*

```
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.sh
5521   dants    15    0   2352    828    732   S     0.0  0.27     0:00.00  cheater
```

Figure 6: *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%.*

The poor accounting information propagates to system usage monitoring tools. Like any monitoring utility, the view presented to the user by top is based on OS billing information, and presents a completely distorted picture as can be seen in Fig. 6. 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 on the other hand is shown as using 0 time and 0% of the CPU. Moreover, it is reported as being 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.

Our demonstration used a setting of 80% for the cheat application (a 0.8 fraction argument to the cheat_attack function in Fig. 3). But other values can be used. Fig. 7 shows that the attack is indeed very accurate, and can achieve precisely the desired level of usage. Thus, an attacker that wants to keep a low profile can set the cheating to be a relatively small value (e.g. 15%); the chances users will notice this are probably very slim.

Finally, our demonstration have put forth only one competitor against the cheater. But the attack is in fact successful regardless of the number of competitors that form the background load. This is demonstrated in Fig. 8: An honest process (left) gets its *equal share* of the CPU, which naturally becomes smaller and smaller as more competing processes are added. For example, when 5 processes are present, each gets 20%. In contrast, when the process is cheating (right) it always gets what it wants, despite the growing competition. The reason of course is that the cheater has very a high priority, as it appears as consuming no CPU cycles, which implies an immediate service upon wakeup.

## 4 Running Unmodified Applications

A potential drawback of the above design is that it requires modifying the application to incorporate the cheat code. Ideally, from an attacker's perspective, there should be a "cheat" utility such that invoking e.g.

```
cheat 95% application
```

would execute the application as a 95%-cheater, without having to modify and recompile its code. This section describes two ways to implement such a tool.

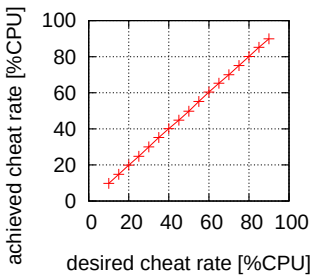



Figure 7: *The attack is very accurate and the cheater gets exactly the amount of CPU cycles it requested.*

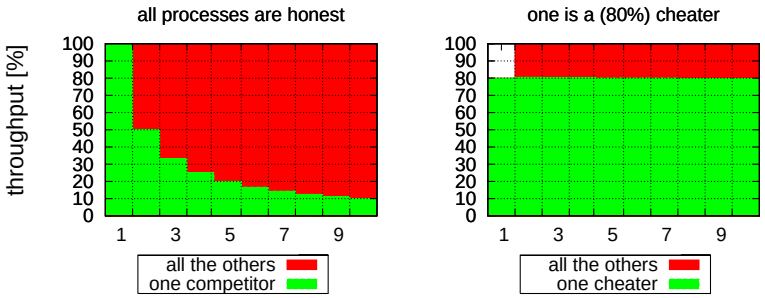



Figure 8: *Cheating is immune to the background load.*

## 4.1 Cheat Server

The sole challenge a cheat application faces is obtaining a timing facility that is finer than the one provided by the native OS. Any such facility would allow the cheater to systematically block before ticks occur, insuring it is never billed and hence the success of the attack. In the previous section, this was obtained by subdividing the work into short chunks and consulting the cycle counter at the end of each. A possible alternative is obtaining the required service using an external machine, namely, a *cheat server*.

The idea is very simple. Using a predetermined *cheat protocol*, a client opens a connection and requests the remote server to send it a message at some designated high-resolution time, before the next tick on the local host occurs. (The request is a single UDP packet specifying the required interval in nanoseconds; the content of the response message is unimportant.) The client then polls the connection to see if a message arrived, instead of consulting the cycle counter. Upon the message arrival, the client as usual sleeps-zero, wakes up just after the next tick, sends another request to the server, and so on. The only requirement is that the server would indeed be able to provide the appropriate timing granularity. But this can be easily achieved if the server busy-waits on its cycle counter, or if its OS is compiled with a relatively high tick rate (the alternative we preferred).

By switching the fine-grained timing source — from the cycle counter to the network — we gain one important advantage: instead of polling, we can now sleep-wait for the event to occur, e.g. by using the select system call. This allows us to divide the cheater into two separate entities: the *target application*, which is the the unmodified program we want to run, and the *cheat client*, which is aware of the cheat protocol, provisions the target application, and makes sure it sleeps while ticks occur. The client exercises its control by using the standard SIGSTOP/SIGCONT signals, as depicted in Fig. 9:

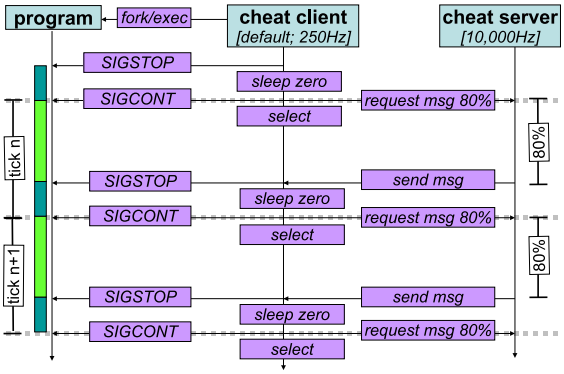



Figure 9: *The cheat protocol, as used by a 80%-cheater.*

1. The client forks the target application, sends it a stop signal, and goes to sleep till the next tick.
2. Awoken on a tick, the client does the following:
   (a) It sends the cheat server a request for a timing message including the desired interval.
   (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.
3. 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).
4. At the due time, the cheat server sends its message to the cheat client. This causes a network interrupt, and typically the immediate scheduling of the cheat client (which also looks like a chronic sleeper).
5. 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-zero, till the next (local) tick.
6. Upon the next tick, it will resume from step 2.

