# A Randomized Sampling Clock for CPU Utilization Estimation and Code Profiling

Steven McCanne* and Chris Torek*†
Lawrence Berkeley Laboratory
One Cyclotron Road
Berkeley, CA 94720
mccanne@ee.lbl.gov, torek@ee.lbl.gov

## Abstract

The UNIX *rusage* statistics are well known to be highly inaccurate measurements of CPU utilization. We have observed errors in real applications as large as 80%, and we show how to construct an adversary process that can use an arbitrary amount of the CPU without being charged. We demonstrate that these inaccuracies result from aliasing effects between the periodic system clock and periodic process behavior. Process behavior cannot be changed but periodic sampling can. To eliminate aliasing, we have introduced a randomized, aperiodic sampling clock into the 4.4BSD kernel. Our measurements show that this randomization has completely removed the systematic errors.

## 1 Introduction

Traditional implementations of the Unix operating system provide coarse grained, statistical measurements of CPU utilization. On each tick of the system clock, the CPU state is examined. If the processor is in *user mode*, the current process is charged with one sampling interval of *user time*. Similarly, if the processor is in *system mode*, the current process is charged *system time*

This approach is problematic. A process can become synchronized with the sampling clock, resulting in large scale errors in the utilization statistic. For instance, a process that runs in phase with the system clock might always surrender the CPU before the clock interrupt arrives, thereby accumulating no usage time.

CPU time estimation is of particular importance, as it drives the scheduling algorithm. If the utilization estimate is in error, scheduling, and hence system performance, can be adversely affected. Furthermore, the accuracies of the *getrusage* system call and the */bin/time* command will be compromised.

In this paper, we outline the theory behind the statistical CPU estimator. We then introduce a new approach based on randomization. Next, we explain how the new model fits into the current 4.4BSD system, and how it can drive code profiling as well. Finally, we give some case studies that demonstrate problems with the existing system and show that our approach has overcome them.

## 2 A Statistical Model

An exact measurement of CPU utilization would require the precise timing of every interrupt and system call. Since this is prohibitive, systems rely on a cheaper methodology based on sampling. Here, a sequence of samples of the CPU state is used to estimate the true utilization percentage, which in turn can be viewed a probability. For example, the probability that the CPU is in a given state is simply ratio of the time spent in that state to the elapsed time.

For the CPU estimator, there are three relevant CPU states: user mode, system mode, and interrupt mode. Call the probabilities of being in each of these states $p_u$, $p_s$, and $p_i$ respectively. Then, if a process runs for $T_e$ time units, the amount of time spent in each CPU state is simply

$$\begin{aligned} T_u &= p_u T_e \\ T_s &= p_s T_e \\ T_i &= p_i T_e \end{aligned}$$

We need to devise a sampling experiment that produces unbiased estimates for $p_u$, $p_s$, and $p_i$. Moreover, the estimates should get better as we make more observations.

The observations of CPU state can be related to the probability estimates using elementary probability theory. The sequence of observations comprises what probability theory calls a *random sample*, and the Law of Large Numbers tells

*This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

†**This is a preprint of a paper to be presented at the 1993 Winter USENIX conference, January 25–29, 1993, San Diego, CA.**

us that the sample mean converges to the mean of each observation, provided the observations are independent. We can view each sample as a *Bernoulli* random variable, which is 1 with probability $p$ and 0 otherwise; its mean is $p$. Thus, assuming independence, the sample mean converges to $p$, which is what we want.

For example, consider the sequence of observations $\{U_1, U_2, \ldots, U_n\}$, where $U_k$ is 1 if the CPU is in user mode, and 0 otherwise. Each $U_k$ is Bernoulli with mean $p_u$. Thus, the Law of Large Numbers says that

$$p_u = \lim_{n \to \infty} \frac{U_1 + U_2 + \cdots + U_n}{n}$$

A good estimate of $p_u$ then is

$$\hat{p_u} = \frac{U_1 + U_2 + \cdots + U_n}{n}$$

Taking the sample sequence to be Bernoulli assumes that the underlying process is *stationary*, which means that the probabilities of being in each state remain constant over time. Although this is not generally true of programs, there is no way to proceed without making this assumption. Programmers often put code to be profiled inside a loop, or otherwise run the code many times. This repetitive behavior then has an overall stationary behavior. Furthermore, long lived processes generally exhibit repetitive behavior, so the assumption is reasonable for CPU estimation as well.

