Visualinux: Visual Debugging for the Linux Kernel

HANZHI LIU, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
YANYAN JIANG, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China
CHANG XU, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, China

Debugging the Linux kernel is challenging because script-based debuggers present overwhelming, unstructured textual information. To address this, we present Visualinux, a system designed to make the kernel state visually tractable. Visualinux introduces two domain-specific languages: ViewCL, for constructing reduced object graphs at controllable abstraction levels, and ViewQL, for customizing these graphs to meet specific debugging objectives. Furthermore, Visualinux can visualize differences between kernel state snapshots, helping developers track changes in data structures over time. Our evaluation shows that Visualinux can effectively visualize complex kernel components and assist in diagnosing sophisticated kernel bugs.

CCS Concepts: • **Software and its engineering → Software testing and debugging**; **Operating systems**; • **Human-centered computing → Visualization toolkits**.

Additional Key Words and Phrases: Linux Kernel, Software Visualization, Debugging

1 Introduction

The inherent complexity of the Linux kernel makes it notoriously difficult to debug: its runtime state contains millions of live objects, and complex objects can have hundreds of fields interconnected through multiple pointers and containers. Developers often struggle to sift through this vast network of objects across different execution points to extract meaningful insights for pinpointing the root cause of a bug.

The heart of debugging is *simplifying* the overwhelming information produced during kernel execution. Existing debugging tools such as GDB [11, 14, 27] and kernel loggers/tracers [19, 22] rely primarily on textual interfaces, which naturally fall short when inspecting high-dimensional information such as interconnected complex data structures. This forces developers to *implement customized simplification scripts* for each debugging task, imposing substantial cognitive overhead and nontrivial programming effort.

To mitigate this debugging challenge, our previous work [80] presented Visualinux, the first debugging framework that makes the state of the Linux kernel visually understandable with low human effort. It introduces a two-layer abstraction for programmatically creating simplified views of kernel states:

This paper, originally titled "Understanding the Linux Kernel, Visually" [80], was invited by ACM TOCS for extended publication on the recommendation of the EuroSys 2025 PC chairs. It extends the prior conference version by introducing a new state diff mechanism, refining the prototype tool, and presenting additional case studies.
This work was supported by the National Natural Science Foundation of China (#62522209, #62272218, #92582201), Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), and Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China.
Authors' Contact Information: Hanzhi Liu, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; e-mail: hanzhiliu@smail.nju.edu.cn; Yanyan Jiang, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; e-mail: jyy@nju.edu.cn; Chang Xu, State Key Laboratory of Novel Software Technology, Nanjing University, Nanjing, Jiangsu, China; e-mail: changxu@nju.edu.cn.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2026 Copyright held by the owner/author(s).
ACM 1557-7333/2026/6-ART
https://doi.org/10.1145/3820892

Just Accepted

(1) View *Construction* Language (ViewCL) for object graph construction, which enables the creation of object graphs at customizable levels of abstraction. By providing a programmatic interface, ViewCL makes it practical to visually understand the extremely large and complex kernel state through simplified state views defined by developers.
(2) View *Query* Language (ViewQL) for state view customization, which enables fine-grained customization of a simplified object graph in a developer-friendly manner. ViewQL provides the ability to exclude specific irrelevant object types or fields, ultimately producing a human-readable plot. The simplicity of ViewQL further enables plot adjustments through natural-language debugging objectives.

This paper extends Visualinux with a *state diff* mechanism to aid in understanding how kernel states progress throughout execution. We propose the concept of a *differential object graph*, which highlights changes between two state views captured at different execution points. Our object graph comparison algorithm enables the automatic generation of state diffs from simplified object graphs produced by ViewCL and ViewQL. We also design a state diff visualization to offer developers a clear, flattened view of state changes.

Overall, our work presents a novel attempt to bridge the gap between the large, complex program state of the Linux kernel and the limited attention span of the human brain. By simplifying the kernel state for specific debugging objectives, selectively visualizing the state for flexible interactive debugging, and constructing state diffs across execution points, we propose the first approach that enables practical visual debugging for the Linux kernel.

We implement these ideas in Visualinux, which functions as a detached front-end for GDB [14]. It provides developers with a user-friendly, push-button interface to manage multiple views. This paper improves the system architecture of Visualinux by consolidating both DSL evaluations into the GDB backend and upgrading the object graph rendering techniques of the visualizer front-end, thereby enhancing customizability and extensibility. Visualinux is publicly available at:

https://icsnju.github.io/visualinux

Visualinux can reconstruct representative textbook-style figures from real runtime states of Linux 6.1, including figures from the well-known (though now obsolete) textbook *Understanding the Linux Kernel* [45]. We further demonstrate its effectiveness through case studies on complex kernel data structures [5] and real-world Linux kernel CVEs [88–90]. Our evaluation results show that Visualinux generally incurs acceptable overhead for interactive debugging scenarios.

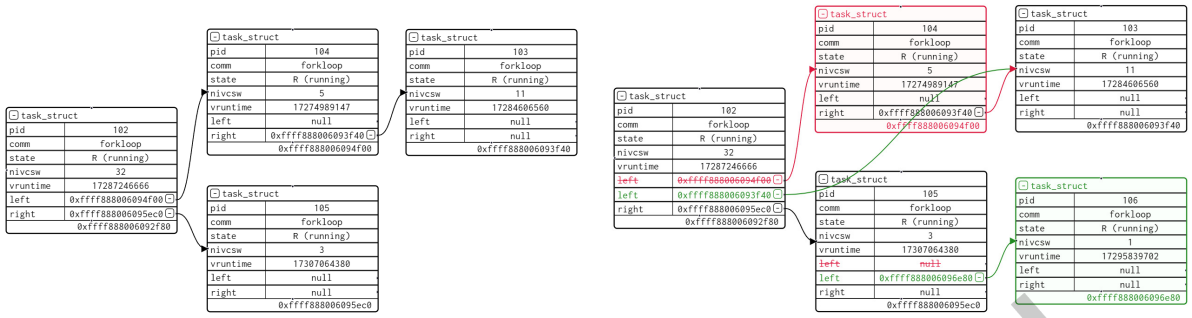
The rest of this paper is organized as follows. Sections 3.1 and 3.2 present the design of the Visualinux DSLs for state simplification and visualization. Section 3.3 introduces the mechanisms for creating and visualizing state diffs. Implementation details and evaluation are discussed in Sections 4 and 5. Section 6 discusses the methodologies and limitations of Visualinux as well as future work. Sections 7 and 8 summarize related work and conclude the paper.

2 Visual Debugging for the Linux Kernel: Overview

When confronted with nontrivial debugging tasks, developers often begin with an observed anomaly and work backward to determine how it arose. Along the way, they form hypotheses grounded in domain knowledge, test these hypotheses by inspecting relevant state and execution, and gradually narrow the search space until the root cause is identified and understood [34, 72, 73, 76].

Debuggers are central to this backward reasoning process. GDB [11, 14, 27] is the most widely used kernel debugger and supports fine-grained runtime control, allowing developers to set breakpoints at arbitrary execution points and inspect any value reachable in memory. In-kernel loggers and tracers [19, 22], along with eBPF-based techniques [29], enhance system observability by intercepting selected execution events with customizable hooks.

Just Accepted



(a) Runtime state of the run queue before the `__schedule()` invocation.

(b) State diff highlighting changes after the `__schedule()` invocation.

Fig. 1. Visualization of the CFS run queue on the first processor. The simplified object graphs are generated by evaluating ViewCL and ViewQL programs at GDB breakpoints, and the plots are rendered in Visualinux's browser-based front-end.

Modern debuggers such as drgn [2] enable dynamic introspection of live kernel state through Python scripts with built-in helpers for common kernel data structures.

One crucial step in debugging is *understanding* what actually happened, and we argue that visual debugging can substantially improve the debugging experience. Suppose the developer is debugging the Linux kernel's Completely Fair Scheduler (CFS) [4]. CFS maintains a per-CPU run queue implemented as a red-black tree that orders runnable threads by virtual runtime (vruntime) to determine scheduling. To determine *which threads are scheduled to run on the first processor*, a developer would typically run a recursive traversal script at a breakpoint:

```
# GDB script
define print_cfs_threads
    set $node = $arg0
    if $node != 0
        set $se = container_of($node, struct sched_entity, run_node)
        set $task = container_of($se, struct task_struct, se)
        print_thread $task
        print_cfs_threads $node->rb_left
        print_cfs_threads $node->rb_right
    end
end
define print_thread
    set $task = $arg0
    set $se = &$task->se
    printf "PID: %d, name: %s, vruntime: %llu\n", \
        $task->pid, $task->comm, $se->vruntime
end
set $root = cpu_rq(0)->cfs.tasks_timeline.rb_root.rb_node
print_cfs_threads $root
```

Ironically, debugging the kernel often entails debugging the debug script itself. The script includes nontrivial logic and relies on internal kernel macros and long reference chains to retrieve required runtime objects. Worse still, the developer must interpret a lengthy pre-order traversal, mentally reconstructing the tree to extract insights. Visualinux simplifies this procedure with a ViewCL abstraction for traversing and reshaping commonly used data structures:

```
define Task as Box<task_struct> [              // Declare a Box for a task_struct object
    Text pid, comm
    Text ppid: parent.pid
    Text<string> state: ${task_state(@this)}   // @this refers to the Box itself
                                               // <...> denotes the display format
```

Just Accepted

```
6       Text se.vruntime
7   ]
8
9   root = ${cpu_rq(0)->cfs.tasks_timeline}        // cpu_rq(0) is the run queue of processor #0
10
11  sched_tree = RBTree(@root).forEach |node| {    // Create a predefined container with given root
12      yield Task<task_struct.se.run_node>(@node) // For each node yield (create) a box of Task
13  }                                              // whose associated object is @node.se.run_node
14
15  plot @sched_tree                               // Plot the object graph rooted at @sched_tree
```

In ViewCL, a `Box` declares the fields of interest (Lines 1–7), and the `${...}` syntax embeds C expressions, allowing field declarations (Line 4) and retrieval of kernel state information (Line 9). ViewCL also provides standard-library container structures such as `RBTree` (Line 11), which exposes a `forEach` iterator that accepts a closure. In this example, the closure instantiates a `Task` box (Line 12) for each node encountered during traversal. When this program is evaluated at a GDB breakpoint, `root` is first resolved as the starting point for `RBTree` to traverse the run queue. The iterator then invokes the closure on each visited node, which yields a corresponding `Task` box for inclusion in the simplified object graph. Finally, the `plot` command (Line 15) sends this object graph to the front-end visualizer for rendering (Figure 1a).

