Analysis of the process lifetime

Jakub Bittara[1,*], Dušan Bernát[1]

[1]Faculty of Mathematics, Physics and Informatics, Comenius University, Mlynská dolina, Bratislava, Slovakia

Abstract

Processes are a fundamental abstraction of modern operating systems and are responsible for executing user code. The execution characteristics of a process provide valuable information not only about the process itself–such as whether it terminated successfully, failed, stalled, or diverged from its expected execution path–but also about the overall state of the system. In this paper, we present a tool that utilises the eBPF subsystem to collect information about every process at the instant of its termination. The collected data include execution time, CPU user and system time, wall-clock time, exit code, and other execution-related metrics. These data are subsequently analysed to estimate execution time distribution and other statistical characteristics that can be used to detect changes in system behaviour and identify anomalies.

Keywords

process, eBPF, execution time, distribution, CPU time, anomaly detection

1. Introduction

Processes are managed by the operating system. In general, the number of processes on a system can exceed the number of available CPU cores. To enable context switching, the operating system saves the execution state of a process to memory so that it can later resume its execution. This state is stored in the Process Control Block (PCB), which is represented in Linux by the `task_struct` structure. The `task_struct` contains information such as the process name, `PID`, `utime`, parent process, owner `UID`, CPU register state, open files, and many other attributes. Threads are represented by the same `task_struct` structure and therefore are treated in the same way as processes in our analysis.

In Linux, new processes are created by a parent process using the `fork` or `clone` system calls, or their related variants `vfork` and `clone3`. The first process in the system (traditionally called `init`) is created directly by the kernel. A common process creation pattern is to call `fork`, which creates a copy of the parent process, followed by `execve`, which replaces the process image with the contents of an executable file.

A process terminates by calling the `exit` system call after completing its execution. It can also be terminated by a signal. This occurs either when the signal cannot be caught or ignored (e.g., `SIGKILL`), or when the default action associated with the signal is to terminate the process.

In Section 2.1 we describe the design and implementation of an eBPF-based tool for recording information related to the termination of processes, including their real elapsed time and CPU time consumed. We then analyze the collected data. In section 3.2 we present the results of statistical distribution tests, which indicate that process lifetimes tend to follow a power-law distribution, in particular, a Type II Pareto distribution. Following sections demonstrates some application of collected processes lifetime data. Section 3.3 presents a method for estimating the type of system load. Finally, Section 3.4 outlines a method for detecting SSH brute-force attacks based on the real-time analysis of process-lifetime statistics.

1.1. Extended Berkeley Packet Filter (eBPF)

The extended Berkeley Packet Filter (eBPF) is a Linux kernel technology that allows programs to be loaded and executed dynamically in kernel space without modifying the kernel source code or loading a

ITAT'26: Information Technologies – Applications and Theory, September 25–29, 2026, Biele Karpaty, Slovakia

*Corresponding author.

bittara3@uniba.sk (J. Bittara); bernat@fmph.uniba.sk (D. Bernát)



© 2026 Copyright for this paper by its authors. Use permitted under Creative Commons License Attribution 4.0 International (CC BY 4.0).

CEUR Workshop Proceedings ceur-ws.org ISSN 1613-0073

published 2026-11-09

CEUR-WS.org/Vol-4323/paper42.pdf

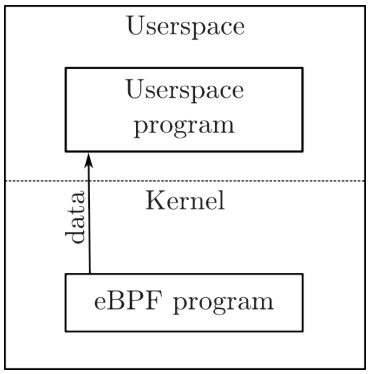


Figure 1: Location of eBPF and userspace programs in the memory space of the operating system.

custom kernel module. This enables the development of efficient networking, observability, debugging, and security tools while minimizing the impact on the running system [1].