## 2.1 The Conventional CPU Estimator

In the conventional method, rather than compute the probability estimates mentioned above, estimates of $T_u$ and $T_s$ are directly maintained. Call these times $\hat{T_u}$ and $\hat{T_s}$. Assuming that the set of samples comprises a true random sample, this method would be equivalent to computing the probability estimates. Let $\Delta$ be the sampling interval. Then the algorithm computes $\hat{T_u}$ as

$$\hat{T_u} = \sum_{k=1}^{n} \Delta U_k = (n\Delta) \left( \frac{1}{n} \sum_{k=1}^{n} U_k \right) = T_e \hat{p_u} \qquad (1)$$

since $\hat{T_e} = n\Delta$ is an estimate of the elapsed time. Thus, $\hat{T_u}$ and $\hat{p_u} T_e$ are approximately equal. A similar analysis holds for $\hat{T_s}$.

This approach fails, however, because the samples used to compute $\hat{T_u}$ and $\hat{T_s}$ do not comprise a truly random sample. Since the sampling mechanism uses fixed intervals, random samples would only result if system and process behavior were itself random. The original implementation was probably based on this assumption. Unfortunately, programs are, for the most part, deterministic, so random behavior should not be expected. The BSD book [5] points out that the run time utilization estimates are in fact "statistical", but does not attempt to clarify the estimation technique.

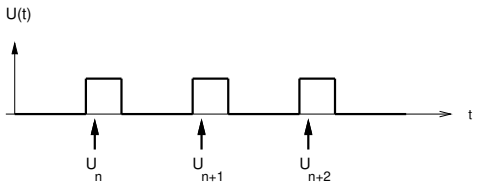


Figure 1: Process Isochrony

In any case, the statistical model allows us to clearly see where the conventional algorithm breaks down. The problem is that the sequence $\{U_n\}$ appearing in Equation 1 is not a sequence of *independent* observations. For example, knowing the previous observation gives you information about the next one—i.e., adjacent observations are dependent. Since the Law of Large Numbers applies only to sequences of *independent* random variables, the probability estimate will not necessarily converge to the true probability.

The problem that arises from this lack of independence is clearly illustrated in the case of an isochronous process. A process is characterized as isochronous if its behavior is periodic and consistent, for instance, as shown by the graph in Figure 1. The function $U(t)$ represents the user mode utilization of the CPU, as a function of time, while the arrows indicated the fixed rate sampling process. Because the sampling process is synchronized with the process behavior, the estimator will compute $\hat{p_u} = 1$, even though $p_u \neq 1$. This is a systematic error as it does not diminish with more samples.

## 2.2 Adding Randomization

Now that we know how the conventional estimator is failing, we would think it would be easy to correct the problem. All we need to do is change the sampling technique so that we get independent observations. This turns out to be a non-trivial problem.

In theory, the most straightforward approach would be to simply choose some number of random samples uniformly distributed over the lifetime of the system. But this approach is obviously infeasible since the system is continually running and its history is not retained.

An attractive alternative is to continue to use an interval based sampling approach, but to use random rather than fixed intervals. If we sample at a time $T_i$, then the next sample time is given by

$$T_{i+1} = T_i + W_{i+1}$$

where each $W_i$ is a random variable, and $T_0 = 0$.

Intuitively, randomizing the sampling clock phase should break any synchronization with process behavior. But how

can we be sure that the observations are statistically independent and hence that the sample mean will converge to the probability estimate? The answer depends on the distribution of $W$. For example, if $W$ is constant, then we have the existing approach, which we know won't work. More generally, if $W$ is *arithmetic*, for instance it takes on values only in $\{nk : n \geq 0, k = 0, 1, \ldots\}$, then we have a similar problem.