For systems under heavy workloads, the plot might still be too large to read. To simplify it further, one can provide a debugging objective as a natural-language refinement request in GDB for the current plot. Visualinux then synthesizes a ViewQL program, applies it to the already-extracted object graph, and refreshes the front-end view accordingly. For example, given the message "Focus on kernel threads", the following ViewQL program is synthesized and applied to the ViewCL-extracted object graph:

```
1  task_all = SELECT task_struct FROM *           // Select all tasks
2
3  task_2 = SELECT task_struct                    // Select tasks that have pid or ppid of 2,
4      FROM task_all                              // i.e., 'kthreadd' and its direct children,
5      WHERE pid == 2 OR ppid == 2                // which are non-user threads in Linux
6
7  UPDATE task_all \ task_2 WITH collapsed: true  // Mark user threads as collapsed
```

ViewQL follows an SQL-like syntax, which `SELECT`s a set of objects and `UPDATE`s their key-value attributes. The example above focuses on kernel threads in the run queue by marking all user threads (i.e., excluding `kthreadd` and its direct children) as "collapsed", compacting them into small, clickable summary nodes that can be expanded on demand.

To understand how the kernel maintains the run queue, one can evaluate the ViewCL and ViewQL programs at two execution points (e.g., before and after an invocation of `__schedule()`, the core function responsible for process scheduling) and request a state diff between them. Visualinux then produces a differential object graph that highlights changes in a format akin to UNIX diff output [57]. As shown in Figure 1b, the scheduler selects the leftmost thread from the red-black tree to run next and re-inserts the previously running thread. The colored objects and pointers make these structural changes to the run queue immediately visible, whereas traditional scripts leave developers to infer them from textual output.

3 View Construction, Query, and Diff in Visualinux

3.1 Simplifying the Kernel State

Constructing a Reduced Object Graph. Daniel Jackson's *small scope hypothesis* [67] posits that most bugs can be exposed by testing a program with inputs within a limited scope. This principle suggests that many errors in software can be revealed by examining a small subset of the program state, providing the theoretical grounding for state simplification.

Just Accepted

The kernel state is essentially a graph, where kernel objects are nodes connected by pointers. Given a debugging objective, simplifying the kernel state is equivalent to extracting a representative part from the full graph. To make this procedure efficient, we introduce ViewCL, a domain-specific language to formally define *which parts of the state are relevant to developers' debugging objective* and construct the simplified object graph. The (simplified) core syntax of ViewCL is listed below:

$$
\begin{array}{llll}
\text{Program} & P & ::= & b^* \\
\text{BoxDecl} & b & ::= & \texttt{Box}\ box\ \{\ v^+\ \} \\
\text{View} & v & ::= & view\ \{\ i^*\ \} \\
\text{Item} & i & ::= & \texttt{Text}(expr) \\
 & & | & box(expr) \\
 & & | & \texttt{Link}(box(expr))
\end{array}
$$

The fundamental building block of ViewCL is `Box`, corresponding to a C `struct` object, which is plotted as a box. One can also construct "virtual" boxes that do not correspond to real objects. Each `Box` encloses its `Views`, with each `View` corresponding to a customized layout for plotting an object. The concept of `View` is inspired by database system views [105]. A `View` consists of items: a `Text`, a `Link`, or another nested `Box`. A `Text` displays a string, and a `Link` denotes a logical relation (e.g., reachability in the object graph) between two boxes. Each `Link` is a member of a `Box` and points to another `Box`. A pointer can naturally correspond to a `Link`.

`Views` allow one to plot an object (box) with different levels of detail and focus. For example, the following ViewCL program offers three views of a `task_struct`:

```
define Task as Box<task_struct> { // Define a Box for task_struct with multiple views
    :default [                    // Default view: only pid and comm (command)
        Text pid, comm
    ]
    :default => :sched [          // Scheduler view: extends default with vruntime
        Text se.vruntime
    ]
    :sched => :sched_rq [         // Scheduler-rq view: displays the task's run queue
        Link runqueue -> @rq      // Declare a link named "runqueue" to @rq defined below
    ]
} where {
    rq = ...                      // Define a box for displaying the run queue
}
```

The `=>` operator is used for view inheritance, allowing a view to include all items from another view. In this example, `:sched_rq` inherits from `:sched`, which includes `pid`, `comm`, and `se.vruntime`.

Given a `Box` type and an expression indicating the root object, a ViewCL program describes the rules and process for constructing views tailored to the developer's debugging objectives. We also provide built-in support for generic data structures (linked lists, red-black trees, etc.) and basic ViewCL programs for frequently used objects (`task_struct`, run queue, files, etc.), so that customized simplification can usually be achieved by modifying only a few lines of ViewCL code.

Plotting the Object Graph. When a ViewCL program is evaluated within a paused debugging context (i.e., at a GDB breakpoint), Visualinux traverses kernel objects in the captured runtime state and constructs the simplified object graph specified by the program. To render this graph in a readable form, Visualinux uses a layered layout algorithm that arranges objects into a tree-like hierarchy whenever possible, so that related objects appear on nearby layers and major link directions remain easy to follow visually. The algorithm first assigns each box to a rank, which roughly corresponds to its display layer in the hierarchy, based on the graph's link structure. It then reorders boxes within each rank and assigns coordinates to reduce edge crossings, overlaps,

Just Accepted

and unnecessary whitespace, making the resulting plot easier to read at a glance. All state plots presented in this paper are generated through this process.

As with any automatic graph layout, readability can still degrade when too many objects or cross-links are shown at once. Visualinux addresses this issue primarily at the graph-construction level rather than relying on the layout algorithm alone. Before rendering, ViewCL and ViewQL reduce the graph so that developers typically inspect only a task-relevant substate of limited scale and complexity. For plots that are still crowded, the front-end further supports interactive refinement operations such as collapsing, trimming, and focusing on selected objects, which are discussed in Sections 3.2 and 4.5. In our evaluation, the simplified plots were readable and none of the plots were excessively crowded.

3.2 Customizing the Visualization

Selective State Visualization. Evaluating a ViewCL program on a kernel state yields a simplified kernel object graph $G(V, E)$, where vertices are objects (Boxes) and edges are pointers (Links). To reduce the complexity of G to a more visually manageable scale, we introduce an additional layer of abstraction for specifying the display modality of each object. This is equivalent to updating a (virtual) database table where object identity serves as the primary key and the other columns encode the object's display attributes. For this purpose, we introduce ViewQL, an SQL-like domain-specific language, limited to the following syntax (nested queries are disallowed):

```
Statement  s  ::= v = SELECT expr FROM v [WHERE c]
              |  UPDATE v WITH attr: value
Condition  c  ::= c AND e
              |  c OR e
              |  e
CondExpr   e  ::= member op value
```

Specifically, given a state graph $G(V, E)$:

- SELECT identifies a subset of vertices (boxes) in V that satisfy the filtering condition in WHERE.
- UPDATE assigns display attributes to the set of selected boxes.

ViewQL attributes of each box control either the information shown in the box or the way the box is displayed in the plot. For example, the *view* attribute selects one of the predefined ViewCL views of the box, while *trimmed* and *collapsed* determine the visibility of the box.

Suppose that a ViewCL program implements three views for `task_struct`: `default` for basic thread information, `show_mm` for thread memory management, and `full` for a more comprehensive display. One can configure the process tree to display memory mappings for all user threads (tasks associated with address space) by changing their displayed view to `show_mm`:

```
user_threads = SELECT task_struct            // Select tasks with a non-null "mm" field
    FROM *
    WHERE mm != NULL

UPDATE user_threads WITH view: show_mm       // Let these tasks display the "show_mm" view
```

One can further focus on writable memory areas by collapsing the non-writable memory regions:

```
non_writable_vmas = SELECT vm_area_struct    // Select memory areas without a writable flag
    FROM *
    WHERE is_writable != true                // is_writable is a ViewCL-defined field

UPDATE non_writable_vmas WITH collapsed: true // Set these VMAs' "collapsed" attribute
```

Just Accepted

Those collapsed VMAs are rendered as compact, clickable summary nodes in the front-end, so the developer can still see that those VMAs exist and selectively expand them later if needed. In short, *view* determines which information a box presents after expansion, whereas *collapsed* determines whether that information is currently exposed in the plot.

The separation of ViewCL and ViewQL, rather than using a single domain-specific language, explicitly divides the tasks of extracting the kernel state and refining it into a visually tractable plot. This collaboration maximizes efficiency and flexibility: since commonly used kernel components and data structures are predefined in ViewCL, most developers can work exclusively with ViewQL, often without being aware of ViewCL's existence. In short, ViewQL offers flexible on-demand customization for developers' own debugging objectives.

Natural-Language-Driven Visual Debugging. The simplicity of ViewQL (only `SELECT` and `UPDATE` without nested queries) enables large language models to synthesize ViewQL programs from natural-language descriptions, allowing developers to directly "talk" to the debugger and obtain a specific plot.

Consider the previous example of displaying the `mm` struct for user threads, which can be described as "display the `task_structs` that have non-null `mm` members with the `show_mm` view." This natural-language description, combined with our predefined ViewQL examples for in-context learning, yields the following correctly synthesized ViewQL program for view customization:

```
threads_mm_not_null = SELECT task_struct
    FROM *
    WHERE mm != NULL
UPDATE threads_mm_not_null WITH view: show_mm
```

3.3 Highlighting the State Changes

Comparing the Object Graphs. Developers understand a program's state transitions by examining the *differences* between snapshots captured at different execution points. A useful diff visualization for kernel understanding should capture three types of state *changes*: (1) primitive value changes, (2) structural changes such as object relocation, and (3) object lifetime changes such as allocation and deallocation.

A straightforward approach would be to traverse the two object graphs in parallel and compare them structurally as trees. However, this falls short of accurately capturing structural or lifetime changes: if the same runtime object is detached from one data structure and later linked into another, a tree-aligned comparison sees it disappear at one position and reappear at another, easily misclassifying a relocation as a deletion followed by a creation.

Visualinux's diff construction follows an object-centric approach: rather than comparing whole graphs, it treats each object as the unit of comparison, builds per-object diffs, and composes them to form the whole state diff. Each object is identified by the pair of its type and address, referred to as the object's key. With the start and end states modeled as two maps from keys to objects, each object is classified as follows:

- *unmodified* or *updated* if its key exists in both states and the object contents are the same or different, respectively.
- *introduced* or *retired* if its key exists only in the ending or starting state, respectively.

For updated objects, fields are similarly categorized as unmodified, updated, introduced, or retired based on their values and presence across the two states. Each updated pointer field emits two explicit links, one to the field's old target and one to its new target; these links are treated the same as normal links.

Combining these per-object diffs highlights changes in each object's connectivity and naturally reveals structural changes, avoiding the complexity of processing and aligning two entire pointer networks. For instance, a new link to an unmodified or updated object indicates an object relocation.

Lifetime changes are subtler due to address reuse. After an object is freed, the kernel may later allocate another object of the same type at the same address, which is a common effect of caching mechanisms in kernel memory

Just Accepted

management [33, 44]. Without additional tracking, this reuse would be misclassified as unmodified or updated in the state diff since the object key remains unchanged.

To disambiguate address reuse from continuity, Visualinux traces the allocation history of kernel objects to distinguish reallocated objects. Only objects that appear in the extracted object graphs are traced, limiting the runtime overhead. Visualinux also creates a pair of introduced and retired boxes with the same type-address key to make the lifetime boundary explicit.