The eBPF technology evolved from the original Berkeley Packet Filter (BPF), introduced for efficient packet filtering [2]. The original BPF architecture defined a simple virtual machine that executed filter programs to determine whether network packets should be accepted or rejected. These programs used a compact instruction set similar to assembly language. Compared to the original BPF, eBPF significantly extends the architecture by increasing the number of registers from 2 to 10, expanding the register width from 32 to 64 bits, introducing the `bpf` system call for communication with user-space applications, and adding the eBPF verifier, which checks programs for safety before they are loaded into the kernel. Once verified, eBPF programs are interpreted or, on most systems, just-in-time (JIT) compiled into native machine code for efficient execution.

Multiple program types and attachment mechanisms are supported. In this work, we use programs from the `tracing` category to monitor kernel events and transfer the collected information to user space. Specifically, we use the `kprobe`, `tracepoint`, and `fentry`/`fexit` attachment points, which are briefly described below.

- `kprobes` – allow dynamic instrumentation of kernel functions without modifying the kernel source code. A `kprobe` can be attached to almost any instruction in the kernel, and a user-defined handler is executed when the probe is triggered [3].
- `tracepoints` – are statically defined instrumentation points embedded in the kernel source code. eBPF programs can be attached to these points at runtime. Unlike `kprobes`, tracepoints provide a stable interface across kernel versions as long as the tracepoint is present in both versions [4].
- `fentry`/`fexit` – attach eBPF programs to the entry and exit of kernel functions. They provide lower overhead and better performance than `kprobes` because they use the kernel function tracing infrastructure. These attachment points are available in Linux kernel version 5.5 and later [5].

eBPF maps are kernel-resident data structures that enable communication between eBPF programs and user-space applications. Common map types include arrays, hash maps, ring buffers, and Bloom filters. User-space applications access and modify these maps through the `bpf` system call.

2. eBPF tool

We developed an eBPF-based tool that collects information about every terminating process in the system. It is composed of two communicating programs. The first one is an eBPF program attached to a kernel tracing attachment point, which collects process information in kernel space. The second program runs in userspace and is responsible for receiving data from the eBPF program and produces output in CSV format. Figure 1 shows the basic setup.

Figure 2: Output records example.

```
start timestamp: Sun May 24 15:24:00 2026
system info: Linux ubuntu 5.15.0-67-generic #74~20.04.1-Ubuntu...
TIME                PCOMM           PID     age     utime   exit
1779636241.186008   pool-tracker-ex 7890    10.387  0.000   0
1779636241.190735   gmain           7886    10.396  0.000   0
1779636241.191042   gdbus           7887    10.393  0.008   0
1779636246.276326   gnome-shell     7912    0.005   0.000   0
1779636246.497459   basename        7923    0.001   0.000   0
1779636246.502161   dirname         7925    0.001   0.000   0
1779636246.502181   lesspipe        7924    0.001   0.000   0
1779636246.503375   lesspipe        7922    0.008   0.000   0
1779636246.504921   dircolors       7926    0.001   0.000   0
1779636246.516159   gdbus           7917    0.150   0.004   0
1779636246.516177   gmain           7916    0.151   0.000   0
1779636246.516182   dconf worker    7918    0.057   0.000   0
1779636246.519200   gnome-terminal  7915    0.178   0.088   0
1779636246.527918   gnome-terminal  7913    0.254   0.048   0
1779636249.776216   bash            7927    0.045   0.000   127
1779636259.879694   python3         7929    7.951   0.012   0
1779636261.684861   bash            7921    15.200  0.008   0
^C
Handled 17 processes.
end timestamp: Sun May 24 15:24:22 2026
```

2.1. eBPF program

The eBPF program is attached to the `sched_process_exit` tracepoint and is executed whenever a process terminates. It obtains a pointer to the `task_struct` of the terminating process using the `bpf_get_current_task` helper. Selected fields, including the `PID`, `UID`, `utime`, `stime`, and others are then read from the structure using the `bpf_probe_read_kernel` helper. The collected data are written to a `BPF_MAP_TYPE_PERF_EVENT_ARRAY` map, from which they are received by the user-space application.