From a theoretical perspective, a particularly nice choice for $W$ is the exponential distribution. The sequence $\{T_k\}$ would then correspond to a Poisson process, for which a well known result is that the conditional distribution of arrivals on a subset of time, given their number, is uniform. In other words, if we know how many samples occurred over some interval, then those samples are uniformly distributed on that interval. These uniformly distributed random samples would result in a truly random aggregate sample.

However, implementation difficulties arise for exponentially distributed intervals. Occasionally, the time difference between adjacent samples will be smaller than the interrupt service time. Depending on how the clock hardware operates, race conditions could result when reprogramming the timer. Also, it is not clear what effect an occasional very large sample interval would have on other aspects of the system (for example, the scheduler).

Our solution is to let $W$ be uniformly distributed on $[T_{min}, T_{max}]$. In this case, $T_{min}$ can be chosen to be much larger than the interrupt service time, simplifying implementation.

We must be sure, however, that this approach, like the others, is unbiased. We can argue this using another result from probability theory, the Ergodic Theorem, which is a generalization of the Law of Large Numbers. This result says that the sample mean will converge to the true mean if the sequence of samples is ergodic (and not necessarily independent). Assuming the underlying process is stationary, and that our sampling process begins at $-\infty$, we can argue that our sample sequence is ergodic. We omit the details and refer the reader to [3, Ch. 6].

Note that the probability estimates converge independently of the frequency of the sampling clock. Only the *rate* of convergence is controlled by the mean sampling period. Thus, the average sampling rate, and hence the overhead of the CPU estimator, is dynamically adjustable. This contrasts with the existing system which required the rate to be configured into the kernel at compile time.

# 3 Implementation

Incorporating the randomized sampling model into the existing system was relatively straightforward. In the 4.4BSD kernel, all real-time and time-of-day events, including process scheduling, are driven off a fixed-rate *hardclock* interrupt. In the old system, the *hardclock* interrupt also gathered statistics; now they are driven off a separate *statclock* timer. Each time *statclock* returns from its interrupt context, the timer is reprogrammed for a random interval chosen from a uniform distribution as described in the previous section.

Process "wall clock time" is computed directly from the actual time of day at process switch. The *microtime* function is used to obtain a high resolution timestamp when the process is continued, and again when the process is suspended; the difference between these times is then added to a running sum. This figure becomes the $T_e$ factor in the approximation formulas.

Meanwhile, on each *statclock* tick, the current process, if any, is charged a user, system, or interrupt tick according to the CPU mode at the time of the *statclock* interrupt; call these counts $u$, $s$, and $i$ respectively. Since the sum of these counts is the total number of samples taken, the probability estimates are easily computed: $p_k = k/(u + s + i)$ for $k \in \{u, s, i\}$. From this and $T_e$, we can then compute $T_u$ and $T_s$. $T_i$ can be similarly computed, but since there is no way to identify the true source of such time it currently disappears into general system overhead.

The *statclock* abstraction is available only on machines with high-precision, programmable clocks. The randomized sampling intervals are generated by programming the clock's limit register with a pseudo-random number. To reduce overhead, a cheap-but-good random number generator is used [1]. On systems without programmable clocks, *statclock* is called directly from *hardclock*, and the functionality is unchanged from the existing system.

# 4 Code Profiling

Kernel support for user level code profiling [4] is carried out in a manner identical to CPU usage estimation. We therefore wanted to apply the lessons learned from the randomized sampling clock to the profiling system. Along the way, we fixed some problems with the traditional profiling support.

When a *statclock* tick occurs while a profiled process is running, a profiling buffer in user address space must be updated. In the previous system, this buffer update cannot be carried out by the clock interrupt handler because page faults are not permitted in an interrupt context. Instead, the clock handler schedules a profiling "asynchronous system trap" or AST, which causes a trap to occur just before returning to user mode. In this trap context, page faults may occur, and the user's profiling buffer is updated.

The 4.4BSD kernel avoids most such ASTs through two new routines to manipulate user memory from interrupt context. These routines attempt to update the user profiling counts directly. If a fault occurs, the update is aborted and the profiling code schedules an AST as before. Typically only a few such

ASTs are required to page in the user profiling buffer. From then on, the updates can be carried out cheaply from interrupt context.