Visualizing a Differential Object Graph. We visualize state diffs in a scheme inspired by UNIX diff [57]: introduced and retired objects are rendered in green and red, respectively, while updated objects expose each modified field as an old/new value pair, analogous to Git's line-based diffs.

When a retired object's address is reused by a later allocation of the same type, the differential object graph must preserve both the pre-state and post-state versions of the same type-address key to expose the lifetime boundary. To keep both boxes visible and queryable, Visualinux annotates them with synthetic suffixes such as `task_struct$new` and `task_struct$old` so that they can be treated temporarily as distinct types.

Since a state diff is still represented as an object graph, developers can customize a differential graph with ViewQL in the same way as a regular object graph. For convenience, a `SELECT` statement over a base type also matches the corresponding diff variants of that type. For example, one can collapse all threads except the introduced ones to focus on process creation:

```
all_tasks = SELECT task_struct FROM *             // Select all (including differential) tasks
new_tasks = SELECT task_struct$new FROM *         // Select all introduced tasks
UPDATE all_tasks \ new_tasks WITH collapsed: true // Make the other tasks collapsed
```

As with normal object graphs, ViewQL programs for differential graphs can also be synthesized from natural-language descriptions.

4 Visualinux Implementation

4.1 Overview

We implement the Visualinux prototype for interactive kernel debugging. Visualinux can debug the Linux kernel via GDB, either in a virtual machine or on a remote physical machine running KGDB. It comprises two components[1]:

- The GDB extension, which is integrated into the GDB host to handle developers' requests, evaluate ViewCL and ViewQL programs, and manage the generated object graphs. It is implemented in ~5,500 lines of Python, along with ~500 lines of GDB scripts that expose kernel functions invisible to GDB in ViewCL, such as static inline functions.
- The visualizer front-end, which is a detached browser-based application that waits for requests from the GDB host, responds to them, and performs interactive state visualization. It is implemented in ~4,000 lines of TypeScript, managing the panes, plotting the simplified object graphs, and providing the interactive visual interface.

Visualinux implements four custom GDB commands, referred to as *v-commands*:

- *vplot* extracts object graphs from the current kernel state according to a given ViewCL program. It can also synthesize basic ViewCL code for trivial debugging objectives.
- *vctrl* controls the panes and the views displayed in them. For instance, it can split an existing pane to create a new one, or apply a ViewQL request to an object graph.

[1]In the conference version of this paper, ViewCL evaluation took place in the GDB extension, while ViewQL evaluation took place in the visualizer front-end. We have redesigned the architecture of Visualinux so that both DSLs are evaluated in the GDB extension. This design centralizes object graph management and makes the visualizer front-end more concise, extensible, and scalable.

Just Accepted

- *vdiff* creates a differential object graph from two ViewCL-extracted object graphs.
- *vchat* relies on large language models to convert the given message into another command (either *vplot*, *vctrl*, or *vdiff*) with well-formed arguments.

Based on large language models (specifically, DeepSeek-V2 [52] in this paper), Visualinux can synthesize ViewQL programs from natural-language descriptions. Specifically, a text message `desc` in *vchat* will be pasted into the following prompt, which includes descriptions, guidance, and several concrete examples:

```
A kernel object graph is ... The vertices are Boxes (objects) and the edges are Links (pointers).
- Each box has a type and members, and may have the following attributes:
  - ... (view, trimmed, ...)
- Each member ... (text, link, ...)
- ...

A domain-specific language ViewQL, whose syntax is similar to SQL database query languages, can
be applied to the kernel object graph. ViewQL has only two types of statements:
- SELECT ...
- UPDATE ...
The syntax of ViewQL is like ...

Here are some examples:
Example 1: select all cfs_rq boxes and change their views to sched_tree.
  - ViewQL code: ...
Example 2: ...

I intend to {{desc}}. Please synthesize a ViewQL program for me.
```

4.2 ViewCL and Interpreter

Language features. While ViewCL provides a programmatic approach to kernel state simplification, challenges still remain when dealing with the Linux kernel. The complexity stems from Linux-specific mechanisms for heterogeneous resource management, including `container_of`-based data structures and CPU-local pointer translation. Furthermore, the Linux kernel employs various C programming techniques to emulate object-oriented and functional programming paradigms, such as multipurpose `void` pointers, complex object unions, and function pointer sets serving as operation interfaces. To improve the usability of ViewCL, we support the following features in addition to its core syntax.

(1) *Data compaction.* The Linux kernel widely adopts data compaction to improve data locality. For example, several short integers may be packed together into a single integer, and the low-order bits of an aligned pointer can be repurposed to store bit flags. ViewCL supports seamless integration of C expressions with ViewCL variables to unpack such compacted data, as well as perform other complex field computations with ease.

(2) *Embedded containers.* In the Linux kernel, container structures (such as linked lists and red-black trees) are typically nested within objects using the `container_of` macro. Figure 2 illustrates a workqueue that maintains asynchronous tasks scheduled in the background. For such embedded containers, the types of contained objects must be inferred from how the container is accessed. As demonstrated in Section 2, ViewCL provides predefined container abstractions (e.g., `RBTree`) to simplify describing these widely used `container_of`-based data structures, improving the practicality of Visualinux.

(3) *Polymorphic objects.* ViewCL supports switch-case statements to model object connections whose existence or target type can only be resolved at runtime. For example, a `file` object may represent different kinds of resources such as pipes and sockets, and the appropriate target object depends on its associated operations table. Using a runtime expression such as `${file->f_op}` as the switch condition, ViewCL can construct links to the appropriate target object according to the file object's current state. Combined with `Container`

Just Accepted

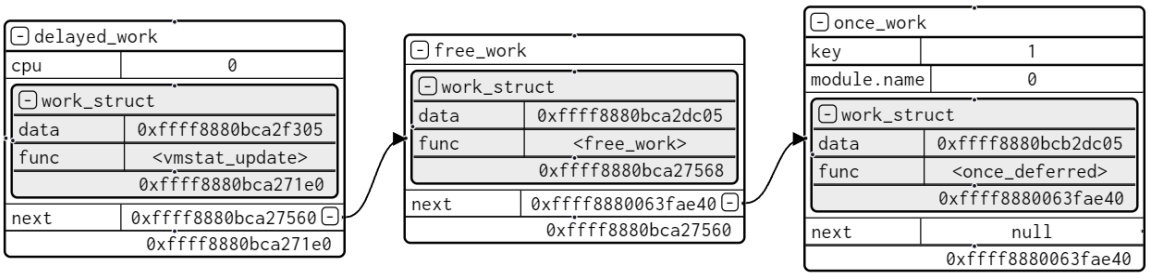


Fig. 2. Visualization of a work list of the `mm_percpu_wq` workqueue in the Linux kernel. Work lists are heterogeneous lists constructed through the `container_of` mechanism, and the types of their nodes are determined by a function pointer field. The displayed next pointer represents the list abstraction extracted by Visualinux rather than the concrete pointer field used in the kernel implementation.

abstractions, the switch-case mechanism also enables elegant handling of heterogeneous structures such as the workqueue list in Figure 2 and the RCU waiting list in Section 5.4.

(4) *Specialized object construction.* Some kernel objects are difficult to locate directly. For instance, obtaining the physical pages from a virtual memory area requires a page table walk, which is architecture-specific and involves multiple branches to handle special pages. On the other hand, inline functions and macros are often optimized out by the compiler and thus invisible to GDB. Kernel developers often develop specialized scripts for these cases [2]. ViewCL supports invoking arbitrary GDB scripts or GDB Python functions with C expressions or ViewCL variables as arguments to enable arbitrary construction of both objects and containers.

(5) *"Virtual" objects.* Logically related kernel objects can be physically disconnected in memory. One typical example is a compact system-status summary, whose key numbers are maintained in separate global variables rather than a dedicated kernel object. To organize such information together, ViewCL allows developers to declare a "virtual" object that has no corresponding runtime object and whose fields are rooted at pure C expressions. For example, one can declare a virtual object including fields like the load averages `${avenrun}` and the current tick counter `${jiffies_64}`.

(6) *Text formatting.* Different fields are easier to inspect in different textual forms. ViewCL provides decorators that control how text fields are rendered, including the size and base of integers, the symbolic name of a function pointer, and the macro names corresponding to bit flags. It also supports emoji decorators for compactly encoding simple state indicators such as boolean values. For example, a spinlock can be rendered as a locked/unlocked icon rather than a raw integer flag, allowing developers to spot contention-related state at a glance without manually decoding bit-level representations. Table 1 lists the text decorators supported by Visualinux.

Implementation of the interpreter. ViewCL programs are interpreted by Visualinux. Starting from a root object specified by the `plot` statement, Visualinux recursively traverses the runtime state to construct the simplified object graph. For each reached object, it creates a box, interprets the expressions of its fields, and resolves the addresses of linked and nested objects to determine which objects should be expanded next. When multiple references point to the same runtime object, the resulting graph preserves that sharing through links rather than duplicating the object along every access path. The traversal stops when no additional objects need to be expanded under the current ViewCL definition.

Just Accepted

Table 1. Text decorators supported by Visualinux

Name	Format	Description	Name	Format	Description
Int	`<type>:<base>`	an integer (e.g. `u64:x`)	RawPtr	`raw_ptr`	raw value of a pointer
Bool	`bool`	true/false	FunPtr	`fptr`	name of a function pointer
Char	`char`	an ASCII character	Flag	`flag:<id>`	macro names of a bit flag
String	`string`	evaluated as `char*`	Emoji	`emoji:<id>`	display of a stateful value
Enum	`enum:<type>`	name of an enum value			

As discussed above, ViewCL supports mixed evaluation of C expressions, GDB scripts, and Python extensions with ViewCL variables. For each expression `${...}`, Visualinux first processes any embedded ViewCL variables (denoted by `@...`) by substituting them with their corresponding C expression values. The special variable `@this` is replaced with the address of the object owning the current scope. Subsequently, Visualinux resolves functions in a specific precedence order: it first checks for GDB Python functions, then GDB macros, and finally C functions, using the first matched identifier. The overall evaluation follows standard C-language precedence and associativity.

4.3 ViewQL and Plot Manager

Language features. In ViewQL, each object (box) can be associated with the following attributes (modifiable by `UPDATE`) for display control:

- *view* (`string`): determines which view of a box is displayed. If absent, `default` is used.
- *trimmed* (`bool`): if set, removes the object and its descendants from the graph.
- *collapsed* (`bool`): if set, displays the object as a small button; clicking the button removes the "collapsed" attribute.
- *direction* (`string`): for containers, indicates whether they grow horizontally (default) or vertically on the screen.

ViewQL also provides a built-in function `REACHABLE(V)` to select all reachable objects from an object set `V`, which can be further filtered to display a set of objects of interest. Additionally, ViewQL supports set operations such as intersection, union, and difference, as well as object-set operators such as `is_inside`.

Implementation of the plot manager. In the GDB extension, each extracted object graph is stored in the `vplot` history list together with a developer-specified identifier, an extraction timestamp, and an in-memory database that maintains object display attributes. ViewQL programs are translated into database queries and evaluated to retrieve and update the corresponding table entries. The updated attributes are then sent to the visualizer to refresh the plot instantly.