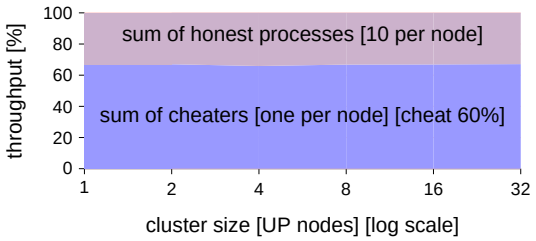



Figure 10: *The combined throughput of honest vs. 60%-cheating processes, as a function of the number of cluster nodes used. On each node there are ten honest processes and one cheater running. The cheaters' throughput indicates that the server simultaneously provides good service to all clients.*

```
void cheat_analysis()
{
    cycle_t c = get_cycles();

    if( c-tick_start >= WORK ) {
        nanosleep(&zero,0);
        tick_start = get_cycles();
    }
}
```

Figure 11: *The injected cheat "analysis" routine. (The WORK macro expends to the number of cycles that reflect the desired cheat fraction; the tick_start global variable is initialized beforehand to hold the beginning of the tick in which the application was started.)*

To demonstrate that this works, we have implemented this scenario, hijacking a shared departmental cluster of Pentium-IV machines. 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 adds overhead [15], this was 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. 10.

## 4.2 Binary Instrumentation

Using a cheat server is the least wasteful cheating method in terms of throughput loss, as it avoids all polling. The drawback however is the resulting network traffic that can be used to detect the attack, and the network latency which is now a factor to consider (observe the cheaters' throughput in Fig. 10 that is higher than requested). Additionally, it either requires a dedicated machine to host the server (if it busy-waits to obtain the finer resolution) or the ability to install a new kernel (if resolution is obtained through higher tick rate). Finally, the server constitutes a single point of failure.

Binary instrumentation of the target application is therefore an attractive alternative, potentially providing a way to turn an arbitrary program into a cheater, requiring no recompilation and avoiding the drawbacks of the cheat-server design. The idea is to inject the cheating code directly into the executable, instead of explicitly including it in the source code. To quickly demonstrate the viability of this approach we used *Pin*, a dynamic binary instrumentation infrastructure from Intel [33], primarily used for analysis tasks as profiling and performance evaluation. Being dynamic, Pin is similar to a just-in-time (JIT) compiler that allows attaching *analysis routines* to various pieces of the native machine code upon the first time they are executed.

The routine we used is listed in Fig. 11. Invoking it often enough would turn any application into a cheater. The question is where exactly to inject this code, and what is the penalty in terms of performance. The answer to both questions is obviously dependent on the instrumented application. For the purpose of this evaluation, we chose to experiment with an event-driven simulator of a supercomputer scheduler we use as the basis of many research effort [39, 19, 18, 51]. Aside from initially reading an input log file (containing a few years worth of parallel jobs' arrival times, runtimes, etc.), the simulator is strictly CPU intensive. The initial part is not included in our measurements, so as *not* to amortize the instrumentation overhead and overshadow its real cost by hiding it within more expensive I/O operations.

Fig. 12 shows the slowdown that the simulator experienced as a function of the granularity of the injection. In all cases the granularity was fine enough to turn the simulator into a full fledged cheater. Instrumenting every machine instruction in the program incurs a slowdown of 123, which makes sense because this is approximately the duration of cheat_analysis in cycles. This is largely dominated by the rdtsc operation (read time-stamp counter; wrapped by get_cycles), which takes about 90 cycles. The next grain size is a *basic block*, namely, a single-entry single-exit instructions sequence (containing no branches). In accordance to the well known assertion that "the average basic block size is around four to five instructions" [56], it incurs a slowdown of of 25, which is indeed a fifth of the slowdown associated with instructions. A *trace* of instructions (associated with the hardware trace-cache) is defined to be a single-entry multiple exits sequence of basic blocks that may be separated spatially, but are adjacent temporally [43]. This grain size further reduces the slowdown to 15. Instrumenting at the coarser function level brings us to a slowdown factor of 3.6, which is unfortunately still far from optimal.

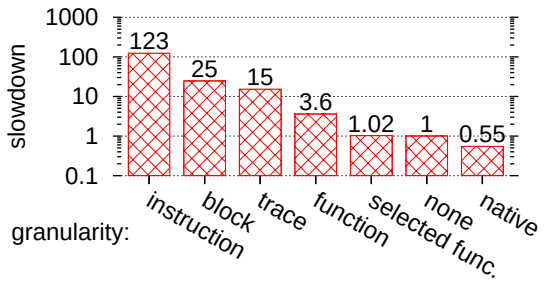



Figure 12: *The overheads incurred by cheat-instrumentation, as function of the granularity in which the* cheat_analysis *routine is injected. The Y axis denotes the slowdown penalty due to the instrumentation, relative to the runtime obtained when no instrumentation takes place ("none").*