2.2. Userspace program

The user-space program first loads the eBPF program into the kernel and then receives events through the eBPF map. The received data are processed and written to the output in selected format. Loading an eBPF program into the kernel requires elevated privileges. Therefore, the application must be executed either as `root` or with the capabilities assigned using `setcap` (`CAP_BPF` and `CAP_PERFMON`, if available, or `CAP_SYS_ADMIN` on older kernels).

After loading the eBPF program, the application prints the current date and time, system information obtained using the `uname` system call, and the names of the process attributes selected through the command-line arguments. For each terminating process, it outputs a single line record containing the selected attributes, separated by tab characters by default or by commas when CSV output is requested. Example output can be seen in the Figure 2.

The application terminates upon receiving the `SIGINT` or `SIGTERM` signal. Before exiting, it stops receiving events from the eBPF program, prints the total number of processed processes together with the current date and time, and unloads the eBPF program from the kernel.

Table 1
Collected datasets.

File	Processes	Time span	Size	Usage
A	150190	22 days	13 MB	passive
B	2470814	40 days	234 MB	passive/computing
C	83749482	17 days	7.5GB	computing

Table 2
Top 5 processes from dataset A.

Process name	count	[%]
postmaster	69248	46.11
sshd	55884	37.21
unix_chkpwd	12042	8.02
kworker/dying	4006	2.67
dnf	1450	0.97

Table 3
Top 10 processes from dataset B.

Process name	count	[%]
sshd	516651	20.91
bash	403932	16.35
cat	201814	8.17
wc	200106	8.10
filter_graphs	200038	8.10
unix_chkpwd	134255	5.43
kissat	121646	4.92
postmaster	118521	4.80
rm	81662	3.31
python3	81428	3.30

Table 4
Top 5 processes from dataset C.

Process name	count	[%]
kissat-sc2024	83585883	99.80
sshd	71028	0.08
postmaster	62103	0.07
unix_chkpwd	12745	0.02
kworker/dying	4912	0.01

3. Resulting data analysis

To analyze the collected data, we developed a statistical analysis tool in the R programming language [6]. The tool processes log files generated by the eBPF monitoring tool. In addition to computing descriptive statistics, such as the mean, median, and standard deviation of selected process attributes (e.g., CPU time), it estimates the probability distribution that best fits the observed data. It can also identify the most frequently terminated processes within a specified time interval and detect anomalies in the number of process terminations.

Three datasets of terminated processes were collected using the eBPF monitoring tool. Their characteristics are summarized in Table 1. All datasets were captured on the same machine during normal operation. The system was a Linux host accessible via `ssh` from the Internet, running a `PostgreSQL` database together with other common system services, while also being used for CPU-intensive computations.

3.1. Processes summary

To obtain an overview of the processes running on the monitored system, we computed the overall frequency of terminated process names. The results for datasets A, B, and C are summarized in Tables 2, 3, and 4, respectively. Dataset A is dominated by the `postmaster` and `sshd` processes. In addition to these processes, dataset B contains computational applications such as `filter_graphs` and the SAT solver `kissat`. Dataset C is dominated by very frequent relatively short SAT solver `kissat-sc2024` processes.

These differences suggest that the relative frequencies of `sshd` and `postmaster` process occurrences can be used as indicators of the system usage mode. In particular, dataset C, which represents a machine heavily loaded by numerous short-running `kissat-sc2024` computations, is clearly distinguishable from the other two datasets. In contrast, dataset A corresponds to an otherwise lightly loaded system, whereas dataset B represents a system executing longer-running, more CPU-intensive `kissat` computations. On the other hand, per day average numbers of terminated `sshd` and `postmaster` processes seems not to be preserved across different system usage modes.