System call profiling in the existing system is inaccurate. In this case, rather than update the user profile buffer during the system call, the kernel reads the process's accumulated system time at entry to each call and again at exit from each call. The difference between these two times is converted to a tick count, which is added as if from an AST. This includes the same interrupt-time excess found in the per-process statistics, and computing this value is complicated due to the need to turn time into ticks.

In 4.4BSD, system call profiling is still done at the end of each call for efficiency, but now it is merely a matter of subtracting the previous system tick count from the current count.

# 5 Results

We devised three experiments, two contrived and one from production software, that uncover the anomalies of the old CPU utilization estimator. The tests were run under both SunOS 4.1.1 and 4.4BSD. In each experiment, the anomalies were clearly evident under the old CPU estimator, while under 4.4BSD, they disappeared.

## 5.1 Interrupt Activity Interference

The first experiment clearly demonstrates that interrupt activity is charged to the current process as system time. A program was written that executes an infinite loop, and runs a 4 Hz alarm that logs system time usage. Since the program uses only a small amount of system CPU, just enough to process an alarm signal every 0.25 seconds, system time should accumulate *very* slowly.

This program was simultaneously run on two SPARCstation 1+ machines, one running SunOS 4.1.1 and the other 4.4BSD. Partway through execution, each host was exposed to an onslaught of interrupt activity[1]. The interrupt activity was then terminated, and finally, the program was stopped.

Figure 2 shows the plot of system CPU time versus real time for both processes. The lower line represents the 4.4BSD behavior, and is as expected—very little system time is accumulated. Under SunOS, however, during the interrupt activity, the process is charged with a significant amount of system CPU time. Note that this process had nothing to do with the interrupt activity; clearly, the statistics are skewed.

[1] The interrupt activity was generated by putting each host's network interface into *promiscuous* mode, causing all network packets to be processed. A 780KB/s Ethernet transfer was then initiated on the local net.

| | | *real* | *cpu* | *%cpu* |
|---|---|---|---|---|
| SunOS | w/o hog | 7.1 | 7.1 | 100% |
| | w/ hog | 584.5 | 334.8 | 57% |
| 4.4BSD | w/o hog | 7.1 | 7.0 | 99% |
| | w/ hog | 15.5 | 7.0 | 45% |

Table 1: Effect of hog on CPU bound process

## 5.2 A CPU Adversary

An adversarial program was written in an attempt to defeat the CPU utilization statistics altogether. This program, which we call *hog*, first estimates the phase of the system clock. It then enters a hard loop, performing *gettimeofday* system calls, until just before a *hardclock* tick is going to happen. At this point, it goes to sleep until the next system clock. Thus, the process is never charged with a sampling tick, and never will accumulate CPU time.[2] Furthermore, its scheduling priority remains favorable, so it always runs, even if there are other processes waiting.

Since *hog* sleeps every other system clock tick, it will use at most half of the CPU. Thus, two *hogs* are required to use up the whole CPU. We augmented *hog* to fork once from *main*, and the results were dramatic. Table 1 shows timings for a CPU bound process when run in the presence and absence of *hog*. The CPU bound test program simply counted to 10 million. The first column gives the real time of execution, while the second column gives the CPU time as measured by the system. Without *hog*, the two systems are similar, as expected. But with *hog*, the SunOS process takes 80 times longer to finish even though the *time* command reports a utilization of 57%. If this figure were correct, it should only have taken 1.75 times longer. Under 4.4BSD, the test process gets a fair share of the CPU. The 15.5 seconds of real time is consistent with 45% utilization.

In taking these measurements, we noticed anomalous scheduler behavior in 4.4BSD. Even though the CPU utilization estimates were accurate, the scheduler often exhibits unfairness between the CPU bound counting program and the *hog*. In the presence of the *hog*, the CPU bound process got as little as 9% and as much as 65% of the CPU. The BSD scheduler is known to be flawed [6], but it should do better here. This remains to be investigated.

