Probabilistically Tracking System Calls

Saurabh Goyal, and Akash Lal

Computer Sciences Department
University of Wisconsin, Madison
{saurabh, akash}@cs.wisc.edu

May, 2005

Abstract

Monitoring system calls made by an application is useful for debugging, for diagnostics as well as for security applications. Existing tools for monitoring system calls either suffer from a large runtime overhead, or require root permission to change the kernel of the operating system. We propose an approach that tackles both of these issues. We implemented a tool that runs completely in user space and requires no change to the operating system. It works by probabilistically sampling a few of the system calls made a process, thus lowering its overhead. We show that by using proper sampling techniques, our tool is able to figure out the same information as a tool that tracks all system calls.

1 Introduction

With increasing software complexity, debugging is becoming an important task. There is a need for tools that can help a user understand the execution behavior of a program running on his system. In the absence of detailed understanding, or the source code, of a program, the only way to understand its execution is by monitoring its interaction with the environment. This type of black-box debugging approach, where no assumptions are made about the program has the advantage of being applicable to all software.

In this paper, we describe an approach for monitoring the interaction of a program with the operating system by recording the system calls made by the program. We focus our attention on the Linux operating system as it is commonly used, and is less helpful than other operating systems. Thus, our approach will work on other operating systems as well. There are a host of utilities on Linux that can monitor system calls of a process [3, 4]. However, they either suffer from a large runtime overhead, or from requiring root permission to change the kernel of the operating system. We propose an approach that tackles both of these issues. We have implemented a tool that runs completely in user space without requiring any changes to the operating system. It probabilistically samples a few of the system calls made a process, thus lowering its overhead. We show that by using proper sampling techniques, our tool is able to figure out the same information as a tool that tracks all system calls.

We also show that our tool will especially be helpful in security applications that require continuous system call monitoring. One such application is host-based intrusion detection systems (HIDS). Like black-box debugging, HIDSs guard against security vulnerabilities without making any assumption about the structure of a program. They monitor system calls made by an application and look for deviations from “normal” system calls behavior to detect a compromise in the security of the application.

The rest of this paper is organized as follows. Sec-
tion 2 describes a Linux tool that can monitor system calls in user space. In Section 3 we modify this tool to probabilistically monitor system calls. In Section 4, we present results that show overhead and precision of our tool. The section also explains host-based intrusion detection systems further. Section 5 discusses some of the related work, and Section 6 concludes with some final remarks and future work.

2 STRACE

A common Linux utility used for tracking system calls made by a process is \texttt{strace} [3]. It entirely executes in user-mode, and can be used to attach onto any process running in the same user space. Once attached, it can track all system calls made by the process, and print them out on standard output in human-readable form. \texttt{strace} also builds up a summary of system calls made that maintains information like the number of system calls made, and the time spent per system call type, e.g., it can show the total time taken by all \texttt{read} system calls, and the average time spent per \texttt{read} system call.

\texttt{strace} is based on a Linux system call, called \texttt{ptrace}, which is also the heart of all debuggers in Linux. \texttt{ptrace} is used to attach a parent process onto a running process, such that the parent can observe and control its execution. The rest of this section describes how \texttt{strace} uses the \texttt{ptrace} system call. Once a parent process $P$ is attached onto a child process $C$, it can ask the kernel, via \texttt{ptrace}, to raise a signal (SIGTRAP) whenever $C$ enters and exits a system call. This signal is then caught by $P$, at which point in time the execution of $C$ is stopped. $P$ can now examine the registers and the core image of $C$. After examining the contents of $C$, $P$ can resume the execution of $C$, again via the \texttt{ptrace} system call. The basic structure of \texttt{strace} is shown in Fig. 1, where we assume that signals are only raised because of system calls. The first line attaches \texttt{strace} onto the process with ID \texttt{pid} and becomes its parent process. Inside the loop, at line 3, \texttt{strace} waits for a signal from the child, which it gets once the child makes a system call. \texttt{strace} then examines the register and stack contents of the child to figure out what system call was made, and with what arguments (line 4). The child process is then resumed, after which \texttt{strace} repeats these three steps in lines 6 – 8 to process the exit from the system call. At the exit, \texttt{strace} records the return value, and also the time it took for the system call to complete. This whole process repeats for the next system call.