Number of terminated processes can be observed as a real time metric when evaluated on the fly for

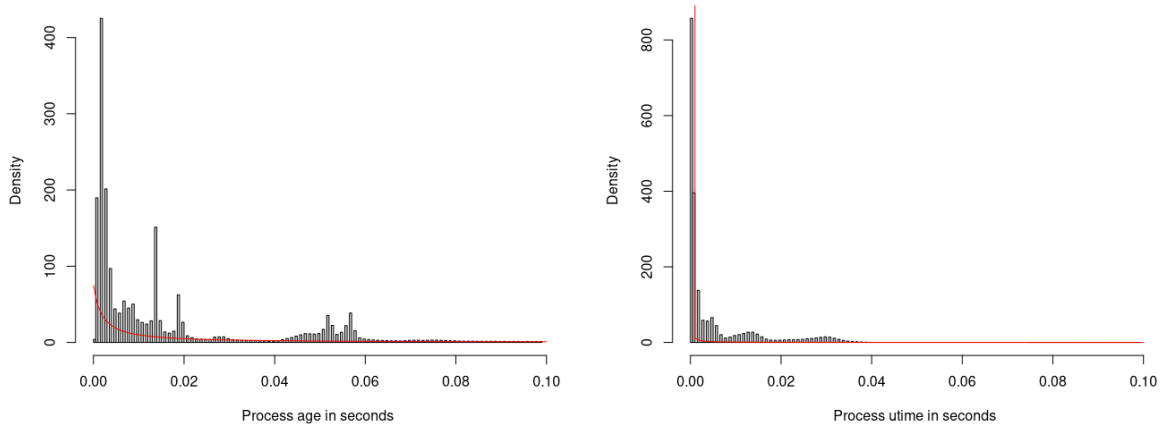


Figure 3: Histogram of process age and CPU time from dataset B with corresponding estimated distribution.

Table 5
Estimated parameters of the fitted Pareto distribution of the second kind for process age and CPU user time.

File	Process age			CPU user time		
	μ	α	AIC	μ	α	AIC
A	0.006861889	0.2747367	285371	0.01213263	2.269046	-1138587
B	0.002150038	0.1600382	14521368	2.220446e-16	0.05188711	-63313909
C*	0.1326968	1.343293	-1140742	0.1409453	3.458188	-3821915

some time window. Results for one hour intervals can be found in Figure 4 for dataset B.

3.2. Estimating process lifetime distribution type

We analyzed the distributions of process lifetime (wall-clock age) and consumed CPU user time. To identify the distribution that best fits the observed data, we used the `gamlss` package for the R programming language [7]. The goodness of fit was evaluated using the Generalized Akaike Information Criterion (GAIC) [8], where smaller values indicate a better fit between the measured data and the assumed probability distribution.

Among the distributions evaluated by `gamlss`, the type II Pareto distribution consistently provided the best fit for all three datasets, regardless of the computational workload. Its probability density function is given by

$$f(y|\mu,\alpha) = \frac{\alpha\mu^{\alpha}}{(y+\mu)^{\alpha+1}}$$

where α is scale and μ is the positional parameter.

This result is consistent with previous studies reporting Pareto-like distributions of Unix process lifetimes [9]. An example histogram of age and CPU time for dataset B can be seen in Figure 3.

The estimated distribution parameters are summarized in Table 5. For computational reasons, only the first 10^6 process records from dataset C were used for distribution fitting, as evaluating all candidate distributions on the complete dataset was prohibitively time-consuming. The estimated parameters vary across workloads and may therefore serve as indicators of changes in system behavior or anomalous operating conditions.

Distribution parameters, as well as the distribution type itself, can be used as indicators for mode change or anomaly detection. However, this procedure can be too computationaly demanding, thus these values seems not to be appropriate candidates for detection systems working in real time.