The code for *hog* is given in the appendix.

## 5.3 The Isochronous Anomaly

The previous two experiments were run under controlled conditions in an attempt to expose the worst case behavior of the utilization error. However, we have experienced problematic

[2] Actually, there is a low probability that the process is run just before a clock interrupt, in which case there is insufficient time to discover that the interrupt is coming.

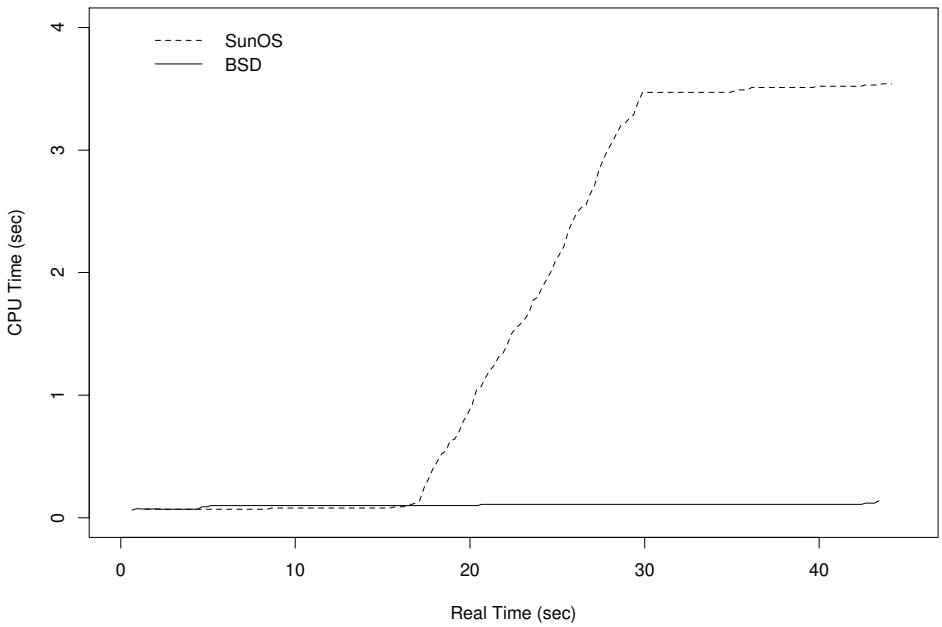


Figure 2: Interrupt Interference with System CPU Time

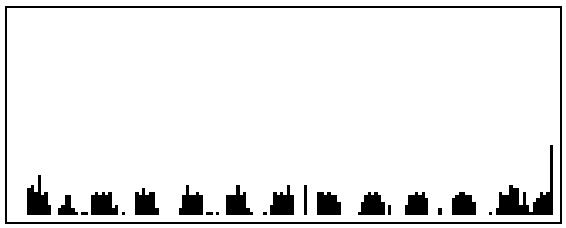

Figure 3: Unexpected CPU Load Oscillations

behavior in a normal operating environment. For example, Figure 3 shows a window dump of *xcpu*, displaying a load oscillating between 0 and 10%, with a period of about one minute. The oscillations in the load average were not expected, and the cause was an audio conferencing program called *vat* [2]. *Vat* processes a frame of audio samples every 22.5ms and should therefore exert a constant CPU load. But *xcpu* indicated otherwise.

To verify our theory that *vat* was causing these load fluctuations, we modified it to log its CPU time, every 22.5ms, to a debugging file, and ran the new version under both SunOS and 4.4BSD. Figure 4 shows these results. Since *vat* operates continuously, you would expect its CPU time to increase linearly. This is the case for 4.4BSD. But under SunOS, there are flat and steep regions of about 30 seconds in length. This anomaly is clearly due to the inaccuracy of the old sampling methodology.