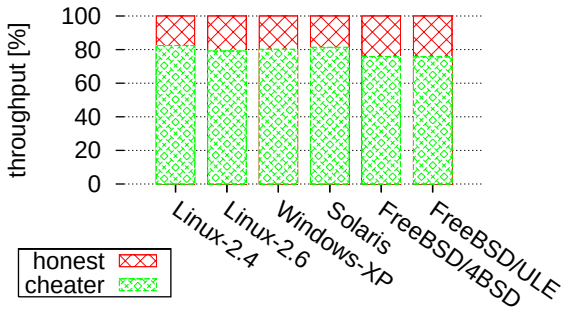



Figure 13: *Throughput of a 80%-cheater competing against an honest process, under the operating systems with which we experimented. These measurements were executed on the following OS versions: Linux 2.4.32, Linux 2.6.16, Window XP SP2, Solaris 10 (SunOS 5.10 for i386), and FreeBSD 6.1.*

The average instructions-per-function number within a simulator run, is very small (about 35), the result of multiple abstraction layers within the critical path of execution. This makes the function-granularity inappropriate for injecting the cheat code to our simulator, when attempting to turn it into an efficient cheater. Furthermore, considering the fact that nowadays a single tick consists of millions of cycles (about 11 millions on the platform we used, namely, a 2.8 GHz Pentium-IV at 250 Hz tick rate), a more adequate grain size for the purpose of cheating would be, say, tens of thousands of cycles. Thus, a slightly more sophisticated approach is required. Luckily, simple execution profiling (using Pin or numerous other tools) quickly reveal where an application spends most of its time; in the case of our simulator this was within two functions, the one that pops the next event to simulate and the one that searches for the next parallel job to schedule. By instructing Pin to instrument only these two functions, we were able to turn the simulator into a cheater, while reducing the slowdown penalty to less than 2% of the baseline. We remark that even though this selective instumentation process required our manual intervention, we believe it reflects a fairly straightforward and simple methodology that can probably be automated with some additional effort.

Finally, note that all slowdowns were computed with respect to the runtime of a simulator that was *not* instrumented, but still executed under Pin. This was done so as not to pollute our evaluation with unrelated Pin-specific performance issues. Indeed, running the simulator natively is 45% faster than the Pin baseline, a result of Pin essentially being a virtual machine [33]. Other binary instrumentation methods, whether static [48], exploiting free space within the executable itself [41], or linking to it loadable modules [4], do not suffer from this deficiency and are expected to close the gap.

## 5 General-Purpose Schedulers

The results shown so far are associated with Linux-2.6. To generalize, we experimented with other ticking operating systems and found that they are all susceptible to the cheat attack. The attack implementation was usually as shown in Fig. 3, possibly replacing the nanosleep with a call to pause, which blocks on a repeated tick-resolution alarm-signal that was set beforehand using setitimer (all functions are standard POSIX; the exceptions was Windows XP, for which we used Win32's GetMessage for blocking). Fig. 13 shows the outcome of repeating the experiment described in Section 3 (throughput of simultaneously running a 80%-cheater against an honest process) under the various OSs.

Our high level findings were already detailed in Section 1.4 and summarized in Fig. 1. In this section we describe in more detail the design features that make schedulers vulnerable to cheating; importantly, we address the "partial quantum decay" mechanism of Windows XP and provide more details regarding FreeBSD, which separates billing from timing activity and requires a more sophisticated cheating approach.

### 5.1 Multilevel Feedback Queues

Scheduling in all contemporary general-purpose operating systems is based on a *multilevel feedback queue*. The details vary, but roughly speaking, the priority is a combination of a static component ("nice" value), and a dynamic component that mostly reflects lack of CPU usage, interpreted as being "I/O bound"; processes with the highest priority are executed in a round-robin manner. As the cheat process avoids billing, it gets the highest priority and hence can monopolize the CPU. This is what makes cheating widely applicable.

A potential problem with the multilevel feedback queues is that processes with lower priorities might starve. OSs employ various policies to avoid this. For example, **Linux 2.4** uses the notion of "epochs" [36]. Upon a new epoch, the scheduler allocates a new quantum to all processes, namely, allows them to run for an additional 60ms. The epoch will not end until *all runnable processes* have exhausted their allocation, insuring all of them get a chance to run before new allocations are granted. Epochs are initiated by the tick handler, as part of the third item in Section 1.1. The remaining time a process has to run in the current epoch also serves as its priority (higher values imply higher priority). Scheduling decisions are made only when the remaining allocation of the currently running process reaches zero (possibly resulting in a new epoch if no runnable processes with positive allocation exist), or when a blocked process is made runnable.

This design would initially seem to place a limit on the fraction of cycles hijacked by the cheater. However, as always, cheating works because of the manner Linux-2.4 rewards sleepers: upon a new epoch, a currently *blocked* process gets to keep half of its unused allocation, in addition to the default 60ms. As a cheater is never billed and always appears blocked when a tick takes place, its priority quickly becomes $\sum_{i=0}^{\infty} 60 \cdot 2^{-i} = 120$ (the maximum possible), which means it is always selected to run when it unblocks.

In **Solaris**, the relationship between the priority and the allocated quantum goes the other way [35]. When a thread is inserted to the run-queue, a table is used to allocate its new priority and quantum (which are two separate things here) 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. The table is designed such that processes that consume their entire allocation receive even longer allocations, but are assigned lower priorities. In contrast, threads that block or sleep are allocated higher priorities, but shorter quanta. By avoiding billing the cheater is considered a chronic sleeper that never runs, causing its priority to increase until it reaches the topmost priority available. The short-quanta allocation restriction is circumvented, because the scheduler maintains its own (misguided) CPU-accounting based on sampling ticks.

