Section 4. Engineering sciences in general

DOI:10.29013/ESR-26-3.4-36-43

ADVANCED LINUX PROFILING IN A TUI

Alice Rogers [1]

[1] Independent researcher

Cite: *Rogers A. (2026). Advanced Linux Profiling in a TUI. European Science Review 2026, No 3–4.* https://doi.org/10.29013/ESR-26-3.4-36-43

Abstract

As software complexity increases, performance profiling remains an essential practice for identifying CPU bottlenecks and optimizing resource allocation. Flamegraphs (Gregg, B., 2016) have emerged as a near-universal visualization tool for this purpose. The data used to make a flamegraph may be large and is traditionally rendered as a web page or as a GUI component within an IDE. When development is split between a local development machine and a remote cloud server, the need to copy profile data is slow and burdensome.

At the same time, with cloud development being a new normal, there has been a renaissance in using the command line. The command line is well-suited to the new AI chat-based tools (Agarwal et al., 2020). The console for the command line has become richer, supporting broad color palettes and mouse interaction. To use the new rich console environment, Text-based User Interface (TUI) libraries have appeared, mirroring user interface development for modern web applications.

This work presents the combination of advanced profiling with modern TUI design, removing the need to gather remote profile data and analyze it locally. Instead, the tooling can work where the data is, be rich, and support the features that users are accustomed to in traditional web or GUI applications.

Keywords: *Linux, Text-based User Interface, System Design, open source tool, profiling*

Introduction

Early Linux performance profiling relied on users or compilers inserting code to record time spent (Graham et al., 1982). Things changed with the Intel Pentium MMX CPU that exposed hardware performance counters. At first, these counters could be read by a privileged user and were for the entire system. OProfile (Levon 2004) added the ability for sampling of data from the kernel and for performance counters to be associated with a process. This required saving and restoring the counters on context switches. Just as early profiling inserted instrumentation, approaches like KProbes and tracepoints allowed information to be gathered from a running Linux kernel without the need to restart a system or rebuild the kernel. In 2008, these approach-

es were unified in the Linux perf subsystem and with the new perf tool command (Molnar, 2008 & Melo, 2010). The perf tool focused on two problems: gathering data from a running system, including the challenge of knowing what data a user wanted to gather, and showing that gathered data afterwards. The visualizations were limited to being either simple print statements or a user interface on the then somewhat standard slang (Davis, 2022) library. The simplicity of the visualization led to a plethora of tools being created that presented the data in a more appealing graphical way (Google, 2026 & Mozilla, 2026 & Wong, 2026 & Intel, 2026). The technology for profiling keeps advancing with BPF (Gregg, 2019) and virtual PMU-based (Google Cloud, 2026) profiling, but perf remains the standard for gathering counter and sampling data while being the reference implementation for Linux's profiling APIs.

The command line interface (CLI) has undergone a significant transformation, evolving from a primitive, monochrome text entry point into a sophisticated environment capable of hosting high-fidelity Text User Interfaces (TUIs). Modern terminal emulators now support 24-bit TrueColor, extensive Unicode character sets, and standard mouse interaction protocols, thus effectively blurring the line between traditional shells and graphical environments. These advancements are particularly relevant for AI interaction and observability, where the density of information – such as real-time model weights, tokenization streams, or complex decision trees – requires visual hierarchy and intuitive navigation. By leveraging these rich capabilities, developers can maintain the low-latency, high-portability benefits of the terminal while employing the visual cues necessary for deep technical analysis. These features are available locally or over the network with SSH (Ylonen, 1996).

The evolution of User Interface (UI) development has been defined by a transition from imperative, hardware-dependent libraries to declarative, high-level abstractions. Early UI engineering relied on system-specific toolkits like Win32, Cocoa, or GTK, where developers manually managed widget lifecycles and pixel-level updates. However, the ubiquity of the web transformed the Document Object Model (DOM) from a static document hierarchy into a dynamic application runtime, catalyzing a fundamental shift in design philosophy. Modern development is now dominated by component-based architectures – pioneered by libraries such as React, Vue, and Angular – which treat the UI as a reactive function of state rather than a series of manual mutations. By utilizing techniques like the Virtual DOM and reconciled state updates, these libraries have normalized a paradigm where complex, interactive interfaces are built with modularity and portability in mind. This dominance of these DOM-based patterns has set a high standard for responsiveness and developer ergonomics, influencing UI design even in specialized environments like mobile apps and rich command line interfaces. The most prominent library for command-line interface UI generation is Textual for Python (McGugan, W. & Contributors, T., 2021), with imitators appearing in other programming languages. This work uses the Textual UI library.