3 Probabilistic Strace

As is evident from the description of \texttt{strace} presented in Section 2, each system call entry and exit requires two context switches: from the monitored process to \texttt{strace}, and back. This results in a large overhead, which sometimes slows down the monitored process considerably. Some measurements of this overhead are shown in Section 4. The context switches are actually unavoidable if we want to track system calls in user space. Therefore, in order to reduce the overhead, we must decrease the number of system calls monitored, or implement the monitoring

```
(1) ptrace(pid, ATTACH);
(2) while(true) {
(3)  wait4(pid);
(4)  ⟨ process call start ⟩
(5)  ptrace(pid, CONT);
(6)  wait4(pid);
(7)  ⟨ process call end ⟩
(8)  ptrace(pid, CONT);
(9) }
```

Figure 1: The basic structure of \texttt{strace}. The system call \texttt{wait4(pid)} suspends the execution of calling process until a signal is raised by process with ID \texttt{pid}. The call \texttt{ptrace(pid, CONT)} is used to resume the execution of the (stopped) process with ID \texttt{pid}. 
inside the kernel. As mentioned in the introduction, we want a tool that can be used by any user without changing his system. Thus, we discard the latter approach of changing the kernel. The rest of this section describes pstrace (Probabilistic-strace), a modified version of strace that lowers the overhead by a user-defined amount by probabilistically monitoring only some of the system calls.

Suppose that we want pstrace to monitor every $10^6$th system call of a process. One approach would be to monitor a single system call like strace does; detach from the process to let it run freely; sleep until the process makes 9 system calls; and then attach back onto the process to repeat this whole thing again. This would decrease the overhead by 9 times, because we only require 2 context switches for 10 system calls, instead of the 20 context switches that strace required (at the cost of monitoring fewer system calls). However, implementing this approach on top of the Linux operating system is impossible because there is no way of finding out the number of system calls that a process made while pstrace was detached from it. To get around this difficulty, we measure the time that the process took to make a single system call, and then use it to estimate the time it would take the process to make 9 more system calls. pstrace can then sleep for this amount of time, and attach back onto the process with the hope that it made exactly 9 system calls during the time for which pstrace slept.

We now formalize this approach. Suppose that the user wants to measure $s$ percent of the system calls ($0 < s < 100$). Then for each system call made by a process, pstrace should sleep for $n_s = \left(\frac{100-s}{s}\right)$ system calls, i.e., for $s = 25$, pstrace will sleep for 3 system calls for each monitored call. The basic structure of pstrace is shown in Fig. 2. It involves one small, but important change. Instead of monitoring just one system call and then sleeping for $n_s$ system calls, we monitor $b$ system calls and then sleep for $b \times n_s$ system calls. There are two advantages of doing this. First, the smallest time measurement for the user time taken by a process is 10 ms (provided in the /proc file system), which is too large to measure the time taken to make a single system call. A reasonably well chosen value of $b$ (1000 or more) ensures that our time measurements are accurate. The second advantage of having the parameter $b$ is that it allows us to obtain consecutive sequences of system calls. As we shall in Section 4.1, this is essential for security applications to obtain a fair estimate of a process’s behavior. We call $b$ as the burst size. The estimate of time taken by a process to make a system call is calculated as follows:

$$\text{process\_time}(\text{pid}) + \frac{\text{real\_time} - \text{process\_time}(\text{pstrace})}{2} \times b$$

Here, $\text{process\_time}$ is the sum of user and system time taken by a process, and $\text{real\_time}$ is wall-clock time. This formula is based on averaging two estimates of how much processing the process took to make a system call. Each measurement is taken for the time it took to execute the loop in lines 3–6 in Fig. 2. A running average of this time is used to calculate the exact time pstrace will sleep for in line 8 by multiplying it by $\left(b \times \frac{100-s}{s}\right)$.