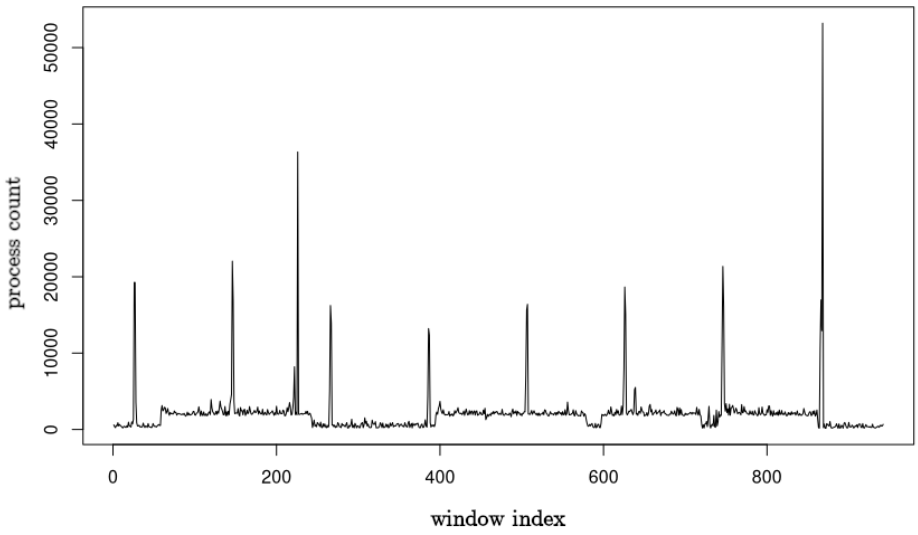


Figure 4: Number of processes terminated in disjoint time intervals of 3600 seconds in file B.

3.3. I/O vs CPU processes

To distinguish between I/O-bound and CPU-bound processes, we define the utilization ratio

$$r = \frac{\texttt{utime} + \texttt{stime}}{\texttt{age}},$$

where `utime` is the CPU time spent executing user-space code, `stime` is the CPU time spent executing kernel code, and `age` is the wall-clock lifetime of the process. The ratio expresses the fraction of the process lifetime during which the process was actively executing on a CPU.

Processes with values of r close to 0 spend most of their lifetime waiting for external events, such as I/O operations, and are therefore classified as I/O-bound. Conversely, processes with values of r close to 1 spend most of their lifetime executing on the CPU and are classified as CPU-bound.

Please note that this metric should be regarded as a heuristic rather than an exact classification criterion. For example, a CPU-intensive process that remains idle after completing its computation will have an increased wall-clock lifetime, reducing the value of r. Similarly, under heavy system load, CPU-bound processes may spend a considerable amount of time waiting to be scheduled, which also decreases the ratio. On the other hand, very short-lived processes may exhibit values of r close to 1 simply because they experience little waiting time, making it difficult to distinguish between CPU-bound and I/O-bound behavior based solely on this metric. Incorporating additional information, such as the overall system load, could improve the accuracy of the classification.

The relative proportions of I/O-bound and CPU-bound processes can be evaluated in real time over a fixed-length time interval. Significant change in this metric may indicate a change in the system usage mode. If such a change is unexpected, it may also indicate a potentially harmful anomaly.

3.4. SSH attack

All datasets contain a continuous occurrence of terminated `sshd` processes. We suppose that at least part of this activity is caused by automated `ssh` login attempts, such as brute-force attacks. To investigate this assumption, we simulated a brute-force attack by generating multiple unsuccessful `ssh` connection attempts. We then analyzed the resulting sequences of terminated `sshd` processes and compared them with the sequences observed in the recorded datasets to determine whether successful and unsuccessful login attempts can be distinguished.

To simulate the attack, we developed a Python program using the `pwntools` library [10], which provides a convenient interface for automating network interactions. The program repeatedly attempts to establish an `ssh` connection using randomly generated usernames and passwords:

```
from pwn import ssh, context
# pwntools uses paramiko for ssh connections
from paramiko import AuthenticationException
from random import choice
import string

context.log_level = 'error'

def random_string():
    return ''.join(choice(string.ascii_letters) for _ in range(8))

conn = None
try:
    conn = ssh(
        host='localhost',
        user=random_string(),
        password=random_string(),
        port=22
    )
except AuthenticationException as e:
    pass # unsuccessful login
except Exception as e:
    print(e)
finally:
    if conn is not None:
        conn.close()
```

After simulating several unsuccessful login attempts, we can see that for each login attempt, two `sshd` processes were terminated immediately:

PCOMM	PID	PPID	UID	age	exit
sshd-session	2831754	2831752	74	2.339	255
sshd-session	2831752	1210	0	2.348	5

This sequence occurs frequently in logs A, B and C created by the eBPF tool. Then we manually simulate a successful attempt and compare logs from eBPF tool:

PCOMM	PID	PPID	UID	age	exit
sshd-session	3185837	3185836	74	2.457	0
sshd-session	3185841	3185836	1000	1.077	255
sshd-session	3185836	1210	0	3.574	255

Compared to an unsuccessful login attempt, a successful login produces a characteristic sequence of three terminated `sshd` processes. The first two processes terminate immediately after authentication, whereas the third terminates only after the user logs out. Another distinguishing feature is the exit code of the process running with `UID` equal to 0. These observations provide a reliable way to distinguish successful and unsuccessful `ssh` login attempts.