System Design & Implementation

The first job the visualization must do is aggregate sampling data. In its "recording" mode, the perf tool creates a ring buffer, shared between the tool and the kernel, for sampling data typically on each CPU. The hardware is configured to generate an interrupt when an event occurs, such as a number of instructions being executed or a number of cache misses. The interrupt causes event data to be written into the ring buffer in the kernel's interrupt handler. As interrupt delivery may be slow, each major computer vendor provides extensions that gather side data such as PEBS on Intel (Intel Corporation, 2026), IBS on AMD (Drongowski, 2007), and SPE on ARM (ARM, 2024). The side data can be interpreted in the kernel's interrupt handler to precisely blame which instruction an event happened and write this into the shared ring buffer. On the user side, the perf tool is blocked from polling on the kernel. When a ring buffer fills, the perf tool awakens and writes the sequence of event data into the perf.data file.

The first job for visualization is to take the events in the perf.data file and turn it into aggregated profile data. A sampling event appears with the values being optional and configurable when the event is opened:

Figure 1. *A Linux perf sample event*

Event header	Describes the size of the event and metadata, such as whether the event happened when running user or kernel code
ID	An identifier that allows the sample to be attributed to an event that the perf tool opened. Grouping of events impacts the ID value.
IP	The virtual memory address of the instruction where a sample happened.
PID/TID	The process and thread where the sample happened.
Time	The time of the sample in nanoseconds since the time the Linux kernel started.
Address	For memory events, the address in memory being accessed.
ID	An optional second location for the ID.
Stream ID	A hardware identifier for the event that caused the interrupt.
CPU	The Linux virtual CPU number on which the interrupt occurred. Multi-core and SMT mean the number of CPUs.
Period	The number of events that occurred before the interrupt.
Read counters	Additional counters are configured to be read when the interrupt happens.
Callchain	An array of virtual addresses gathered by a kernel stack walker. Each virtual address is the location within a function that calls the next.
Raw data	An array of raw data is often used to describe tracepoint events.
Branch stack	Generally, an array of “from” and “to” values describes the most recent branches performed before the interrupt.
User registers	The user registers at the point of the interrupt.
User stack	A snapshot of a certain amount of the stack at the point of the interrupt (by default, 8 kilobytes). In combination with the user registers, this allows user stack unwinding rather than the kernel.
…	Additional fields for things like user registers, latency information associated with a sample, etc.