There is still one deficiency in the pstrace model presented in Fig. 2. Consider monitoring a program that makes 100 read calls followed by 100 write calls, and then 100 more read calls. If we want to monitor 10 percent of the system calls, we would use $n_s = 9$ system calls for each monitored call. This would result in pstrace sleeping for 3 system calls (10 ms), then waking up, then sleeping for 9 more system calls (90 ms), then waking up, and then sleeping for 9 more system calls (90 ms), for a total of 210 ms. However, the process actually made 120 system calls, so pstrace is sleeping for 30 ms more than it should be. To avoid this problem, we can use a different value of $b$ for each system call. In this case, we would use $b = 10$ for the first two system calls and $b = 9$ for the third system call. This would result in pstrace sleeping for 30 ms, then waking up, then sleeping for 90 ms, then waking up, and then sleeping for 30 ms, for a total of 150 ms. This would be a better estimate of the time taken by the process to make the system calls.
where each call takes approximately the same amount of time. If we choose \( b = 100 \), and our timing is accurate enough, we will observe all `read` calls but will entirely miss out on the `write` system calls. This violates our goal of providing a fair estimate of a program’s behavior. We use randomization to solve this problem.

Consider a sequence of events (in our case, events are system calls). We want to observe each event with an equal probability, say \( p (0 < p < 1) \). Then the following derives the probability of skipping the first \( n \) events, and measuring \((n + 1)^{st}\) event:

\[
\begin{align*}
Pr(\text{Measuring 1}^{st} \text{ event}) &= p \\
Pr(\text{Skipping 1 event, Measuring 2}^{nd} \text{ event}) &= (1 - p) \times p \\
Pr(\text{Skipping 2 events, Measuring 3}^{rd} \text{ event}) &= (1 - p)^2 \times p \\
&\vdots \\
Pr(\text{Skipping n events, Measuring (n + 1)}^{st} \text{ event}) &= (1 - p)^n \times p
\end{align*}
\]

The above is a geometric distribution with mean \((1/p)\). Therefore, instead of sleeping for a fixed number \((b \times \frac{100-s}{s})\) of system calls, we should sleep for \( k \) system calls, where \( k \) is a random number obtained from a geometric distribution with mean \((b \times \frac{100-s}{s})\). This gives us two probabilistic guarantees: First, on an average, \( p_{\text{strace}} \) will sleep for \((b \times \frac{100-s}{s})\) number of system calls; and second, each system call is measured with an equal probability. These guarantees are based on the assumption that the timing measurements are accurate. In practice, the variation in timing is also a source of randomness, but is not enough to ensure fair system call estimates, as we shall see in Section 4. A similar strategy of sampling events is used in program analysis as well, where the objective is to sample certain predefined runtime events, instead of system calls [8]. The final code structure of \( p_{\text{strace}} \) is shown in Fig. 3.

### 3.1 Perfect Knowledge \( p_{\text{strace}} \)

In order to measure the accuracy of \( p_{\text{strace}} \), we modified the Linux kernel\(^1\) to provide a per-process measure of the number of system calls the process made since it was started. We used this measure to rewrite \( p_{\text{strace}} \) and create \( k_{\text{strace}} \) (Kernel-based-probabilistic-strace), which also runs as a user-process but has a much better estimate of the time to sleep for while detached from the monitored processes. Instead of sleeping, \( k_{\text{strace}} \) could have spin-locked on the system call number to start monitoring as soon as the right number of system calls have taken place, but this would incur an unnecessary overhead. Thus, we use same time estimates that \( p_{\text{strace}} \) uses to calculate sleep-time, but we use the system-call-number information to resolve inaccuracies in such timing measurements.

After waking up \( k_{\text{strace}} \) looks at the number of system calls made while it was sleeping. If this number, say \( n_1 \), was different from the required number \( n_2 = (b \times \frac{100-s}{s}) \) of system calls it wanted to sleep for, then it adjusts the sleep time by multiplying it with \((n_2/n_1)\). This approach will dynamically try to

---

1 Appendix A describes the changes.
approach the correct sleep time value. As a fall-back
guard, kstrace also maintains two global counts: the
system calls it has monitored ($g_1$), and the total num-
ero of system calls the process has made ($g_2$), the
latter of which is simply obtained from the modified
kernel. If the lag in system calls monitored, defined
as ($g_2 \times s/100) - g_1$), exceeds twice the burst size,
kstrace decides not to sleep at all and continues to
monitor the next 6 system calls. This ensures that
kstrace is never off its target by much.

We present a detailed evaluation of kstrace along
with that of pstrace as it illustrates the improvement
in performance of pstrace when the kernel is more
helpful that what the Linux kernel is currently.