4.4 State Diff Generator

Comparing the object graphs. We implement our object-centric state comparison algorithm in the visualizer front-end. Given two state views and a *vdiff* request, Visualinux first scans the object lists of the two compared object graphs and identifies the update type of each object (unmodified, updated, introduced, or retired), including both boxes and containers. Next, it consolidates all objects and pointers, generates diff fields for updated objects, and connects the special diff links to construct and render the final differential graph. At the same time, Visualinux attaches the recorded reallocation history for the two compared states to the *vdiff* request. Using this additional

Just Accepted

lifetime information, the front-end constructs a pair of introduced and retired objects for each reallocated object and assigns them special keys to avoid conflicts.

Tracing kernel object reallocations. Visualinux uses eBPF [29] to trace reallocated objects by instrumenting the return points of kernel object allocators (e.g., `__kmalloc` and `kmem_cache_alloc`) and recording allocation history in preallocated eBPF data structures. The eBPF program is loaded into the kernel during user-space initialization and remains active throughout the debugging session. When kernel execution pauses at breakpoints, the GDB extension inspects kernel internals to fetch records from eBPF memory and sends them to the front-end together with *vdiff* requests.

To prevent data explosion, Visualinux records allocation events only for objects that may later appear in a differential graph, that is, the objects present in any previously generated object graph. The eBPF program maintains a dedicated hash table for this filtering purpose. Whenever a plot is generated through *vplot*, the GDB extension records the addresses of objects in the plot and writes them into the hash table in kernel memory. During subsequent execution, the eBPF probes only record allocations for addresses that match the hash table, which is typically far smaller than the full stream of kernel allocations.

4.5 The Visual Interactive Debugger

The visualizer front-end is implemented as a detached browser-based application. It refreshes the panes and their visualized contents upon receiving HTTP POST requests from *v*-commands executed in GDB: *vplot* with extracted object graphs, *vctrl* with ViewQL programs or other pane operations, or *vdiff* with a pair of IDs indicating the two object graphs to be compared.

The paneling mechanism. Understanding a complex program state often requires juxtaposing related information without losing context [47, 72]. For example, developers may need to compare the process parent tree and the per-CPU scheduling run queue side by side, then focus on one process while keeping the surrounding structures available for reference. To support this style of inspection, Visualinux implements a *paneling mechanism* for creating linked or detached views over two types of panes (each pane displays an object graph)[2]:

- *Primary pane* displays an entire object graph received from *vplot* or *vdiff*, which preserves a full object-graph context for comparison.
- *Secondary pane* displays a focused object picked by the developer from another primary or secondary pane, isolating a local focus without discarding the original context.

The visualizer front-end initially contains a single primary pane, from which developers can grow a hierarchical pane tree using the following operations for comparison, focused inspection, and refinement of state views:

- *Split* (vertically or horizontally) an existing primary pane to create another primary pane.
- *Select* (by clicking objects in the front-end or using ViewQL within *vctrl*) a set of objects from a pane to create a new secondary pane for focused inspection.
- *Refine* an object graph displayed in a pane by applying a ViewQL program or switching the displayed object graph to adapt the current view as the debugging task evolves.

Interactive visualization support. For nontrivial debugging tasks, even a simplified view can be overwhelming for developers to understand at a glance. Therefore, an effective visual debugging interface requires lightweight interactions for navigating the displayed object graph, locating objects and relations of interest, and refining the current view as the debugging focus evolves.

[2]This design is inspired by tmux [25], a widely used terminal multiplexer that provides pane-based window management.

Just Accepted

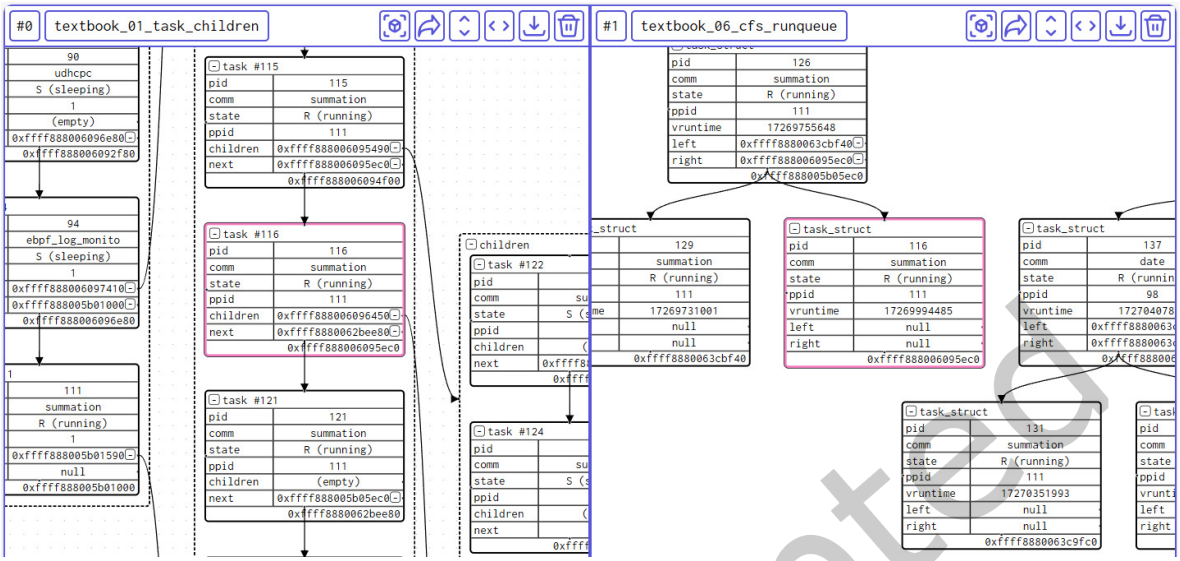


Fig. 3. The front-end of the Visualinux debugger with two primary panes displaying different state views. Through the "focus" operation, the developer can easily locate the same `task_struct` across two process management structures in the visualization.

In response, Visualinux does not merely render the object graph as a static figure. Instead, it supports interactive adjustment of both the displayed views and the panes that contain them, without forcing developers to rebuild the visualization from scratch. These interactions include:

- *Object-level controls*, such as changing the display attributes of an object and highlighting the links associated with a specific object.
- *Pane-level controls*, such as zooming the view in or out, or navigating through the object graph using the mouse.

Developers can also simultaneously explore multiple plots displayed in different panes. For instance, the focus operation highlights the selected object across all plots for cross-context inspection. This capability is particularly useful when a developer identifies an object in one data structure and wants to understand how it is co-managed by other data structures. Figure 3 shows an example where the developer searches for a process in a parent tree (left pane) and a scheduling tree (right pane).

5 Evaluation

In this section, we evaluate Visualinux to answer the following questions:

(1) Can Visualinux construct readable, task-relevant state views across diverse kernel data structures and mechanisms?
(2) Can Visualinux assist in forming and testing debugging hypotheses for real-world kernel bugs that require an understanding of the kernel state?
(3) What performance overhead does Visualinux incur when constructing and simplifying kernel state views for interactive use?

Just Accepted

Table 2. Representative figures in the book *Understanding the Linux Kernel* ported to Linux kernel 6.1.

#	Diagram description	LOC	Δ‡	#	Diagram description	LOC	Δ‡
1	Fig 3-4. process tree	27	○	12	Fig 14-3. block device	75	◓
2	Fig 3-6. PID hash tables	48	◓	13	Fig 15-1. file page cache	70	●
3	Fig 4-5. IRQ descriptors	59	●	14	Fig 16-2. file memory mapping	53	◓
4	Fig 6-1. dynamic timers	46	◔	15	Fig 17-1. page reverse mapping	154	◓
5	Fig 7-1. CFS scheduler runqueue	35	●	16	Fig 17-6. swap area management	19	○
6	Fig 8-2. buddy system and pages	64	◓	17	Fig 19-1†. IPC semaphore	126	●
7	Fig 8-4. slab allocator	102	◔	18	Fig 19-2†. IPC msgqueue	–	●
8	Fig 9-2. process address space	145	●	19	workqueue example	89	●
9	Fig 11-1. signal handler	71	○	20	from process to VFS	96	○
10	Fig 12-3. process fd array	55	◔	21	socket connection	92	◓
11	Fig 13-3. device driver	55	◓				

† ULK plots these diagrams as separate figures for better readability. We can generate both separate and merged ones.
‡ The data structure changes over the evolution from Linux 2.6.11 to 6.1:

- ○ Negligible changes.
- ◔ Some variables or fields have changed.
- ◓ Some fields, data structures or object relations have changed.
- ● The underlying data structure underwent significant changes, e.g., upgraded to a more efficient implementation.

5.1 Reviving Textbook Figures with Contemporary Kernel State

The classic textbook *Understanding the Linux Kernel* [45], often referred to as ULK, has educated a generation of Linux kernel programmers. It provides a comprehensive introduction to the Linux kernel based on version 2.6.11 (in the 3rd Edition), covering nearly all core topics such as process management, memory management, concurrency, and more. ULK also includes illustrative figures that effectively help readers understand these kernel components.

Unfortunately, ULK has become outdated: the Linux kernel is rapidly evolving with emerging hardware and software technologies, data structures have changed over time, and much of the example code no longer compiles. No textbook can keep pace with such rapid development of the Linux kernel [40, 81, 101].

Visualinux offers a unique opportunity to extract high-quality illustrative figures from real kernel runtime states. For each chapter of ULK, we evaluate Visualinux by visualizing one or two of the most representative diagrams of that kernel mechanism, with the following exceptions:

(1) Figures in Chapters 1, 2, 10, 18, and 20 do not involve in-memory runtime state and are thus beyond our scope. For instance, Visualinux cannot generate high-level overview diagrams of operating system architecture.
(2) Chapter 5 (synchronization) focuses heavily on the evolution of lock states over time, which is also outside our scope. However, we can visualize the lock state with a single line of ViewCL code, as described in Section 4.2.
(3) ULK lacks a chapter on the network stack. To address this, we added a figure of a socket connection as if we were introducing the Linux kernel's network subsystem to readers.

Table 2 summarizes the results: Visualinux can indeed "revive" ULK, demonstrating its ability to handle various kernel data structures and components. For fair per-figure evaluation, each plot is generated independently, and code shared by multiple plots is re-executed rather than reused across figures. We also observe substantial divergence between the textbook figures and modern Linux implementations: 17 out of 21 (81%) figures based

Just Accepted

Table 3. Debugging objectives for ViewQL usability evaluation. Descriptions are simplified.