The proposed approach is not specific to `sshd`. It can be applied to any event that produces a characteristic sequence of process terminations observable by the eBPF monitoring tool.

4. Conclusion

We developed an eBPF-based monitoring tool that records information about terminating processes and deployed it on a production Linux system. The tool operated reliably for several weeks under different

workload conditions, producing three datasets for further analysis while introducing negligible runtime overhead.

To analyze the collected data, we implemented a statistical tool in the R programming language. It computes descriptive statistics, estimates probability distributions of selected process attributes, and evaluates process termination rates over time. Our analysis showed that process lifetimes are best described by the Pareto distribution of the second kind across all evaluated workloads. We also proposed a simple method for classifying processes as I/O-bound or CPU-bound based on the time spent in user space, kernel space, and the overall process lifetime.

The collected datasets were further used to identify and analyze anomalous system behavior. In particular, the frequent occurrence of `sshd` processes motivated a detailed analysis of their termination patterns. Based on the observed sequences of process terminations, we proposed a method for distinguishing successful and unsuccessful `ssh` login attempts. More generally, the same approach can be applied to detect other events that produce characteristic sequences of process terminations.

The current implementation performs statistical analysis offline after the data have been collected. Although the eBPF monitoring tool is suitable for continuous deployment in production environments, the analytical component should be viewed as a proof of concept demonstrating several metrics that could serve as inputs to a real-time anomaly detection or intrusion detection system. Future work will focus on online processing of collected events and evaluating the proposed metrics in a broader range of workloads and security scenarios.

Declaration on Generative AI

During the preparation of this manuscript, the authors used ChatGPT (OpenAI) to improve the grammar, style, and clarity of the English text. The authors reviewed and edited all generated content and take full responsibility for the final manuscript.

Acknowledgments

This publication is the result of support under the Operational Program Integrated Infrastructure for the project: Advancing University Capacity and Competence in Research, Development and Innovation (ACCORD, ITMS2014+:313021X329), co-financed by the European Regional Development Fund.

References

[1] L. Rice, Learning eBPF, O'Reilly Media, Inc., 2023.

[2] S. McCanne, V. Jacobson, The bsd packet filter: A new architecture for user-level packet capture (December 19, 1992). https://www.tcpdump.org/papers/bpf-usenix93.pdf.

[3] J. Keniston, P. S. Panchamukhi, M. Hiramatsu, Kernel probes (kprobes), n.d. https://docs.kernel.org/trace/kprobes.html.

[4] M. Desnoyers, Using the linux kernel tracepoints, n.d. https://docs.kernel.org/trace/tracepoints.html.

[5] T. kernel development community, Fprobe - function entry/exit probe, n.d. https://docs.kernel.org/trace/fprobe.html.

[6] R Core Team, R: A language and environment for statistical computing, 2019. URL: https://www.R-project.org/.

[7] R. A. Rigby, D. M. Stasinopoulos, Generalized additive models for location, scale and shape,(with discussion), Applied Statistics 54 (2005) 507–554.

[8] J. D. Mitchell, E. S. Allman, J. A. Rhodes, A generalized aic for models with singularities and boundaries, 2022. URL: https://arxiv.org/abs/2211.04136. arXiv:2211.04136.

[9] H.-B. Mor, A. B. Downey, Exploiting process lifetime distributions for dynamic load balancing, ACM SIGMETRICS Performance Evaluation Review 24 (1996) 13–24. URL: https://cir.nii.ac.jp/crid/1360855569013803776. doi:10.1145/233008.233019.

[10] Gallopsled and contributors, pwntools: Ctf framework and exploit development library, https://github.com/Gallopsled/pwntools, 2015.