The problem is that *vat* is running synchronously with the system clock. Since *vat* runs *exactly* every 22.5ms, it is aliased onto the 10ms system clock in only four possible slots.[3] As a result, when the minimum phase difference allows *vat* to carry out all of its processing before the clock ticks arrive, CPU time never accumulates. On the other hand, when the phase is such that ticks always occur, too much CPU time is charged. These two modes of operation are reflected in the SunOS data as the flat and steep regions. While this argument predicts that we should remain in a given mode indefinitely, the data actually oscillates between the two modes with a period of about one minute. Without going into detail, various effects, some internal to *vat* and some due to unrelated interrupt activity, can cause the phase between *vat's* behavior and the system clock to drift.

# 6 Conclusion

We have presented a new approach for measuring CPU utilization that uses randomized sampling to overcome the deficiencies of the old approach. Randomization prevents an adversary from foiling the utilization estimator and precludes synchronization between the sampling system and process behavior. We have corrected problems with erroneous accounting of interrupt activity, and we have streamlined the kernel support for code profiling. Finally, we have conducted experiments to demonstrate that the new system performs as expected.

[3] The least common multiple of the two periods is 90ms. Therefore, there are $90/22.5 = 4$ phase positions for *vat* to cycle through.

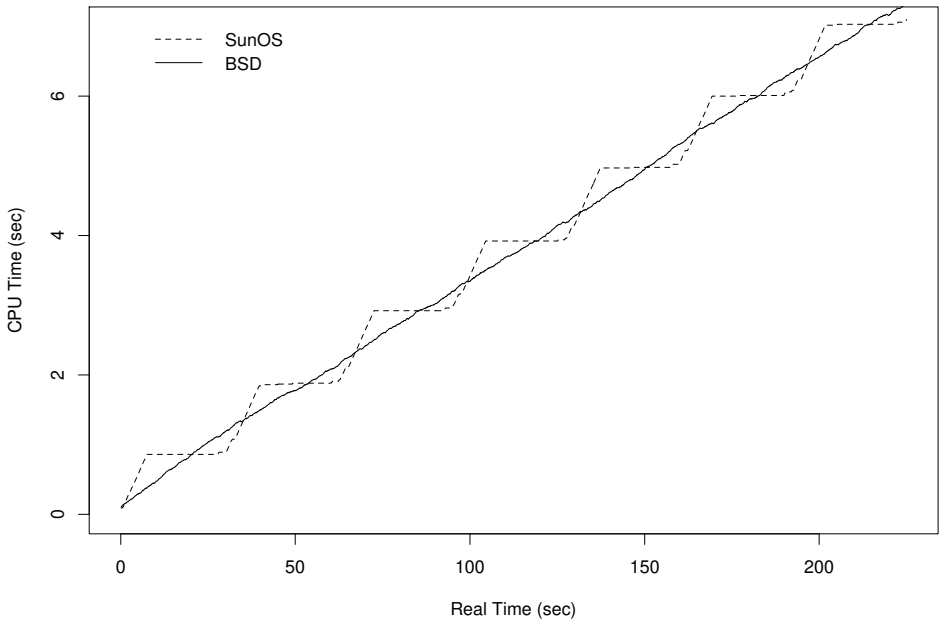


Figure 4: System Time Oscillations

# 7 Acknowledgements

Van Jacobson originally suggested that randomization be used to circumvent the statistical biases in the CPU estimator. Additionally, he helped interpret the results of our experiments and provided suggestions for implementation strategies.

The idea that the CPU scheduler can be defeated is not new. Dheeraj Sanghi and Olafur Gudmundsson wrote a program similar to the hog presented in this paper. According to them, the idea was originally proposed by Ashok Agrawala.

We are grateful to Spyros Papadakis for helping us formulate our probabilistic arguments. His advice was impeccable; any errors were introduced by us.

Finally, we would like to thank Vern Paxson, Van Jacobson, Craig Leres, Deana Goldsmith, and the referees for their helpful comments on drafts of this paper.

# 8 Availability

The *statclock* code appears in the 4.4BSD alpha release. Currently, SPARCstation and HP-9000/300 series machines are supported.

# References

[1] CARTA, D. G. Two fast implementations of the "minimal standard" random number generator. *Communications of the ACM 33*, 1 (Jan. 1990).

[2] CASNER, S., AND DEERING, S. First IETF Internet audiocast. *ConneXions 6*, 6 (1992), 10–17.