4 Evaluation

For probabilistic tracing of system calls, there are two
major criteria that we have to evaluate the output
on. The first is overhead of the probabilistic tracing
and the second is how well the output statistics
match the true statistics. In this section we show the
results for both the above criteria. We also present
an experimental result comparing the reliability of
pstrace with kstrace. The last experimental result
will show the effect of using the geometric distribu-
tion for calculating the sleep time.

We evaluated our tools based on three programs,
postmark [2], which is a standard filesystem bench-
mark, thttpd [5], which is a simple web server and
a toy filesystem program Toy-fs that does a read(),
write() and lseek() in a loop. These programs were
first run standalone and then attached to the original
strace program. Then, we ran the programs with pstrace and kstrace, four times each. The burst size for postmark and thttpd was fixed to 1000,
and the burst size for Toy-fs was fixed to 10000. The
parameter for percentage of system calls monitored
was set to 50%, 25%, 10%, and 1% in the four runs
respectively.

Fig. 4 shows the overhead of tracking system calls
for the postmark benchmark. The original program
ran in 5.62 seconds. The total number of system calls
made by the program are 517K. Tracking all these calls
using strace took 15.05 seconds. As we reduce the
number of system calls traced, the execution time
goes down linearly. For tracing 50K calls the time
taken is 6.55 seconds. The time reduces further, but
as we show in other graphs, the accuracy of the out-
put below this point suffers. Fig. 5 shows the same
graph for Toy-fs. It also shows a linear decrease in
execution time. The standalone program takes 10.12
seconds.

The table below compares the reliability of
kstrace with pstrace. The first column is the
requested sampling percentage and the other two
columns show how many calls were actually tracked.
This is for the postmark program. The total number
of system calls made by the program are 517K.
The table shows that \textbf{kstrace} tracks almost the same percentage of system calls that it is asked to, whereas \textbf{pstrace} is slightly off the asked percentages. This difference was more for the toy program. Out of a total of 6 million calls, the following is what was traced.

<table>
<thead>
<tr>
<th>Sample %</th>
<th>pstrace</th>
<th>kstrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>194K</td>
<td>255K</td>
</tr>
<tr>
<td>25</td>
<td>120K</td>
<td>127K</td>
</tr>
<tr>
<td>10</td>
<td>63K</td>
<td>50K</td>
</tr>
<tr>
<td>1</td>
<td>7K</td>
<td>5K</td>
</tr>
</tbody>
</table>

The table shows that \textbf{kstrace} tracks almost the same percentage of system calls that it is asked to, whereas \textbf{pstrace} is slightly off the asked percentages. This difference was more for the toy program. Out of a total of 6 million calls, the following is what was traced.

<table>
<thead>
<tr>
<th>Sample %</th>
<th>pstrace</th>
<th>kstrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>0.8M</td>
<td>3M</td>
</tr>
<tr>
<td>25</td>
<td>0.63M</td>
<td>1.48M</td>
</tr>
<tr>
<td>10</td>
<td>0.28M</td>
<td>0.57M</td>
</tr>
<tr>
<td>1</td>
<td>0.07M</td>
<td>0.04M</td>
</tr>
</tbody>
</table>

Again, \textbf{kstrace} follows the requested percentages closely. \textbf{pstrace} monitors less system calls than requested because the system call rate of Toy-fs was very high, which made it overestimate the sleep-time.

The next table shows the accuracy of the output statistics. When the postmark program is run with \textbf{strace}, most of the time is spent in the system calls \texttt{open()}, \texttt{write()}, and \texttt{unlink()}. The percentage of time spent in these calls is 27%, 22%, and 19% respectively. The table below shows how closely these percentages match when all the calls are not traced.

<table>
<thead>
<tr>
<th>Sample %</th>
<th>pstrace, kstrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>50</td>
<td>\texttt{open()}: 27, 36</td>
</tr>
<tr>
<td>25</td>
<td>\texttt{open()}: 27, 26</td>
</tr>
<tr>
<td>10</td>
<td>\texttt{open()}: 27, 27</td>
</tr>
<tr>
<td>1</td>
<td>\texttt{open()}: 26, 20</td>
</tr>
</tbody>
</table>

The time statistics seem to be reasonably close for sampling percentages as low as 10%. For 1% sampling, the accuracy is a bit off, in fact the order of percentage time spent has changed. More time is shown for \texttt{write()} than for \texttt{open()}. In other experiments, the accuracy for the toy filesystem program was good only for 50% sampling, while for the thttpd program, it was good down to 1%. One thing to note here is that the actual number of system calls tracked was not the same for \textbf{kstrace} and \textbf{pstrace}, these numbers have been reported earlier.