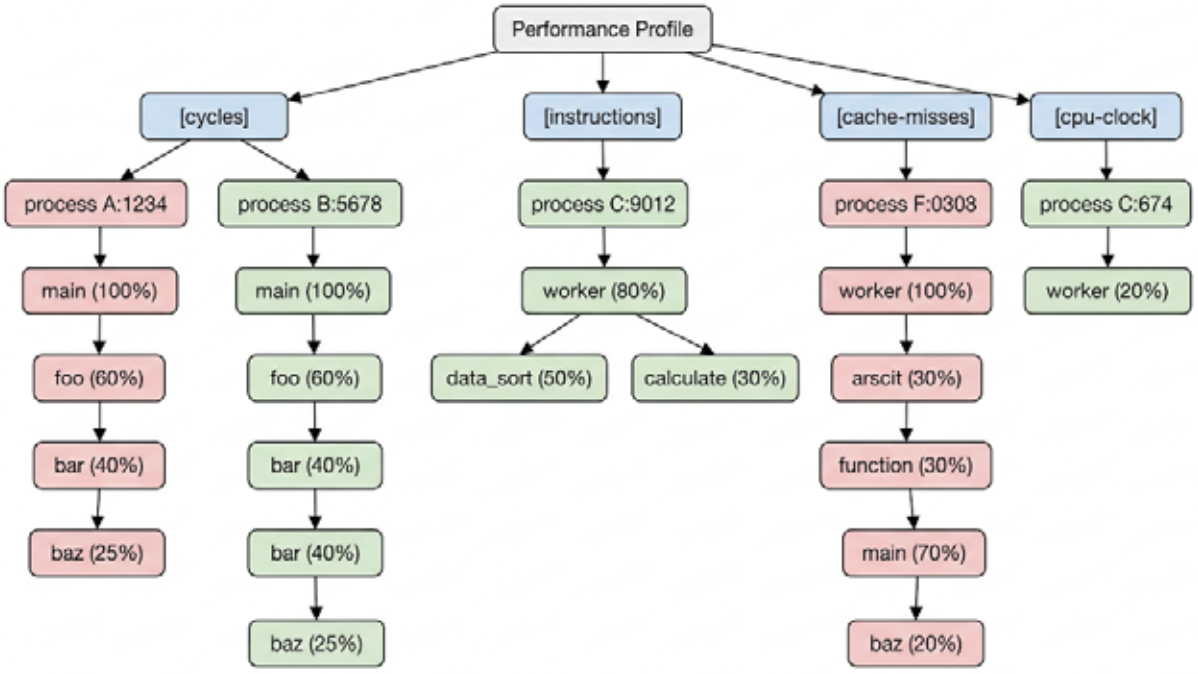
Three places within the event can describe which functions call which, and allow for the period of an event to be aggregated from the leaf function that takes the interrupt, up to the functions that called it. The call chain is the most straightforward to process; it is generally formed by the kernel walking frame pointers within the stack and writing out the return information from the stack, which is adjacent to the frame pointer. The branch stack gathers the most recent branch from and to addresses and can be configured to record function calls and returns. The branch stack provides a window of instruction addresses when an interrupt occurs. This window of samples can be interpreted and stuck together to create a more complete calling information. The user stack relies on a library like libdw (Mc Grath, R. & Drepper, U., 2026) or libunwind (Mosberger et al., 2002) that can interpret debug information from a binary to walk the stack, especially for binaries that have been optimized with the default `-fomit-frame-pointers` flag. In our tool, the stack is walked from any source thanks to a common abstraction within the perf tool.

Having a stack isn’t sufficient to create a profiler; the virtual addresses need to be turned into symbols – the functions that call one another and that the programmer is familiar with. In the perf tool, the challenge of turning a virtual address into a symbol is done by gathering additional “sideband” data of “comm” events that describe the command a PID is running, and mmap events that describe memory layout kernel calls. The perf tool configures the kernel to gather sideband data, but when sampling in already started

processes, it must synthesize the data, creating events for processes that are already running. When processing the perf.data file, each process must have its virtual address space simulated so that an address within it can be turned back into the file and offset within the file it came from. These files and offsets can then be translated into the function symbol and possibly inlined functions if debug information is available.

In the tool, every event forms a root stack node, and the children of this node are the process samples of the event that have occurred within. Generally, after the process, the first node is the main function of the running program, with nodes below this being the functions called within the program. The period data is aggregated from the leaf function that was interrupted up through the callers.

Figure 2. *A depiction of the nodes built when processing the perf.data file*



Now that we have aggregated profile data, we need to show it to the user. Our application has different "tabs" across the top of the screen. The first default tab just presents the profile data above as a tree, as shown in the next section. The second tab is our flame graph visualization that uses a custom Textual Widget.

Figure 3. *The FlameVisitor abstract base class*

```
class FlameVisitor(ABC):
    """Parent for visitor used by ProfileNode.flame_walk"""
    @abstractmethod
    def visit(self, node: Optional["ProfileNode"], width: int) -> None:
        """Visit a profile node width the specified flame graph width.

        Args:
            node: The `ProfileNode` for the current segment. This may be `None`
                to represent a gap or an unknown portion of the stack.
            width: The calculated width of the flame graph rectangle for this
                node, which is proportional to its sample count.
        """
```

A common feature of flame graphs is to support zooming into a function, where the selected function is expanded to the width of the screen. Our widget needs to do two jobs: render

the profile onto the screen and determine from a mouse coordinate which function has been selected for zooming into. In order to do these two tasks, the code uses a visitor pattern with a FlameVisitor abstract base class, a Find Visitor implementation to locate functions from a mouse click, and a Strip Visitor that renders strips of the flame graph within the TUI.