Diagram	Debugging objective (simplified)
Fig 3-4.	Display view "show_children" of all tasks and trim tasks that have no address space
Fig 3-6.	Trim all PID hash table entries except for a set of specific PIDs.
Fig 4-5.	Trim IRQ descriptors whose action is not configured.
Fig 7-1.	Display view "sched" of all processes, and display the red-black tree top-down.
Fig 9-2.	Display view "show_mt" of mm_struct, collapse the slot pointer list, and trim all writable vm_area_structs.
Fig 11-1.	Trim all non-configured sigactions.
Fig 14-3.	Display the superblock list vertically, and collapse superblocks that are not connected to any block device.
Fig 15-1.	Trim the extremely large page list in file mappings.
Fig 16-2.	Trim all files that have no memory mapping.
N/A	New figure (socket connection): Trim sockets whose write/receive buffers are both empty.

on Linux 2.6.11 no longer reflect Linux 6.1; among these outdated figures, 14 (82%) involve kernel mechanisms whose implementations have undergone major changes.

These discrepancies between textbooks and the actual kernel implementation hinder new generations of developers from understanding the Linux kernel. With Visualinux, developers can obtain a visual (and even interactive) representation of the structure of modern Linux with low programming effort, making it easier to intuitively understand the kernel state. All generated plots are available on our website.

Visualinux also enables textbook writers to plot illustrative figures from real kernel runtime states. For example, ULK does not cover the network subsystem, which is an important component of the Linux kernel. We added visualizations of the kernel work queue, file systems, and live socket connections in Table 2 (#19–21). For the socket connection case, a few lines of ViewCL code reveal the connection between the read/write buffer queues via the process file descriptor table, while flattening several irrelevant intermediate objects.

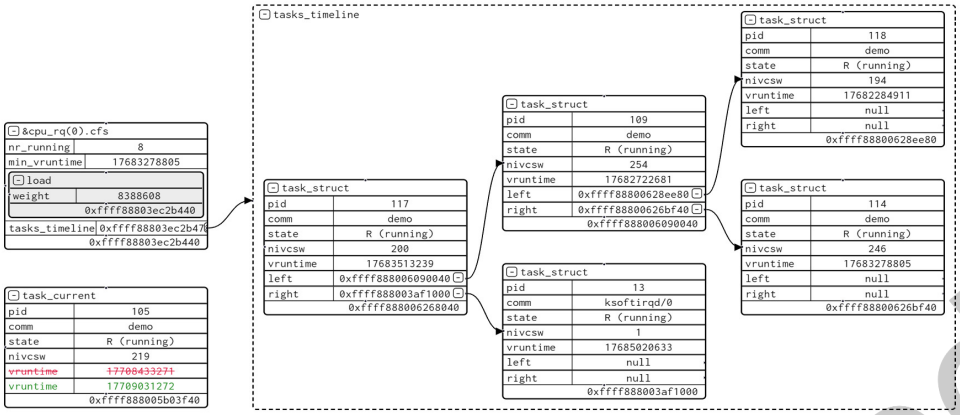
5.2 Going Beyond the Textbook through Interactive Visualization

The interactive nature of ViewQL provides a flexible way to support varying debugging objectives, helping developers inspect the kernel state on demand. We constructed hypothetical debugging objectives for 10 selected diagrams in ULK as listed in Table 3. All of these debugging objectives involve fewer than 10 lines of ViewQL code, and DeepSeek-V2 [52], the model used in our evaluation, successfully generates all of them from natural-language descriptions. This result suggests that Visualinux does not rely on frontier model capabilities: since ViewQL inherits SQL's language features, an earlier general-purpose large language model is sufficient to synthesize ViewQL programs from a few in-context examples.

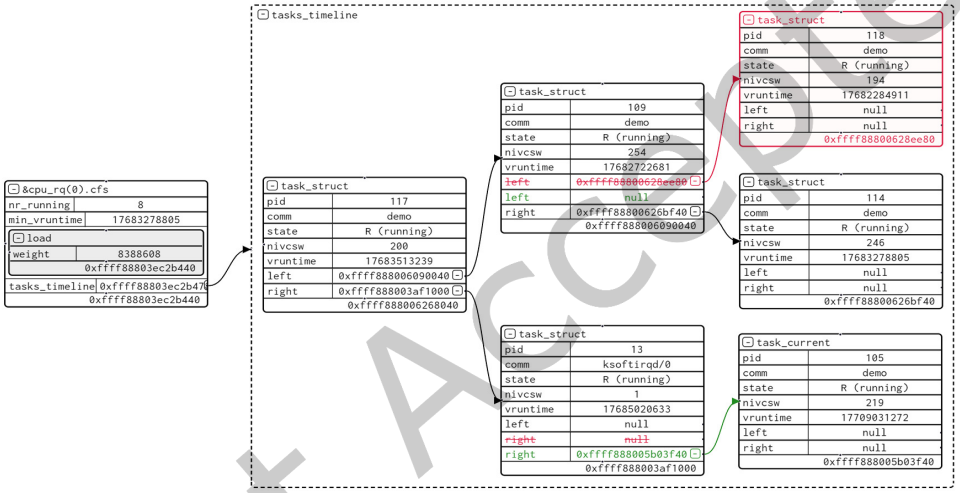
For Fig 14-3 of ULK, the following LLM-generated ViewQL program customizes the visualization to display all superblocks vertically. It further collapses the superblocks that are not attached to any block device, assuming that we intend to inspect disk-based file systems:

```
a = SELECT List
    FROM *
UPDATE a WITH direction: vertical
b = SELECT super_block
    FROM *
    WHERE s_bdev == NULL
UPDATE b WITH collapsed: true
```

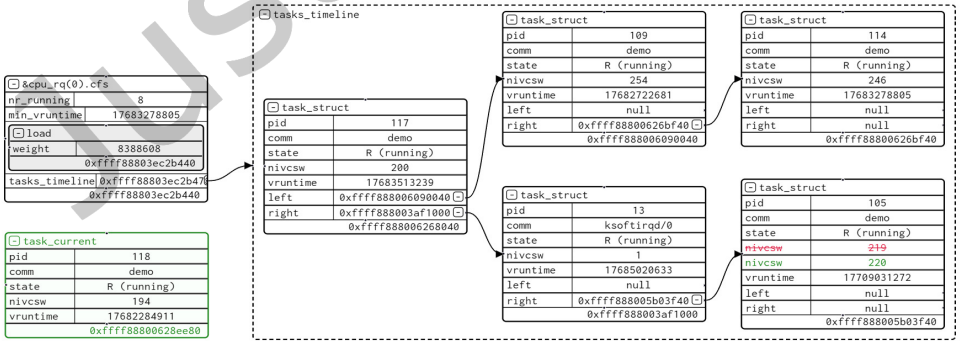
Just Accepted



(a) Updating the vruntime of the current thread.



(b) Maintaining the run queue data structure.



(c) Changing the current thread pointer to complete the context switch.

Fig. 4. Step-by-step state diff visualization of the first processor's CFS run queue and current thread during process scheduling. Each subfigure captures several key changes in an execution of `__schedule()`.

Just Accepted

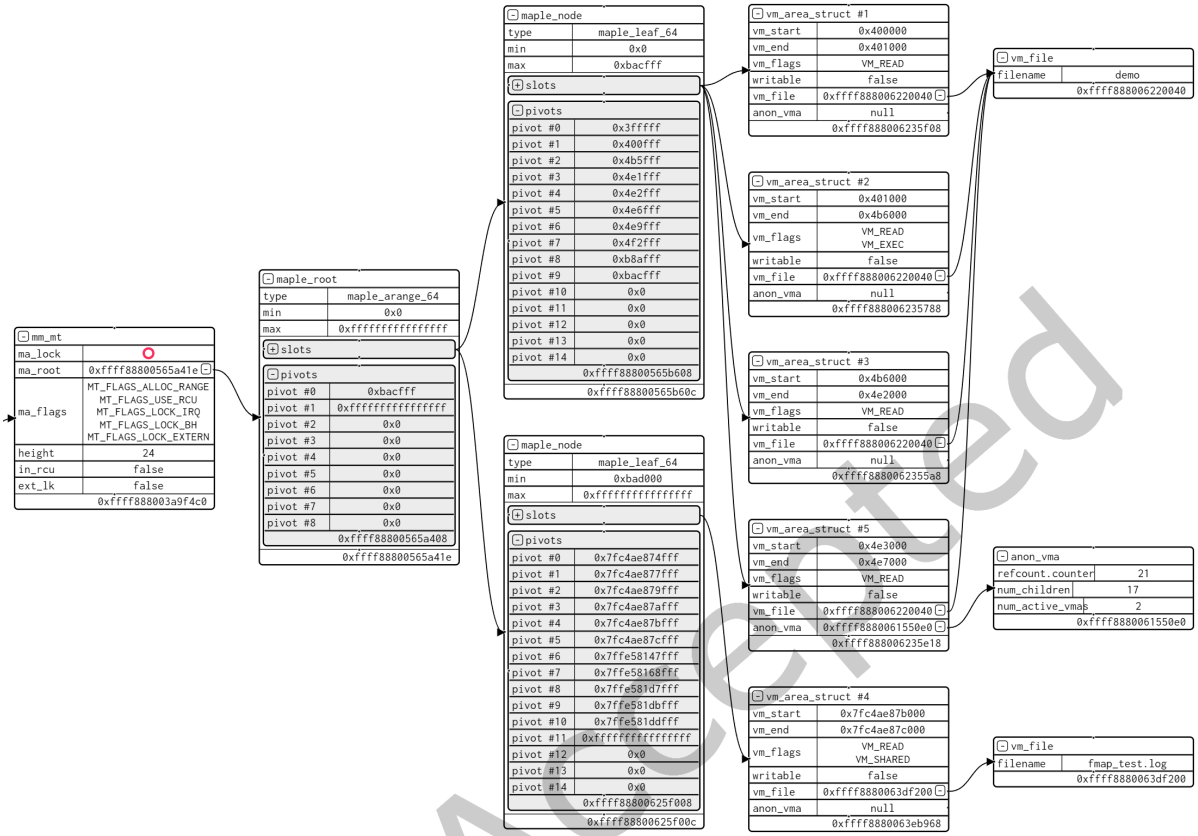


Fig. 5. Visualizing a maple tree representation of the read-only memory areas of a process.

Just Accepted

Furthermore, state diff visualization provides developers with an interactive approach to understanding the logic and semantics of the kernel implementation, serving as an effective complement to static source code and documentation. As an illustrative example, to understand how the kernel selects the next thread to schedule, we step through the execution of `__schedule()` (the core function of process scheduling) from its entry point to the completion of the context switch, requesting a state diff at each step relative to the previous state. Figure 4 shows the critical steps captured in this process. We first write a ViewCL program (similar to the one in Section 2) to visualize the CFS run queue and the current running thread of the first processor. Starting from the entry point of `__schedule()`, the vruntime (which determines the scheduling order) of the current thread is first updated (**??**). The current thread is then inserted into the run queue, and the leftmost thread is selected to run next (**??**). After updating the context switch counter, the current thread pointer is finally changed to point to the scheduled thread (**??**), completing the CFS scheduling procedure.