The next experiment shows that using the geometric distribution to calculate the sleep-time results in improved accuracy at lower sampling rates. We show the comparison on the thttpd program. We made a workload of simple HTTP GET requests and traced the webserver’s system calls during that workload. The webserver makes 270K calls during this time, and most time is spent in the system calls \texttt{poll()}, \texttt{mmap2()}, and \texttt{send()}. The actual percentage time spent in these calls is 84%, 11% and 3% respectively.

The first table, here, shows the percentage time spent when the sleep time was just a constant times the burst size (Fig. 2).

<table>
<thead>
<tr>
<th>Sample %</th>
<th>pstrace, kstrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>\texttt{open()}: 47K, 33K</td>
</tr>
<tr>
<td>1</td>
<td>\texttt{open()}: 6K, 4K</td>
</tr>
</tbody>
</table>

We see that the accuracy at 1% sampling is off the original values. The same figures when taken after modeling the sleep time using a geometric distribution are shown in the following table. The values at 1% sampling here, are closer to the original values.

<table>
<thead>
<tr>
<th>Sample %</th>
<th>pstrace, kstrace</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>\texttt{open()}: 43K, 33K</td>
</tr>
<tr>
<td>1</td>
<td>\texttt{open()}: 5K, 4K</td>
</tr>
</tbody>
</table>

While not exhaustive, these results demonstrate that probabilistic tracing of system calls can result in significant reduction in overhead without much loss in the usability of its output. They also demonstrate the improvements made by making the kernel give system call information and by using a geometric distribution for calculating the sleep time.

### 4.1 Host-Based Intrusion Detection

In this section, we briefly explain the utility of \textbf{pstrace} in security applications. As software is getting more complicated, it is increasingly common to
find applications that have security flaws in them. For example, wu.ftpd (Washington University ftpd 2.4), a ftp daemon, if misconfigured at compile time, allows users SITE EXEC access to /bin. Users can then run executables such as bash with root privilege. Such flaws, until discovered, pose a big risk to the system using these applications, especially when the applications interface with untrusted users on the Internet. Examples of such applications are Web-servers, email clients and ftp clients [1].

Host-based intrusion detection systems (HIDS) are used by system administrators to guard against unknown vulnerabilities in software. One way in which they are used is the following [9]: First, the potentially-vulnerable application is used by trusted users. HIDS monitors the system calls made by the application under such usage, and builds a model of the system calls that the application should make. This model is typically built from consecutive system call sequences made by the application. In the second stage, the application is put on the Internet, where untrusted users can use the application. The system calls are still monitored, but are checked against the model built in the previous stage. If in a short period of time, many system call sequences do not match any in the model, the application is stopped. The intuition is that if there is a break-in into the application, the behavior of the application will change substantially so that the system calls sequences made will differ from those obtained under valid executions of the application. Thus, a deviation from the normal system calls that the application makes, means a potential attack is taking place.

HIDS presents an interesting scenario for using pstrace. HIDS require continuous monitoring of system calls, and overhead must be small enough to avoid affecting the performance of the application in question. We can use pstrace to sparsely sample the system calls made by the application with low overhead. As soon as some system call sequence does not satisfy the system-call-model, we can increase the sampling rate to monitor more system calls till the point we are convinced if this is an attack (more system call sequences do not match the model) or not (other sequences match the model). With the assumption that attacks are rare, pstrace will run with very little overhead most of the time.

We have not used pstrace inside a HIDS, because we were not able to simulate real attack scenarios. However, we ran pstrace on postmark and recorded all system calls. Then, we used a tool called stide [1] to build a database of all contiguous sequences of system calls of length 6. We found that out of 480K system calls made by postmark, there were only 159 unique sequences. When we sampled down the measured system calls to 30K, we still obtained 125 sequences. This shows that there is a high redundancy in the system call sequences made by a process, i.e., the same system call sequence is made over and over again. Thus, pstrace misses very few system call sequences.