[3] DURRETT, R. *Probability: Theory and Examples*. Brooks/Cole Publishing Company, 1991.

[4] GRAHAM, S. L., KESSLER, P. B., AND MCKUSICK, M. K. gprof: A call graph execution profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (June 1982).

[5] LEFFLER, S. J., MCKUSICK, M. K., KARELS, M. J., AND QUARTERMAN, J. S. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Addison-Wesley, 1989.

[6] STRAATHOF, J. H., THAREJA, A., AND AGRAWALA, A. UNIX scheduling for large systems. In *Proceedings of the 1986 Winter USENIX Technical Conference* (Denver, CO, Jan. 1986), USENIX, pp. 111–138.

# Author Information

Steven McCanne has been with the Lawrence Berkeley Laboratory since 1988, working on network analysis tools and remote conferencing applications. He holds a B.S. degree in Electrical Engineering and Computer Science from U.C. Berkeley, and is currently a Ph.D. student in Computer Science at U.C.B.

Chris Torek has been rewriting bits of the Berkeley kernel for about six years. He joined LBL in 1991, and has been working on porting BSD to the SPARCstation. In his off hours he spends far too much time on USENET.

# Appendix: Adversary Source Code

```
#include <signal.h>
#include <sys/param.h>
#include <sys/time.h>
#include <sys/resource.h>

#define tvdiff(x, y) \
        (1000000 * ((y).tv_sec - (x).tv_sec) + (y).tv_usec - (x).tv_usec)

struct timeval hc;          /* our best guess for when a hardclock happened */
struct timeval now;         /* hold time-of-day for signal handler */
volatile int ntick;

alarm_handler()
{
        u_long u;
        struct timeval tv;
        static int mindel = 5000000;

        ++ntick;
        gettimeofday(&tv, 0);
        u = tvdiff(now, tv);
        if (u < mindel) {
                mindel = u;
                hc = tv;
        }
        usleep(1);
}

/*
 * Try to figure out when hardclock happens.
 */
struct timeval
train()
{
        struct itimerval it;

        signal(SIGVTALRM, alarm_handler);
        it.it_interval.tv_usec = it.it_value.tv_usec = 1;
        it.it_interval.tv_sec = it.it_value.tv_sec = 0;
        /*
         * Sleep right before we set the timer.  This way, we're sure to
         * get a whole time slice, and we won't be switched out before
         * we estimate the hardclock time.
         */
        usleep(1);
        setitimer(ITIMER_VIRTUAL, &it, 0);
        for (ntick = 0; ntick < 20; )
                gettimeofday(&now, 0);
        /*
         * Turn the timer off.
         */
        it.it_interval.tv_usec = it.it_value.tv_usec = 0;
        setitimer(ITIMER_VIRTUAL, &it, 0);

        return (hc);
}
```

```
int
main(argc, argv)
        int argc;
        char **argv;
{
        long us, s, bias, delta, off;
        struct timeval tv;

        /*
         * Determine when hardclocks are happening then compute a bias
         * with respect to an even multiple of hardclock ticks.
         * Assume 10ms tick.  Since a second is an even multiple of
         * a tick, we only need to look at usecs.
         */
        tv = train();
        bias = tv.tv_usec % 10000;

        /*
         * Make one copy of ourself.
         * We need two processes to do real damage.
         */
        fork();

        for (;;) {
                gettimeofday(&tv, 0);
                /*
                 * Round down to even tick multiple, then add in bias.
                 * Compute estimate of next hardclock into s and us.
                 */
                s = tv.tv_sec;
                us = tv.tv_usec;
                delta = us % 10000;
                off = bias - delta;
                if (off < 0)
                        off += 10000;
                us += off;
                if (us >= 1000000) {
                        us -= 1000000;
                        ++s;
                }
                /*
                 * Spin until 1ms before next hardclock.
                 */
                us -= 1000;
                while (tv.tv_sec < s || (tv.tv_sec == s && tv.tv_usec < us))
                        gettimeofday(&tv, 0);
                usleep(1);
        }
}
```