5.3 Live Visualization of an Under-documented Data Structure

Linux 6.1 introduced the maple tree [5], a scalable data structure for maintaining a set of ordered intervals, replacing the two-decade-old red-black tree-based implementation of memory-mapped regions. The maple tree

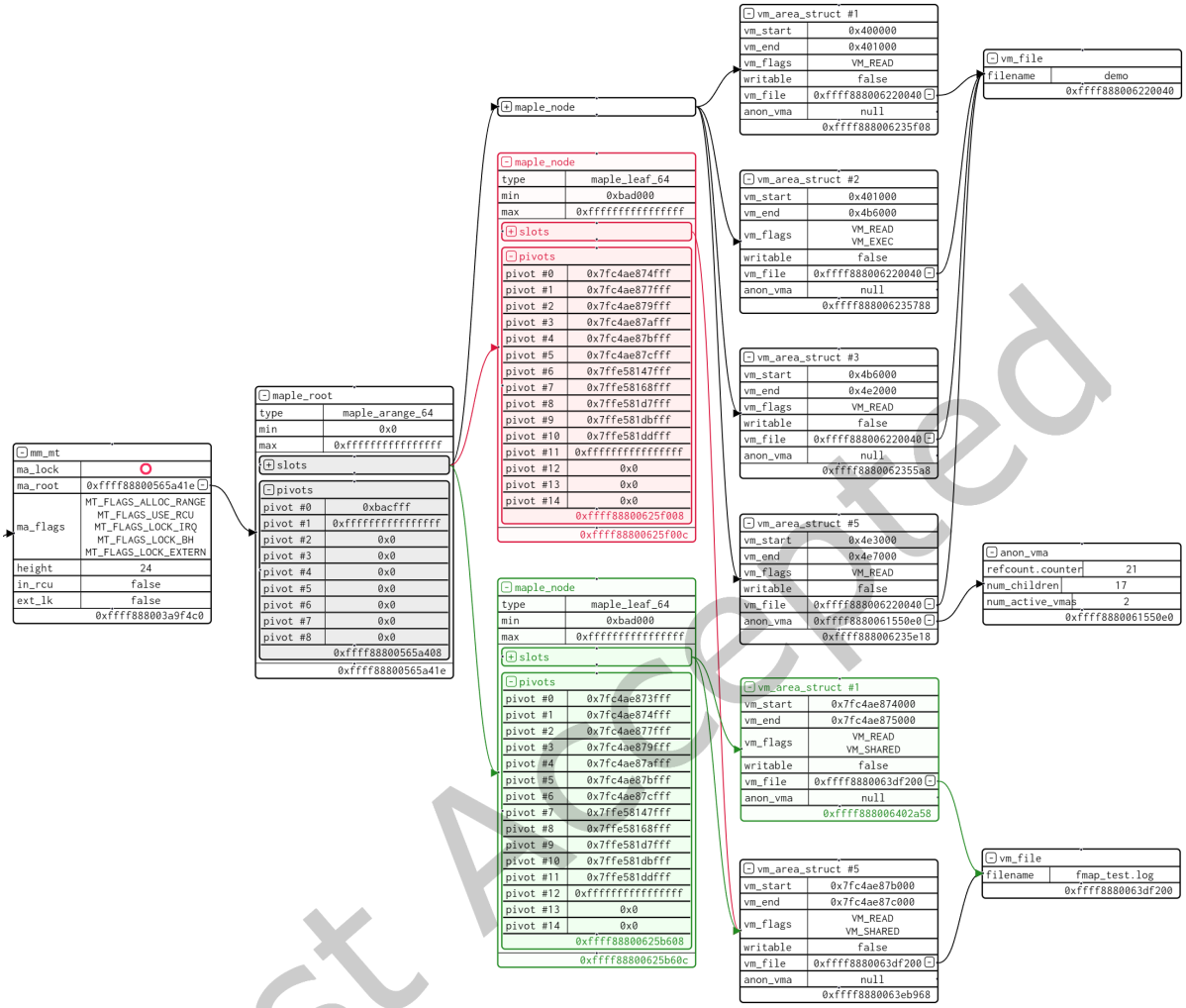


Fig. 6. Visualizing the state diff across a `vma_mas_store` invocation during a user-space memory mapping.

Just Accepted

is a range-based B-tree for tracking gaps and storing ranges using read-copy-update (RCU) [23]. It is one of the most complex generic data structures in the Linux kernel.

Unfortunately, like many newly developed components, the maple tree still lacks detailed documentation and tutorials. Developers may find it challenging to understand its design and implementation details. Existing documentation [5, 21, 66] covers introductory concepts, design principles, and APIs. However, it provides limited detail on the data structure internals, including how it maintains node relations, compacts data for efficiency, and uses RCU for lock-free updates.

At the time of writing, we are not aware of any existing diagrammatic representation of this modern kernel data structure. With Visualinux, we plot the maple tree in approximately 70 lines of ViewCL code and around 100 lines of GDB Python extensions for node type checking, tree traversal, and object graph construction. About

half of the ViewCL code consists of straightforward declarations of object fields, and most of the GDB Python code implements helper functions for parsing the data structure.

One can use ViewQL to further simplify and customize the plot. The following ViewQL program collapses the large but irrelevant lists of slot pointers and trims all writable memory areas to focus on the read-only ones (assuming this is the debugging objective), yielding the plot in Figure 5, which provides a readable representation of how the address space of a process is managed.

```
// Collapse the slots field of all maple_node objects
slots = SELECT maple_node.slots
    FROM *
UPDATE slots WITH collapsed: true

// Make all writable memory areas invisible
writable_vmas = SELECT vm_area_struct
    FROM *
    WHERE writable == true
UPDATE writable_vmas WITH trimmed: true
```

State diff visualization can effectively assist developers in understanding the maintenance of complex data structures. For example, to observe how a new node is inserted into the maple tree during an `mmap` system call, one can evaluate the ViewCL program at both the entry and exit points of `vma_mas_store` and compare the two generated plots. Figure 6 shows the resulting differential graph under a specific insertion scenario. The new memory region begins exactly at the boundary of an existing maple node's managed address range, and that node still has free slots. Under this condition, the kernel reallocates the node, updates metadata such as pivots, and accommodates the new mapping. The following ViewQL program collapses unrelated maple nodes to help developers further narrow their focus:

```
// Collapse all nodes except the introduced and retired ones
all_nodes  = SELECT maple_node FROM *
diff_nodes = SELECT maple_node$old, maple_node$new FROM *
UPDATE all_nodes \ diff_nodes WITH collapsed: true
```

5.4 Diagnosing Kernel Bugs through Complex State Understanding

In this subsection, we examine whether Visualinux can help developers test debugging hypotheses for real-world kernel bugs by exposing task-relevant state evidence through simplified views and diffs. We revisit two high-severity Linux CVEs as reproducible but initially unexplained vulnerabilities. For each case, we reconstruct a hypothesis-driven debugging trajectory from the observed symptom to the root cause, focusing on steps where the decisive evidence lies in complex kernel state spanning multiple subsystems or execution points.

Case 1: Maple tree and RCU waiting list. CVE-2023-3269 [89], named StackRot, is a use-after-free bug in Linux memory management that can be exploited for arbitrary kernel code execution. When a thread touches an address below its `MAP_GROWSDOWN` stack, the kernel expands the stack but mistakenly retires a maple-tree node while holding only the `mm` read lock. Since the node is freed only after RCU-delayed reclamation, the decisive evidence of memory corruption is split across three execution contexts: the reader-side traversal that fetches the node, the update path that retires it, and the deferred RCU callback that later frees it. Although the failure is reproducible and follows a classic use-after-free pattern, Linux kernel developers still took nearly two weeks to converge on a robust fix [8]. Figure 7 summarizes this cross-CPU execution flow.

Initial symptom. The bug reproducer (a typical stress-testing workload) runs two concurrent threads: the reader repeatedly reads `/proc/self/maps`, while the writer repeatedly touches addresses just below the current stack mapping to trigger stack expansion. The kernel address sanitizer [16] reproducibly reports a use-after-free on a maple tree node. The fault is localized to the reader-side maple tree traversal path, where the node is dereferenced shortly after being fetched from the tree.

Just Accepted

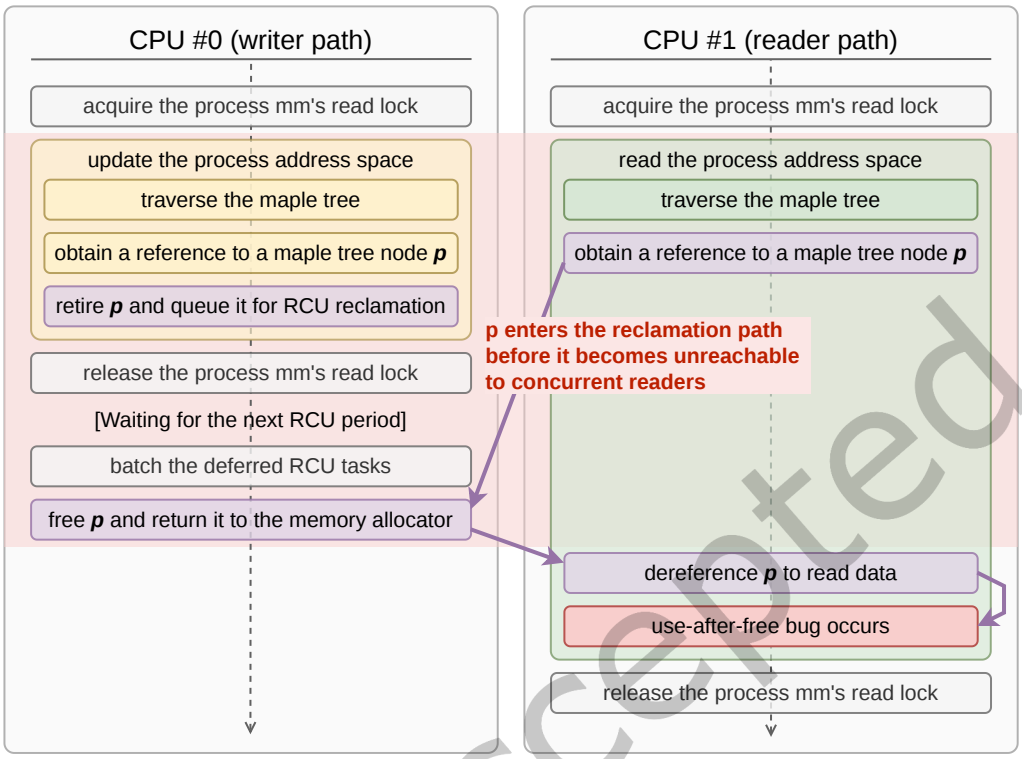


Fig. 7. A simplified execution trace of the StackRot use-after-free bug. The writer path on CPU #0 retires a maple tree node *p*, queues it for deferred RCU reclamation, and later frees it after the grace period. During the same interval, the reader path on CPU #1 traverses the tree, obtains a reference to *p*, and later dereferences it. The bug is that the writer path retires *p* while holding only the read lock; as a result, *p* can enter reclamation before it becomes unreachable to concurrent readers, and the later dereference triggers a use-after-free.