Figure 4. *The flame_walk mouse click and rendering function*

```
def flame_walk(self, wanted_strip: int, cur_strip: int, parent_width: int,
               selected: "ProfileNode", visitor: FlameVisitor) -> None:
    """Recursively walks the tree to visit a single flame graph row.

    This method calculates the proportional width for each child
    based on its value (sample count) relative to its parent. It
    then invokes a `visitor` to process each segment of the flame
    graph row.

    Args:
        wanted_strip (int): The target depth (Y-axis) of the flame graph row
                            to generate.
        cur_strip (int): The current depth of the traversal.
        parent_width (int): The width of the parent of this node.
        selected (ProfileNode): The currently selected node in the UI, used
                                to adjust rendering to highlight the
                                selected path.
        visitor (FlameVisitor): A visitor object whose `visit` method is
                                called for each segment of the flame graph
                                row.
    """
    if parent_width == 0:
        return

    parent_selected = selected == self or self.has_parent(selected)
    child_selected = not parent_selected and self.has_child(selected)
    if not parent_selected and not child_selected:
        # Branches of the tree with no node selected aren't drawn.
        return

    # left_over is used to check for a gap after the children due
    # to samples being in the parent.
    left_over = parent_width
    for child in sorted(self.children.values(), key=lambda node: node.value,
                        reverse=True):
        if parent_selected:
            if self.value:
                desired_width = int((parent_width * child.value) / self.value)
            else:
                desired_width = parent_width // len(self.children)
            if desired_width == 0:
                # Nothing can be drawn for this node or later smaller children.
                break
        elif child == selected or child.has_child(selected):
            desired_width = parent_width
        else:
            # A sibling or its child are selected, but not this branch.
            continue

        # Either visit the wanted_strip or recurse to the next level.
        if wanted_strip == cur_strip:
            visitor.visit(child, desired_width)
        else:
            child.flame_walk(wanted_strip, cur_strip + 1, desired_width,
                             selected, visitor)
        left_over -= desired_width
        if left_over == 0:
            # No space left to draw in.
            break

    # Always visit the left_over regardless of the wanted_strip as there
    # may be additional gap added to a line by a parent.
    if left_over:
        visitor.visit(None, left_over)
```

So that gaps between flame graph entries may be accounted for, or drawn blank, gaps in the flame graph are indicated to the visitor with “None”. The StripVisitor maintains state to alternate the colors of the strips in the flame graph to make adjacent stacks easier to differentiate.

The function to walk the profile nodes and call the visitor functions is flame_walk. The flame walk is given the character width of the screen and recursively calls itself, doing a depth-first traversal of the profile tree. Both the FindVisitor and StripVisitor indicate which line (y-axis) they want to render or find within. The visitor only descends down the profile enough to answer this question; it must, however, recurse on every node above the sought line in order for the width information of the children to be accurate. For lines below the selected node, or if there is no selected node, the width of the line is:

$$parents\_width \times \frac{node\_period}{parents\_period}$$

Before the selected node, the width is just the parents_width if the parent is part of the selected profile tree or zero otherwise. To determine the relationship with the selected node, a has_parent and a has_child function were added to the profile. We consider their performance in the section Performance Considerations.

Results

Figures 5 and 6 show screenshots of the tree report and flame graph views. The workload is a test workload built into the perf tool itself called brstack.

Figure 5. *A tree report showing aggregated profile data for the perf test brstack workload*