## 5.2 Prioritization For Interactivity

An obvious feature of the Linux 2.4 and Solaris schemes is that modern interactive processes (as games or movie players that consume a lot of CPU cycles) will end up having low priority and will be delayed as they wait for all other processes to complete their allocations. This is an inherent feature of trying to equally partition the processor between competing processes. **Linux 2.6** therefore attempts to provide special treatment to processes it identifies as interactive by maintaining their priority high and by allowing them to continue to execute even if their allocation runs out, provided other non-interactive processes weren't starved for too long [32]. A similar mechanism is used in the **ULE** scheduler on **FreeBSD** [42].

In both systems, interactive processes are identified based on the ratio between the time they sleep and the time they run, with some weighting of their relative influence. If the ratio passes a certain threshold, the process is deemed interactive. This mechanism plays straight into the hands of the cheater process: as it consistently appears sleeping, it is classified interactive regardless of the specific value of the ratio. The anti-starvation mechanism is irrelevant because other processes are allowed to run at the end of each tick when the cheater sleeps. Thus, cheating would have been applicable even in the face of completely accurate CPU accounting. (The same observation holds for Windows XP, as described in the next subsection.)

We contend that the interactivity weakness manifested by the above is the result of two difficulties. The first is how to identify multimedia applications, which is currently largely based on their observed sleep and CPU consumption patterns. This is a problematic approach: our cheater is an example of a "false positive" it might yield. In a related work we show that this problem is inherent, namely, that typical CPU-intensive interactive application can be paired with non-interactive applications that have identical CPU consumption patterns [14]. Thus, it is probably impossible to differentiate between multimedia applications and others based on such criteria, and attempts to do so are doomed to fail; we argue that the solution lies in directly monitoring how applications interact with devices that are of interest to human users [16].

The second difficulty is how to schedule a process identified as being interactive. The problem here arises from the fact that multimedia applications often have both realtime requirements (of meeting deadlines) and significant computational needs. Such characteristics are incompatible with the negative feedback of "running reduces priority to run more", which forms the basis of the classic general-purpose scheduling [5] (as in Linux 2.4, Solaris, and FreeBSD/4BSD) and only delivers fast response times to applications that require little CPU. Linux-2.6, FreeBSD/ULE, and Windows XP tackled this problem in a way that compromises the system. And while it is possible to patch this to a certain extent, we argue that any significant divergence from the aforementioned negative-feedback design necessitates a much better notion of what is important to users than can ever be inferred solely from CPU consumption patterns [14, 16].

## 5.3 Partial Quantum Decay

In **Windows XP**, the default time quantum on a workstation or server is 2 or 12 timer ticks, respectively, with the quantum itself having a value of "6" ($3 \times 2$) or "36" ($3 \times 12$), implying that every clock tick decrements the quantum by 3 units [44]. The reason a quantum is stored internally in terms of a multiple of 3 units per tick rather than as single units is to allow for "partial quantum decay". Specifically, each waiting thread is charged one unit upon wakeup, so as to prevent situations in which a thread avoids billing just because it was asleep when the tick occurred. Hence, the cheater loses a unit upon each tick. Nonetheless, this is nothing but meaningless in comparison to what it gains due its many sleep events.

After a nonzero wait period (regardless of how short), Windows XP grants the awakened thread a "priority boost" by moving it a few steps up within the multilevel feedback queue hierarchy, relative to its base priority. Generally, following a boost, threads are allowed to exhaust their full quantum, after which they are demoted one queue in the hierarchy, allocated another quantum, and so forth until they reach their base priority again. This is sufficient to allow cheating, because a cheater is promoted immediately after being demoted (as it sleeps on every tick). Thus, it consistently maintains a higher position relative to the "non-boosted" threads and therefore always gets the CPU when it awakes. By still allowing others to run at the end of each tick, it prevents the anti-starvation mechanism from kicking in.

Note that this is true regardless of whether the billing is accurate or not, which means XP suffers from the interactivity weakness as Linux 2.6 and FreeBSD/ULE. To make things even worse, "in the case where a wait is not satisfied immediately" (as for cheaters), "its [the thread's] quantum is reset to a full turn" [44], rendering the partial quantum decay mechanism (as any hypothetical future accurate billing) completely useless.

## 5.4 Dual Clocks

Compromising **FreeBSD**, when configured to use its **4BSD** default scheduler [37], required us to revisit the code given in Fig. 3. Noticing that timer-oriented applications often tend to synchronize with ticks and start to run immediately after they occur, the FreeBSD designers decided to separate the billing from the timing activity [26]. Specifically, FreeBSD 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 billing statistics (with frequency *STATHZ*). A running thread's time quantum is decremented by 1 every STATHZ tick. The test system that was available to us runs with a HZ frequency of 1000Hz and STATHZ frequency of ∼133Hz.

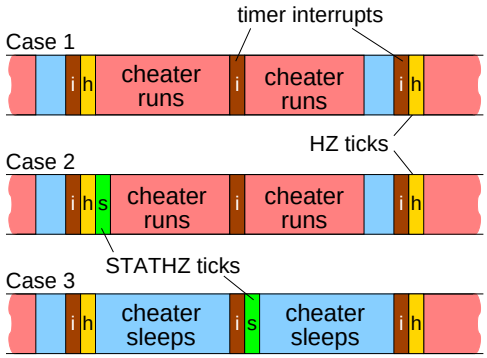



Figure 14: *The three possible alignments of the two FreeBSD clocks: no STATHZ tick between consecutive HZ ticks (case 1), STATHZ ticks falls on an even timer interrupt alongside a HZ tick (case 2), and a STATHZ tick falling on an odd clock interrupt between HZ ticks (case 3).*