Just Accepted

Hypothesis H1: The anomaly is caused by corruption inside the maple tree. A natural first hypothesis is that the reader fetches an invalid node because the structure of the maple tree is corrupted, so the developer first determines what role the fetched node plays in the current address space. To understand the structural context, the developer visualizes the fetched node in the maple tree (similar to Section 5.3) instead of interpreting it from raw debugger output.

Conclusion: H1 weakened. Across failing runs, the node consistently appears as a leaf node rather than an internal one, and no obvious evidence of tree corruption is found. This makes a structural error in the internal tree organization less likely and narrows the search space to leaf-level address space entries. However, this still does not explain why an apparently valid leaf node later becomes invalid. The next possibility is therefore not that the tree structure is broken, but that this otherwise valid leaf node is later corrupted after being fetched from the tree.

Hypothesis H2: The fetched leaf node is corrupted during a tree update. The developer next asks whether and where the leaf node itself is corrupted. To test this possibility, the developer places breakpoints around suspicious code paths, compares the state views extracted immediately before and after the update path with *vdiff*, and checks whether the fetched node changes unexpectedly. To focus on the narrowed scope, the developer can modify the

ViewCL program to remove all non-leaf nodes by creating a new container through ViewCL's converter function (Line 15):

```
  define MMStruct as Box<mm_struct> {
      :default [
          Text<u64:x> mmap_base
          Text mm_count: mm_count.counter
          Text map_count
      ]
      :default => :show_vmas [
          Link mt -> @mm_mt
      ]
+     :default => :show_addrspace [
+         Link addrspace -> @mm_as
+     ]
  } where {
      mm_mt = MapleTree(@this.mm_mt)
+     mm_as = Array.convFrom(@mm_mt, VMArea)
  }
```

A `MapleTree` maintains an ordered set. `Array.convFrom` traverses this set and produces a sorted list of its objects. We assign this list to `mm_as` (Line 15), which is displayed in the "address space" view (Lines 10–12), to provide a `pmap`-like representation of memory-mapped regions.

Multiple leaf nodes may change concurrently, cluttering the visualization. To keep attention on the fetched node, the developer can "pin" it with a natural-language instruction:

Hide all `vm_area_struct` *objects except for the one with address* `<node_address>`.

Large language models can provide the following ViewQL program to prune unrelated memory areas from the plot:

```
a = SELECT vm_area_struct
    FROM * AS vma
    WHERE vma != <fetched_node_address>
UPDATE a WITH trimmed: true
```

Conclusion: H2 rejected. The focused inspection does not reveal evidence that the fetched node's metadata is corrupted in place. Instead, when the two states bracket the buggy update path, the state diff shows that the suspicious node disappears from the maple tree before the reader-side failure manifests. This unexpected transition suggests that the problem lies not in corruption of the leaf node itself, but in how the node is removed from the tree, which raises the next question: where does the retired node go?

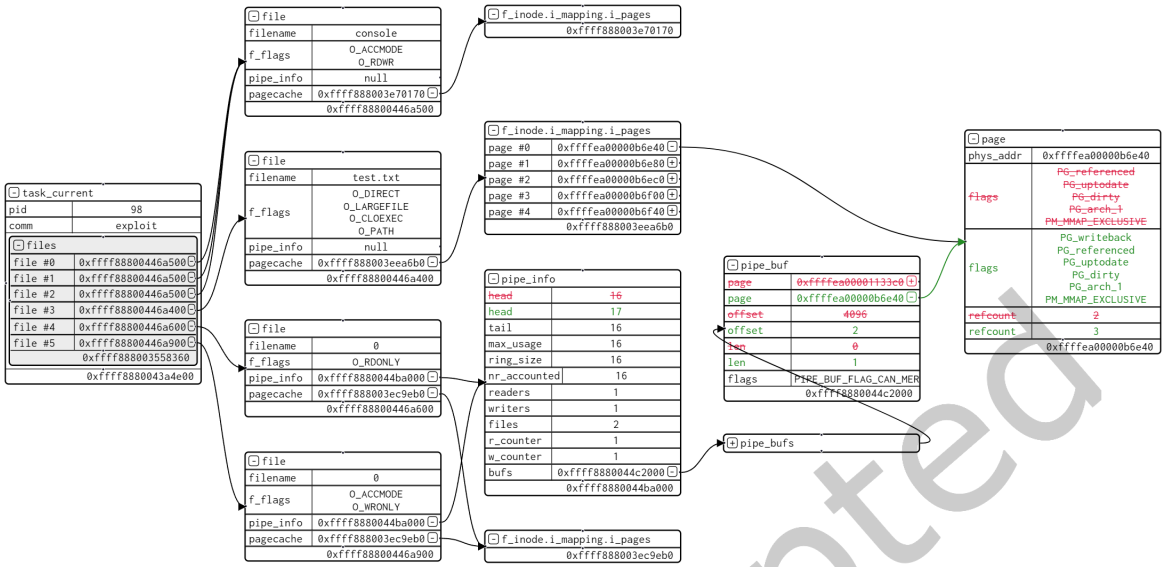
Hypothesis H3: The retired node enters an RCU reclamation path. A brief inspection of the corresponding code suggests that this path is closely related to RCU-based reclamation. To understand the relevant state more comprehensively, the developer updates the ViewCL program to visualize the RCU waiting list together with the maple tree. Based on the existing visualization codebase, the developer adds a basic view of the RCU waiting list with only ~30 additional lines of ViewCL code[3]. The developer then repeats the procedure and compares the state views extracted immediately before and after the suspicious update path identified in H2.

Conclusion: H3 supported. This time the fetched node does not simply disappear in the state diff; instead, it moves onto the RCU waiting list with a free callback attached. This confirms that the node is prematurely reclaimed by RCU and becomes invalid before another CPU dereferences it.

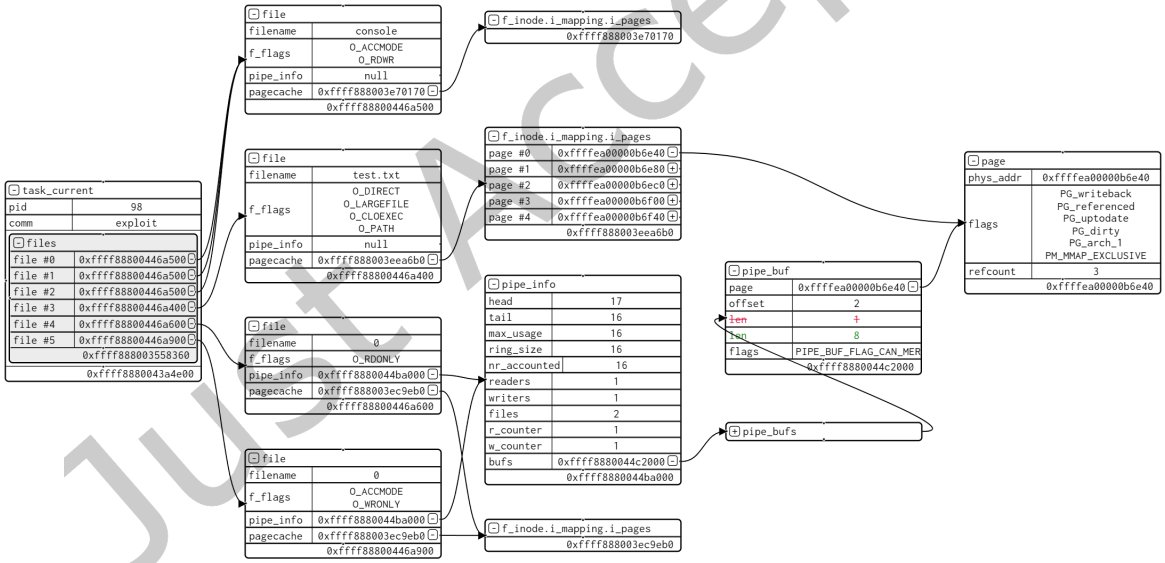
Root cause confirmation. Finally, based on the previous findings, the developer revisits the state view and notices that the transition into reclamation occurs while only a read lock is held. A subsequent source-level inspection confirms that retiring the node on this path requires the updater to hold the write lock, identifying the root cause of StackRot.

[3]Once the ViewCL library predefines common data structures and subsystems, such coding effort will be further eliminated.

Just Accepted



(a) State diff comparing the kernel state before and after a `splice` invocation, revealing that a cached page of the file becomes shared with a pipe buffer.



(b) State diff comparing the kernel state before and after the subsequent pipe `write`, revealing that the shared file-backed page is overwritten.

Fig. 8. Visualizing the Dirty Pipe-related state changes during a deterministic buggy workload. ViewCL plots the current thread's file descriptor table as well as the cached pages of all files and pipes. ViewQL trims and collapses irrelevant objects to focus attention and improve clarity.

Just Accepted

Case 2: Page cache shared between file and pipe. CVE-2022-0847 [88], known as Dirty Pipe, is a local privilege escalation vulnerability whose visible symptom is that a data write to a pipe overwrites the cached contents of a file opened read-only. It arises on the `splice`/`write` path, where the kernel may mistakenly treat a file-backed pipe buffer as mergeable, causing the subsequent pipe `write` to reuse that file-backed buffer instead of allocating a fresh anonymous pipe buffer. Here, `splice` transfers file data into a pipe by sharing pages from the file's page cache with the pipe buffers. Diagnosing this bug is difficult because the visible corruption emerges from cross-subsystem interaction among the file page cache, pipe buffers, and the `splice`/`write` sequence, rather than from a single obvious write path. Moreover, the underlying bug had been present since Linux 4.9 but remained harmless until Linux 5.8, where a later change made it exploitable without directly touching pipe buffers and thus obscured the actual faulty logic.

Initial symptom. A deterministic buggy workload first invokes `splice` to transfer data from a file into a pipe and then writes different data to the same pipe. The developer observes that the later pipe write appears in the supposedly read-only file. This symptom suggests that a later pipe operation corrupts state created earlier during the file-to-pipe transfer, but it does not yet reveal which kernel object carries the corruption.

Hypothesis H1: `splice` *does not construct the zero-copy transfer correctly.* By inspecting the workload, the developer quickly suspects the `splice` syscall, which is the point where the file and the pipe become correlated. To understand exactly how `splice` changes the kernel state, the developer implements a roughly 60-line ViewCL program to visualize the opened files and pipes of the current process. Since `splice` constructs the zero-copy transfer by associating a file page with a pipe buffer, the developer naturally includes both kinds of objects in the visualization.

The number of cached pages remains large even for a simplified workload. To focus on the relevant state, the developer asks Visualinux to:

Display only those `page` *objects that are shared between files and pipes.*

Large language models can synthesize a ViewQL program equivalent to the following one, narrowing the object graph to a visually tractable size:

```
file_pgc = SELECT file->pagecache FROM *          // Find pages belonging to any file
file_pgs = SELECT page FROM REACHABLE(file_pgc)

pipe_buf = SELECT pipe_inode_info->bufs FROM *  // Find pages belonging to any pipe
pipe_pgs = SELECT page FROM REACHABLE(pipe_buf)

UPDATE pipe_pgs \ file_pgs WITH trimmed: true   // Trim pages except for shared ones
```

In the workload we debugged, this filter leaves only one shared page, which appears on the right side of the plot.