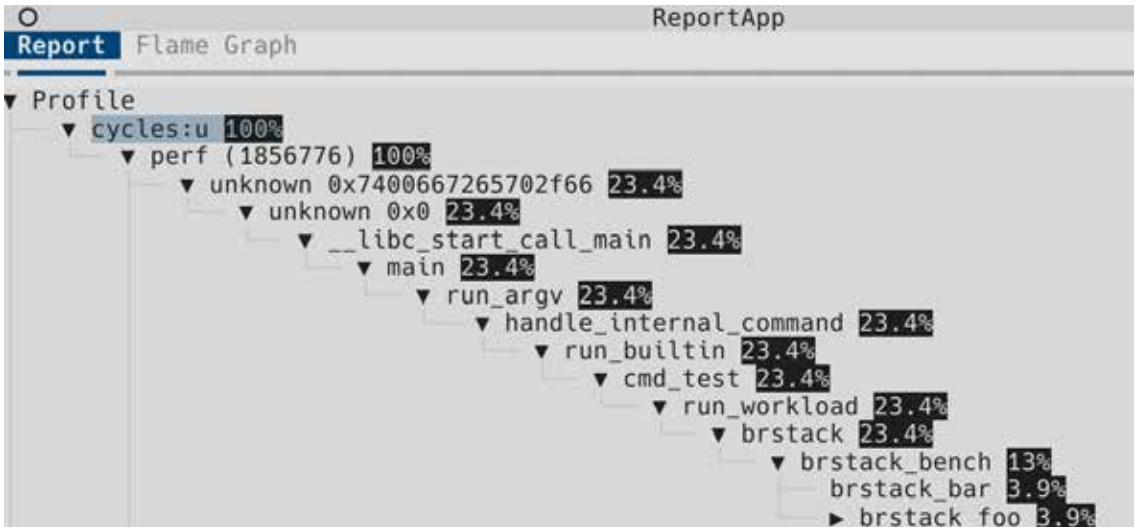
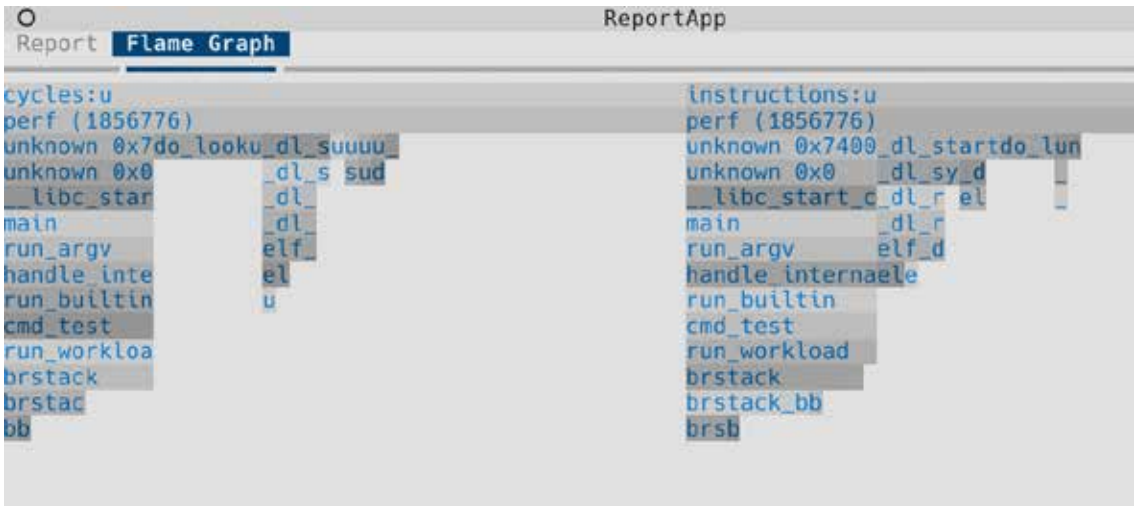


Figure 6. *A flame graph visualization of the perf test brstack workload*



This simple profile data has near-instantaneous rendering; we've also tested with large profiles where the user interface remains fast and performant.

Performance Considerations

It is common in a flame graph implementation to cache information about the flame graph produced by something akin to the flame_walk function. This has been found unnecessary in our implementation due to some key optimizations:

- Profile nodes record their parents so that the has_parent operation can be computed in O(log N), where N is the number of profile nodes and log N approximates the height of the tree. The operation walks up the tree, checking each node if it is the one sought;
- The has_child operation is also O(log N) by observing that asking a profile node whether it has a child of a particular node is equivalent to asking whether the sought node has a parent that is the node being sought;
- When the width of part of the tree has become zero, we don't need to descend or visit any children. This limits the number of possible flame_walk recursive calls to the width of the screen multiplied by the tree's depth.

By avoiding caching data and just needing the profile and width to generate the graph, the widget can use less memory and be responsive in the case of the screen being resized. The widget itself is embedded in a ScrollView to enable an unlimited height for the individual flame graphs.

Future Updates

The initial implementation of the tool was posted to the Linux Kernel Mailing List in July 2025 (Rogers, 2025). A feature missing from this initial version was support for Textual's theming for the color scheme of the widget. The flames alternated between shades of red and white. The color API in Textual allows for two colors to be blended; by using this, it is possible to follow the user's selected theme.

The profile API in perf's Python support creates a dictionary for all data in performance events. This introduces overhead that slows the parsing of the perf.data file. Using perf's integrated Python support is also problematic, as both Textual and perf think they are the main thread of the application. To avoid issues, Textual is started only when perf reaches the end of the perf.data file. Work is underway to make perf a full Python module. This work lowers the overhead of processing events by lazily computing the values needed by the Python application. It also more naturally fits with other libraries like Textual that can now control the main application thread.