Both the HZ and STATHZ timers are derived from a single timer interrupt, configured to fire at a higher frequency of $2 \times$ 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 ($\approx \frac{2000}{133}$). The possible alignments of the two are shown in Fig. 14. The HZ ticks are executed on each even timer interrupt (case 1). Occasionally the HZ and STATHZ ticks align on an even timer interrupt (case 2), and sometimes STATHZ is executed on an odd timer interrupt (case 3). By avoiding HZ ticks we also avoid STATHZ ticks in case 2. But to completely avoid being billed for the CPU time it consumes, the cheater 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 every second. Based on this, we modified the code 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. Further, the code reads the number of timer interrupts that occurred since the system was started, available through a sysctl call. The cheater then requests the system for signals at a constant HZ rate. The signal handler in turn accesses the STATHZ bitmap with a value of $(interrupt\_index + 1) \bmod (2\times \text{HZ})$ to check whether the *next* timer interrupt will trigger a STATHZ tick. This mechanism allows the cheater thread to identify case 3 and simply sleep until the next HZ tick fires. The need to occasionally sleep for two full clock ticks slightly reduces the achievable throughput, as indicated in Fig. 13.

## 6 Protecting Against the Cheat Attack

### 6.1 Degrees of Accuracy

While all ticking OSs utilize information that is exclusively based on sampling for the purpose of scheduling, some operating system also maintain precise CPU-usage information (namely, Solaris and Windows XP). Under this design, each kernel entry/exit is accompanied by reading the cycle counter to make the kernel aware of how many cycles were consumed by the process thus far, as well as to provide the user/kernel usage statistics. (Incidentally this also applies to the one-shot Mac OS X.) Solaris, provides even finer statistics by saving the time a thread spends in each of the thread states (running, blocked, etc.). While such consistently accurate information can indeed be invaluable in various contexts, it does not come without a price.

Consider for example the per system call penalty. Maintaining user/kernel statistics requires that (at least) the following would be added to the system call invocation path: two rdtsc operations (of reading the cycle counter at kernel entry and exit), subtracting of the associated values, and adding the difference to some accumulator. On our Pentium-IV 2.8GHz this takes $\sim$200 cycles (as each rdtsc operation takes $\sim$90 cycles and the arithmetics involves 64bit integers on a 32bit machine). This penalty is significant relative to the duration of short system calls, e.g. on our system, sigprocmask takes $\sim$430/1020 cycles with an invalid/valid argument, respectively, implying 20-47% of added overhead.

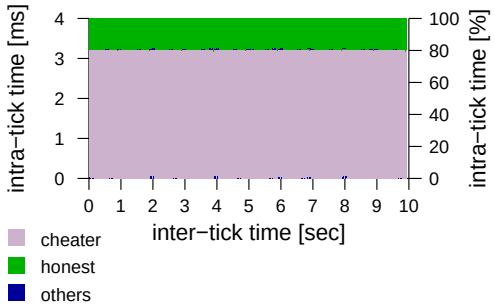
Stating a similar case, Liedtke argued against this type of kernel fine-grained accounting [30], and indeed the associated overheads may very well be the reason why systems like Linux and FreeBSD do not provide such a service. It is not our intent to express an opinion on the matter, but rather, to make the tradeoff explicit and to highlight the fact that designers need *not* face it when protecting against cheat attacks. Specifically, there is no need to know *exactly* how many cycles were consumed by a running process upon *each* kernel entry (and user/kernel or finer statistics are obviously irrelevant too). The scheduler would be perfectly happy with a much lazier approach: that the information would be updated *only upon a context switch*. This is a (1) far less frequent and a (2) far more expensive event in comparison to a system call invocation, and therefore the added overhead of reading the cycle counter is made relatively negligible.

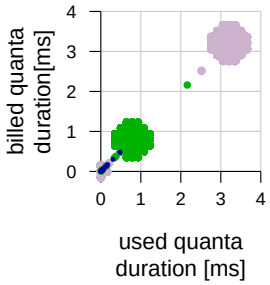
### 6.2 Patching the Kernel

We implemented this "lazy" perfect-billing patch within the Linux 2.6.16 kernel. It is only a few dozen lines long. The main modification is in the task_struct structure to replace the time_slice field that counts down a process' CPU allocation in a resolution of "jiffies" (the Linux term for clock ticks). It is replaced by two fields: ns_time_slice, which counts down the allocated time slice in nanoseconds instead of jiffies, 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 (this simply replaces the original time_slice jiffy decrement, but with the improvement of only accounting for cycles actually used by this process), and from within the schedule function just before a context switch (this is the new part). The rest of the kernel is unmodified, and still works in a resolution of jiffies. This was done by replacing accesses to time_slice with an inlined function that wraps ns_time_slice and rounds it to jiffies.

Somewhat surprisingly, using this patch did not 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 had full information about this (Fig. 15). As explained in Section 5, this happened because of the extra support for "interactive" processes introduced in the 2.6 kernel. The kernel identifies processes that yield a lot as interactive, provided their nice level is not too high. When an "interactive" process exhausts its allocation and should be moved from the "active array" into the "expired array", it is nevertheless allowed to remain in the active array, as long as already expired processes are not starved (they're not: the cheater runs less than 100% of the time by definition, and thus the Linux anti-starvation mechanism is useless against it). In effect, the scheduler is overriding its own quanta allocations; this is a good demonstration of the two sides of cheating prevention: it is not enough to have good information — it is also necessary to use it effectively.