5 Related Work

The idea of sampling program execution behavior is common in dynamic program analysis [8, 6]. The goal there is to monitor the execution of a program by recording the values of program variables at certain program points. Instead of recording the program state after each instruction, a sampling strategy is used to monitor only a few locations. After recording such sampled data, certain properties of the program can be inferred. Like pstrace, the effort is to have a sampling strategy that provides a fair estimate of the program’s execution. Unlike program variables, tracking system calls does not require any semantic knowledge about the program.

The systems community also makes use of sampling to figure out program behavior. The VMware ESX server [10] samples the memory pages used by a client operating system to estimate the idle memory of the operating system. This estimate is used in the page-replacement policy, where higher preference is given to the pages of an operating system with greater amount of idle memory. The Lottery Scheduler [11] uses a randomized selection policy. The guarantees of fairness and proportional sharing it provides are probabilistic. Anticipatory scheduling [7] makes scheduling decisions based on an estimate of the time it would take for a process to make the next request. This is similar to the sleep-time estimate...
that `pstrace` makes for time till the next system call.

Besides `strace`, another tool used for monitoring system calls is `syscalltrack` [4]. It is not based on the `ptrace` system call, but it *hijacks* the system call table inside the kernel to call its own functions before passing control onto the actual system call handlers. This tool can only be used by a root user, as changing the system call table requires root permission. However, monitoring system calls only requires an extra function call instead of a context switch. This means that `syscalltrack` will run with a very small overhead. One can envision implementing the sampling strategy of `pstrace` inside `syscalltrack` to further reduce its overhead in performance critical applications.

6 Conclusions and future work

In this paper we have shown the utility of `pstrace`, a probabilistic system call monitoring tool, as a low overhead alternative to `strace`. We have presented results that show that overhead can be controlled by the user by modifying the sampling rate of `pstrace`. We also show that a decrease in the sampling rate is not necessarily accompanied by a loss in precision. The system call estimates, in the form of percentage time taken per system call type, remain fairly accurate in most scenarios. Moreover, the system call sequences captured at low sampling rates also faithfully model the actual system call sequences. This motivates the use of `pstrace` in security systems that require such system call information, and also have a great need for low-overhead continuous system call monitoring.

As future work, one can add a recent-time-window summary of system calls. This will include timing information of recent system calls, and will incrementally forget timing information of previous system calls. The benefit of having such a summary is that it reflects the *current* interaction of an application with the operating system. This is useful for diagnostic purposes. In case the network or the file system slows down, the recent summary will show the increased time for system calls made to those devices.

A limitation of `pstrace` is that it calculates sleep-time based on the estimates obtained from timing previous system call behavior regardless of what part of a process is execution. Some parts of a process might make system calls at a faster rate than other parts. Thus, a better approach for `pstrace` would be to track the sleep-time relative to the program-counter of a process, such that fewer calls are monitored from regions with high system call rate, and more calls are monitored from regions with low system call rate. This type of sampling strategy is followed in [5].

The current implementation of `pstrace` lacks the ability to follow forks in a process. Monitoring multiple processes using one instance of `pstrace` would require implementing an event-driven loop, instead of a simple loop that sleeps when detached from the monitored process. The current implementation can still be used by manually invoking `pstrace` to track child processes.

References

In this section, we describe some of the changes we made to the Linux kernel version 2.4.27 to make it record the number of system calls made by a process in its lifetime. The kernel maintains a data-structure of type `task_struct`, declared in `include/linux/sched.h`, for each process running on the machine. We modified this structure to include an extra field (`syscalls`) that counts the system calls made by the process. The macro `INIT_TASK`, declared in the same file, is used to initialize the task structure for a new process. This was changed to set `syscalls` to zero.

The next step was to change `arch/i386/kernel/entry.S` assembly file (for i386 architecture). It contains the piece of code that a process traps to whenever it makes a system call. We added a call to a new function at this point that increments the `syscalls` of the current process. These changes together keep a perfect account of the system calls made by a process. The next change was to report this value to the user. This was accomplished via the `/proc` file system. The files `fs/proc/base.c` and `fs/proc/array.c` were changed to create an extra file in `/proc/pid/` directory that reported the system call count.