A common complaint in using a profiler is the delay in waiting to visualize large data files. As the UI is reactive, it is possible to make the profiler refresh as the file is loaded. Allowing the profile to be navigated and used while the profile data processing is underway eliminates potentially long file loading delays.

Conclusion

This work has introduced tooling that allows command-line profiling both locally and remotely, with the output displayed in the portable terminal, minimizing copying of data while being rich and fully functional. Profiling is a foundation for understanding machine performance and for software development; improving tooling here lifts the entire ecosystem. As a part of the standard Linux perf profiling environment, this tooling won't replace existing UI frameworks, but it will introduce tooling that is sufficient in a great number of use cases. As an open source tool, the widget can be shared widely, and the tool can form a foundation for more profile visualization work.

References

Agarwal, M. & Barroso, J. & Chakraborti, T. & Dow, E. M. & Fadnis, K. & Godoy, B. & Pallan, M. & Talamadupula, K. (2020). IBM Research. Project CLAI: Instrumenting the Command Line as a New Environment for AI Agents.

Arm. (2024). Statistical Profiling Extension for the Armv8-A Architecture. Arm Developer Documentation. URL: https://developer.arm.com/documentation/102557/latest

Davis, J. E. (2022). The S-Lang Programming Library [Computer software]. Jedsoft. URL: https://www.jedsoft.org/slang
Drongowski, P. J. (2007). Instruction-Based Sampling: A New Approach to Performance Analysis for AMD Family 10h Processors. AMD Open Source Lab.
Google (2026). Perfetto: System profiling, app tracing and trace analysis. Android Open Source Project. URL: https://perfetto.dev
Google Cloud. (2026). PMU overview | Compute Engine Documentation. Google Cloud. URL: https://cloud.google.com/compute/docs/pmu-overview
Graham, S. L. & Kessler, P. B. & Mckusick, M. K. (1982). Gprof: A call graph execution profiler. Published in ACM SIGPLAN Notices, – Vol. 17. – Issue 6. – P. 120–126. URL: https://doi.org/10.1145/872726.806987
Gregg, B. (2016). The flame graph. Published in Communications of the ACM. – Vol. 59. – Issue 6. – P. 48–57, URL: https://doi.org/10.1145/2909476
Gregg, B. (2019). Book: BPF Performance Tools: Linux System and Application Observability. Addison-Wesley Professional. ISBN-13: 978-0136554820.
Intel. (2026). Intel® VTune™ Profiler User Guide. URL: https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html
Intel Corporation. (2026). Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3B: System Programming Guide, Part 2. (See Chapter 19: Performance Monitoring).
Levon, J. (2004). OProfile manual. Victoria University of Manchester.
McGrath, R. & Drepper, U. (2026). Elfutils: A collection of utilities and libraries for ELF files and DWARF data. Sourceware. URL: https://sourceware.org/elfutils
McGugan, W. & Contributors, T. (2021–2026) Textual: A Rapid Application Development framework for Python [Computer software]. Textualize.io. URL: https://github.com/Textualize/textual
Melo, A. C. D. (2010). The New Linux 'perf' tools, presentation from Linux Kongress.
Molnar, I. (2008). [Announcement] Performance Counters for Linux. Linux Kernel Mailing List. Available at: URL: https://lwn.net/Articles/310176
Mosberger, D. & Eranian, S. (2002). IA-64 Linux Kernel: Design and Implementation. Prentice Hall. (See Chapter 6: Stack Unwinding).
Mozilla. (2026). Firefox Profiler: Web app for Firefox performance analysis. URL: https://profiler.firefox.com
Rogers, A., Rogers, I. (2026). Perf script: New treport script. URL: https://lore.kernel.org/linux-perf-users/20250725082425.20999–1-irogers@google.com
Wong, J. (2026). Speedscope: An interactive flamegraph visualizer. URL: https://www.speedscope.app
Ylonen, T. (1996). SSH – Secure Login Connections over the Internet. Proceedings of the 6th USENIX Security Symposium, – P. 37–42.

submitted 13.04.2026;
accepted for publication 27.04.2026;
published 30.04.2026

Contact: alice.mei.rogers@gmail.com