In order to rectify the situation, we disallowed processes to circumvent their allocation by commenting out the line that reinserts expired "interactive" processes to the active array. As shown in Fig. 16, this has finally succeeded to defeat the attack. The timeline is effectively divided into epochs of 200ms (corresponding to the combined duration of the two 100ms time-slices of the two competing processes) in which the processes share the CPU equitably. While the "interactive" cheater has higher priority (as its many block events gains it a higher position in the multilevel queue hierarchy), this is limited to the initial part of the epoch, where the cheater repeatedly gets to run first upon each tick. However, after $\sim$125ms of which the cheater consumes 80%, its allocation runs out (125ms$\cdot\frac{80}{100}$=100ms). It is then moved to the expired array and its preferential treatment is temporarily disabled. The honest process is now allowed to catch up and indeed runs for $\sim$75ms until it too exhausts




| *process* | *total resource use* | |
|---|---|---|
| | *default (figs. on left)* | *nice +19* |
| cheater | 80.28% | 46.13% |
| honest | 19.71% | 53.87% |
| other | 0.01% | 0.00% |

Figure 15: *In Linux 2.6, cheating is still possible even with perfect billing (compare with Figs. 4-5).*

its quantum and is removed from the active array, leaving it empty. At this point the expired/active array are swapped and the whole thing is repeated.

The above exposes a basic tradeoff inherent to prioritizing based on CPU consumption patterns: one must either enforce a relatively equitable distribution of CPU cycles, or be exposed to attacks by cheaters that can easily emulate "interactive" behavior. (We note in passing that processes with +19 nice value are never regarded as interactive by the 2.6 kernel, so the "optimization" that allows interactive processes to deprive the others is effectively disabled; see right table in Fig. 15.)

Finally, let us discuss the patch overheads. The schedule function was ∼80 cycles (=5%) slower: $1636 \pm 182$ cycles on average instead of $1557 \pm 159$ without the patch ($\pm$ denotes standard deviation). At the same time, the overhead of a tick handler (the scheduler_tick function) was reduced by 17%, from $8439 \pm 9323$ to $6971 \pm 9506$. This is probably due to the fact that after the patch, the cheater ran much less, and therefore generated a lot less timers for the handler to process. Note that these measurements embody the *direct* overhead only (does not include traps to the kernel and back, nor cache pollution due to the traps or context switches). Also note that as the high standard deviations indicate, the distribution of ticks has a long tail, with maximal values around 150,000 cycles. Lastly, the patch did not affect the combined throughput of the processes, at all.

## 6.3 Other Potential Solutions

Several solutions may be used to prevent cheating applications from obtaining excessive CPU resources. Here we detail 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 overcharge 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 4. 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 as suggested by Liedtke in relation to user/kernel statistics [30]. This can work, but at the cost of overheads and complexity. Note however that true randomness is hard to come by, and it has already been shown that a system's random number generator could be reverse-engineered in order to beat the randomness [24]. 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.

## 6.4 A Note About Sampling

In the system domain, it is often tempting to say "let us do this chore periodically". It is simple and easy and therefore often the right thing to do. But if the chore is somehow related to accounting or safeguarding a system, and if "periodically" translates to "can be anticipated", then the design might be vulnerable. This observation is hardly groundbreaking. However, as with ticks, we suspect it is often brushed aside for the sake of simplicity. Without any proof, we now list a few systems that may posses this vulnerability.

At a finer granularity than ticks, one can find Cisco's *NetFlow* router tool that "preforms 1 in N periodic [non-probabilistic] sampling" [13] (possibly allowing an adversary to avoid paying for his traffic). At coarser granularity is found the per-node *infod* of the *MOSIX* cluster

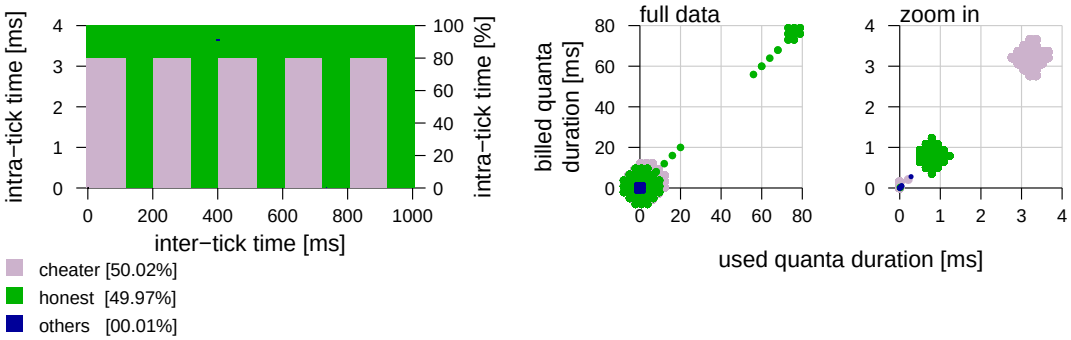



Figure 16: *Cheating is eliminated when expired processes are not reinserted to the active list (compare with Fig. 15).*

infrastructure [7], which wakes up every 5 seconds to charge processes that migrated to the node (work can be partitioned to shorter processes). The FAQ of IBM's internal file infrastructure called *GSA* (Global Storage Architecture) states that "charges will be based on daily file space snapshots" [22] (raising the possibility of a well-timed `mv` between two malicious cooperating users). And finally, the US Army *MDARS* (Mobile Detection Assessment Response System) patrol robots that "stop periodically during their patrols to scan for intruders using radar and infrared sensors" in search of moving objects [45] again raise the question of what exactly does "periodically" mean.

## 7 Conclusions

The "cheat" attack is a simple way to exploit computer systems. It allows an unprivileged user-level application to seize whatever fraction of the CPU cycles it wants, often in a secretive manner. The cycles used by the cheater are attributed to some other innocent application or simply unaccounted for, making the attack hard to detect. Such capabilities are typically associated with rootkits that, in contrast, require an attacker to obtain superuser privileges. We have shown that all major general-purpose systems are vulnerable to the attack, with the exception of Mac OS X that utilizes one-shot timers to drive its timing mechanism.

Cheating is based on two dominant features of general-purpose systems: that CPU accounting and timer servicing are tied to periodic hardware clock interrupts, and that the scheduler favors processes that exhibit low CPU usage. By systematically sleeping when the interrupts occur, a cheater appears as not consuming CPU and is therefore rewarded with a consistent high priority, which allows it to monopolize the processor.

The first step to protect against the cheat attack is to *maintain* accurate CPU usage information. This is already done by Solaris and Windows XP that account for each kernel entry. In contrast, by only accounting for CPU usage before a context switch occurs, we achieve sufficient accuracy in a manner more suitable for systems like Linux and FreeBSD that are unwilling to pay the associated overhead of the Solaris/Windows way. Once the information is available, the second part of the solution is to *incorporate* it within the scheduling subsystem (Solaris and XP don't do that).

The third component is to *use the information judiciously*. This is not an easy task, as indicated by the failure of Windows XP, Linux 2.6, and FreeBSD/ULE to do so, allowing a cheater to monopolize the CPU regardless of whether accurate information is used for scheduling or not. In an attempt to better support the ever increasing CPU-intensive multimedia component within the desktop workload, these systems have shifted to prioritizing processes based on their sleep-events *frequency*, instead of *duration*. This major departure from the traditional general-purpose scheduler design [5] plays straight into the hands of cheaters, which can easily emulate CPU-usage patterns that multimedia applications exhibit. A safer alternative would be to explicitly track user interactions [14, 16].

## Acknowledgments

Many thanks are due to Tal Rabin, Douglas Lee Schales, and Wietse Venema for providing feedback and helping to track down the Tsutomu Shimomura connection.

## References

[1] *access(2) manual, FreeBSD*. URL http://www.freebsd.org/cgi/man.cgi?query=access.

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

[3] J. Andrews, "*Interview: Ingo Molnar*". URL http://kerneltrap.org/node/517, Dec 2002.

[4] A. Anonymous, "*Building ptrace injecting shellcodes*". *Phrack* **10(59)**, p. 8, Jul 2002. URL http://www.phrack.org/archives/59.

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

[6] A-L. Barabasi, V. W. Freeh, H. Jeong, and J. B. Brockman, "*Parasitic compuiting*". *Nature* **412**, pp. 894–897, Aug 2001.

[7] A. Barak, A. Shiloh, and L. Amar, "*An organizational grid of federated MOSIX clusters*". In 5th *IEEE Int'l Symp. on Cluster Comput. & the Grid (CCGrid)*, May 2005.

[8] N. Borisov, R. Johnson, N. Sastry, and D. Wagner, "*Fixing races for fun and profit: how to abuse atime*". In 14th *USENIX Security Symp.*, pp. 303–314, Jul 2005.

[9] J. Butler, J. Undercoffer, and J. Pinkston, "*Hidden processes: the implication for intrusion detection*". In *IEEE-Information Assurance Workshop (IAW)*, pp. 116–121, Jun 2003.

[10] A. Cockcroft, "*How busy is the CPU, really?*". *SunWorld* **12(6)**, Jun 1998.

[11] D. Dean and A. J. Hu, "*Fixing races for fun and profit: how to use access(2)*". In 13th *USENIX Security Symp.*, pp. 195–206, Aug 2004.

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

[13] C. Estan and G. Varghese, "*New directions in traffic measurements and accounting: focusing on the elephants, ignoring the mice*". *ACM Trans. Comput. Syst.* **21(3)**, pp. 270–313, Aug 2003.

[14] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Desktop scheduling: how can we know what the user wants?*". In 14th *Int'l Workshop on Network & Operating Syst. Support or Digital Audio & Video (NOSSDAV)*, pp. 110–115, Jun 2004.

[15] 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.

[16] Y. Etsion, D. Tsafrir, and D. G. Feitelson, "*Process prioritization using output production: scheduling for multimedia*". *ACM Trans. on Multimedia Comput. Commun. & Appl. (TOMCCAP)* **2(4)**, pp. 318–342, Nov 2006.

[17] Y. Etsion, D. Tsafrir, S. Kirkpatrick, and D. G. Feitelson, "*Fine grained kernel logging with Klogger: experience and insights*". In *ACM EuroSys*, Mar 2007.

[18] D. G. Feitelson, "*Experimental analysis of the root causes of performance evaluation results: a backfilling case study*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)* **16(2)**, pp. 175–182, Feb 2005.

[19] D. G. Feitelson, "*Metric and workload effects on computer systems evaluation*". *Computer* **36(9)**, pp. 18–25, Sep 2003.

[20] E. Gabber, C. Small, J. Bruno, J. Brustoloni, and A. Silberschatz, "*The pebble component-based operating system*". In *USENIX Annual Technical Conference*, June 1999.

[21] S. Govindavajhala and A. W. Appel, "*Using memory errors to attack a virtual machine*". In *IEEE Symp. on Security and Privacy (SP)*, pp. 154–165, May 2003.

[22] "*Global storage architecture (GSA): FAQs*". URL http://pokgsa.ibm.com/projects/c/ccgsa/docs/gsa_faqs.shtml. IBM internal document.

[23] M. Guirguis, A. Bestavros, and I. Matta, "*Exploiting the transients of adaptation for RoQ attacks on internet resources*". In 12th *IEEE Int'l Conf. on Network Protocols (ICNP)*, pp. 184–195, Oct 2004.

[24] 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.

[25] G. Hoglund and J. Butler, *Rootkits: Subverting the Windows Kernel*. Addison-Wesley, 2005.

[26] P. Jeremy et al., "*CPU-timer rate*". Dec 2005. Thread from the "freebsd-stable – Production branch of FreeBSD source code" mailing list. URL http://lists.freebsd.org/pipermail/freebsd-stable/2005-December/020386.html.

[27] S. T. King, P. M. Chen, C. V. Yi-Min Wang, H. J. Wang, and J. R. Lorch, "*SubVirt: implementing malware with virtual machines*". In *IEEE Symp. on Security and Privacy (SP)*, p. 14, May 2006.

[28] A. Kuzmanovic and E. W. Knightly, "*Low-rate TCP-targeted denial of service attacks (the shrew vs. the mice and elephants)*". In *ACM SIGCOMM Conf. on Appli. Technologies Archit. & Protocols for Comput. Commun.*, pp. 75–86, Aug 2003.

[29] I. Leslie, D. McAuley, R. Black, T. Roscoe, P. Barham, D. Evers, R. Fairbairns, and E. Hyden, "*The design and implementation of an operating system to support distributed multimedia applications*". *IEEE J. Select Areas in Commun.* **14(7)**, pp. 1280–1297, Sep 1996.

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

[31] U. Lindqvist and E. Jonsson, "*How to systematically classify computer security intrusions*". In *IEEE Symp. on Security and Privacy (SP)*, pp. 154–163, May 1997.

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

[33] C-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "*Pin: building customized program analysis tools with dynamic instrumentation*". In *ACM Int'l Conf. on Programming Lang. Design & Impl. (PLDI)*, pp. 190–200, Jun 2005. Site: rogue.colorado.edu/Pin.

[34] J. Markoff, "*Attack of the zombie computers is growing threat*". *New York Times*, Jan 2007.

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

[36] S. Maxwell, *Linux Core Kernel Commentary*. Coriolis Group Books, 2nd ed., 2001.

[37] 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.

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

[39] A. W. Mu'alem and D. G. Feitelson, "*Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling*". *IEEE Trans. Parallel & Distributed Syst.* **12(6)**, pp. 529–543, Jun 2001.

[40] C. M. Olsen and C. Narayanaswami, "*PowerNap: An efficient power management scheme for mobile devices*". *IEEE Trans. on Mobile Computing* **5(7)**, pp. 816–828, Jul 2006.

[41] P. Padala, "*Playing with ptrace, part II*". *Linux J.* **2002(104)**, p. 4, Dec 2002. URL http://www.linuxjournal.com/article/6210.

[42] J. Roberson, "*ULE: a modern scheduler for FreeBSD*". In *USENIX BSDCon*, pp. 17–28, Sep 2003.

[43] E. Rotenberg, S. Bennett, and J. E. Smith, "*Trace cache: a low latency approach to high bandwidth instruction fetching*". In 29th *IEEE Int'l Symp. on Microarchit. (MICRO)*, pp. 24–35, 1996.

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

[45] B. Shoop, D. M. Jaffee, and R. Laird, "*Robotic guards protect munitions*". *Army AL&T*, p. 62, Oct-Dec 2006.

[46] S. Sparks and J. Butler, "*Shadow walker: raising the bar for windows rootkit detection*". *Phrack* **11(63)**, p. 8, Jul 2005. URL http://www.phrack.org/archives/63.

[47] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "*A firm real-time system implementation using commercial off-the-shelf hardware and free software*". In 4th *IEEE Real-Time Technology & App. Symp.*, pp. 112–119, Jun 1998.

[48] A. Srivastava and A. Eustace, "*ATOM: a system for building customized program analysis tools*". *SIGPLAN Notices (Best of PLDI 1979-1999)* **39(4)**, pp. 528–539, Apr 2004.

[49] 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.

[50] H. Sun, J. C. S. Lui, and D. K. Y. Yau, "*Distributed mechanism in detecting and defending against the low-rate TCP attack*". *Comput. Networks* **50(13)**, p. Sep, 2006.

[51] D. Tsafrir, Y. Etsion, and D. G. Feitelson, "*Backfilling using system-generated predictions rather than user runtime estimates*". *IEEE Trans. on Parallel & Distributed Syst. (TPDS)*, 2007. To apper.

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

[53] D. Tsafrir, Y. Etsion, D. G. Feitelson, and S. Kirkpatrick, "*System noise, OS clock ticks, and fine-grained parallel applications*". In 19th *ACM Int'l Conf. on Supercomput. (ICS)*, pp. 303–312, Jun 2005.

[54] W. Z. Venema, D. L. Schales, and T. Shimomura, "*The novelty and origin of the cheat attack*". Jan 2007. Private email communication.

[55] K. M. Walker, D. F. Sterne, M. L. Badger, M. J. Petkac, D. L. Shermann, and K. A. Oostendorp, "*Confining root programs with domain and type enforcement (DTE)*". In 6th *USENIX Security Symp.*, Jul 1996.

[56] T-Y. Yeh and Y. N. Patt, "*Branch history table indexing to prevent pipeline bubbles in wide-issue superscalar processors*". In 26th *IEEE Int'l Symp. on Microarchit. (MICRO)*, pp. 164–175, May 1993.