Conclusion: H1 rejected. As Figure 8a shows, comparing the states before and after a `splice` invocation reveals that one of the file's cached pages becomes shared with a pipe buffer. This observation is expected for zero-copy transfer and therefore does not, by itself, indicate a bug. Instead, it raises the next question of what the subsequent pipe `write` does to the shared page backed by the file.

Hypothesis H2: The subsequent pipe `write` *drives the shared page into an abnormal state.* To test this possibility, the developer examines the state diff before and after invoking the pipe `write`. By stepping through this syscall and observing the resulting diff, the developer captures the moment when the unexpected modification reaches the shared file page cache.

Conclusion: H2 supported. As Figure 8b shows, the `write` syscall extends the existing file-backed buffer and increases its `len` field instead of allocating a new anonymous buffer. This is the crucial clue: such reuse implies that the `splice`-introduced file-backed buffer still carries mergeable state.

Just Accepted

Root cause confirmation. The developer therefore turns to the pipe buffer's metadata and notices the unexpected `CAN_MERGE` flag on that buffer. A detailed source code inspection then reveals that `copy_page_to_iter_pipe()` does not properly initialize the `flags` field, allowing a stale `CAN_MERGE` flag to persist. Finally, the developer can fully explain why this bug is exploitable: the attacker first fills and drains the pipe so that reusable pipe buffer slots retain mergeable state, then uses `splice` to insert a reference to the target file page into one of those slots. Because the buffer flags are not reinitialized, the subsequent pipe `write` still treats the file-backed buffer as mergeable and overwrites the shared file page instead of allocating a fresh anonymous buffer.

Summary. Across both cases, GDB remains responsible for execution control, breakpoint placement, and source-level inspection, whereas Visualinux is used to visualize relevant objects, narrow the search space, and test intermediate hypotheses. Moreover, the key evidence lies in relationships among objects that span multiple subsystems and execution points, rather than scattered suspicious values. These cases therefore require the developer to correlate objects across subsystems and follow how they change over time before confirming the code-level cause.

5.5 A Case Study of Bug Diagnosis in an Unfamiliar Subsystem

We also conduct a controlled case study on debugging a high-severity Linux CVE in the `io_uring` subsystem. This study examines whether Visualinux can help developers with only partial subsystem familiarity understand the bug-relevant objects and mechanisms, form plausible hypotheses from a reproducible symptom, and progressively narrow the problem space toward the root cause.

Experimental setup. CVE-2024-0582 [90] is a kernel-to-user resource leak in the `io_uring` subsystem [28] that can be exploited to overwrite arbitrary files through leaked pages. The root cause is a cross-subsystem lifetime inconsistency: after a memory-mapped `io_uring` buffer is unregistered, the backing page can be released while the corresponding user mapping remains intact. The workload fills memory-mapped `io_uring` buffers with canaries, then performs a sequence of non-`io_uring` operations. After partially unregistering memory-mapped buffers, canary corruption is observed.

We recruit a non-author PhD student who has basic Linux kernel familiarity and prior experience with GDB, but only limited prior knowledge of `io_uring` and no knowledge of this CVE. The participant is allowed to inspect the kernel source code, read the provided `io_uring` documentation, modify the workload, and debug with GDB and Visualinux, but is prohibited from consulting CVE-related articles or bug reports. AI chatbots and agents may be used only to answer factual questions about the code, trace, and documentation. Direct questions about the bug or its diagnosis, as well as web searches, are prohibited.

To avoid confounding the usefulness of Visualinux with the participant's familiarity with its interface and DSLs, we do not require the participant to manually write GDB scripts or ViewCL programs. Instead, when the participant requests inspection of a particular part of the state or comparison of states across specified execution points, we provide the corresponding textual outputs or Visualinux plots without exposing the underlying implementation details. The case study is limited to 2 hours; if the participant remains stuck for more than 15 minutes, we provide only interpretations of already observed facts and do not suggest the next hypothesis to test.

Debugging trajectory. Table 4 lists the major questions and hypotheses raised by the participant during the case study. Below, we focus on the turning points that redirected the debugging process.

Phase 1: establish a working mental model of the subsystem. The participant first reproduces the workload and confirms the visible symptom: the registered buffers are initially intact, but become corrupted only after buffer unregistration and the subsequent I/O operations unrelated to `io_uring`. At this point, however, the participant still lacks a usable mental model of `io_uring` and struggles to diagnose the bug. Therefore, the participant proposes and answers *Q1*, *Q2*, and *Q3* in Table 4.

Just Accepted

Table 4. Key questions and hypotheses in the exploratory debugging exercise on CVE-2024-0582. Minor local checks are omitted.

ID	Question / Hypothesis	Evidence	Outcome
Q1	What kernel object corresponds to the user-level `io_uring` instance?	● ▲	⊙ Answered
Q2	How are registered buffers represented inside an `io_uring` context?	◇	⊙ Answered
Q3	How is the `io_uring` context connected to the process's file descriptors?	◇	⊙ Answered
H1	The workload misconfigures the `io_uring` context, so the buffers are already malformed after initialization.	● ◇	⊗ Rejected
H2	Unexpected `io_uring` data flow on the submission/completion path corrupts the registered buffers.	▲ ◇	⊗ Rejected
H3	The unrelated I/O operations, rather than buffer unregistration itself, are the primary cause of the corruption.	▲	◓ Partially supported
H4	Buffer unregistration leaves the `io_uring` context's own bookkeeping inconsistent.	▲ ◇	⊗ Rejected
H5	The `mmap`ed memory region remains mapped after buffer unregistration.	◇	⊘ Supported
Q4	What happens to the backing page when the buffer is unregistered but the mapping remains?	◇	⊙ Answered
Q5	What does the unregistration path do to that page at the code level?	□ ▲	◉ Confirmed

Evidence symbols: ● documentation or tutorials; □ kernel source inspection; ▲ runtime execution or workload modification; ◇ Visualinux plots or diffs. Combined evidence symbols indicate that multiple sources are used together.
Outcome symbols: ⊙ answered; ⊘ supported; ◓ partially supported; ⊗ rejected; ◉ confirmed.

By stepping through the `io_uring_setup` syscall with GDB, the participant answers *Q1* and learns that the kernel creates an `io_ring_ctx` object for the user-level `io_uring` instance. Raw GDB output soon becomes hard to follow: long pointer chains and nested structures make it difficult to see how the relevant objects are organized. The participant therefore requests Visualinux plots after context initialization and buffer registration. These views answer *Q2* by showing that an `io_ring_ctx` manages an array of buffer groups, each of which contains a buffer ring for the group's buffer list, and that groups backed by user mappings carry the `PBUF_RING_MMAP` flag. Later, while revisiting the workload and source code, the participant proposes and answers *Q3*, clarifying that `ring_fd` connects the process to the `io_uring` context through a special file object whose `private_data` points to `io_ring_ctx`.

Phase 2: rule out early explanations and narrow the suspicious interval. With the working mental model in place, the participant forms and tests two early explanations, *H1* and *H2*, for the symptom. *H1* attributes the corruption to incorrect `io_uring` setup, but is rejected after reinspecting the post-registration `io_ring_ctx` state and cross-checking the workload against the provided documentation: no evidence suggests that the context or its buffer groups are malformed. *H2* attributes the corruption to unexpected activity on the normal data submission/completion path, but is also rejected after stepping through the workload and observing that the corresponding queues remain unchanged and that no registered buffer is consumed for I/O.

The participant then turns to a narrower hypothesis *H3*: the unrelated I/O operations, rather than buffer unregistration itself, are the primary cause of the symptom. To test this possibility, the participant modifies the workload and removes the unrelated I/O sequence either before or after buffer unregistration. Removing the pre-unregistration sequence does not affect the symptom, whereas removing the post-unregistration one makes the corruption disappear. These results partially support *H3*: the later unrelated I/O exposes the corruption, but the suspicious state transition must already have occurred around buffer unregistration.

Phase 3: reveal the cross-subsystem lifetime inconsistency. The participant next checks *H4*: the unregistration path leaves the `io_uring` buffer management in an erroneous state. By stepping through the corresponding

Just Accepted

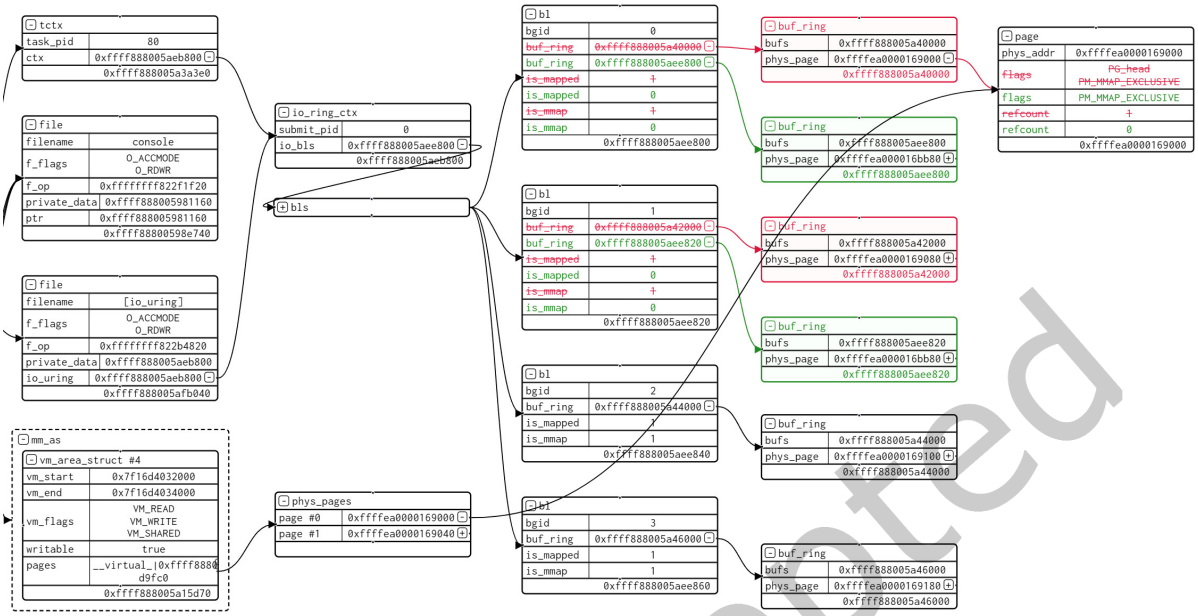


Fig. 9. Differential object graph produced by Visualinux, which compares the kernel states before and after an `io_uring` buffer unregistration. The target buffer disappears from the `io_uring` context and the backing page is released, but the virtual memory mapping remains unchanged. This mismatch exposes the kernel-to-user resource leak behind CVE-2024-0582. Process information on the left side of the plot is cropped for clarity.

`io_uring_register` syscall (with an `UNREGISTER` opcode) with GDB and comparing the relevant buffer group before and after the syscall invocation with Visualinux diff plots, the participant rejects *H4*: the target buffer is removed from the `io_uring` context as expected. This eliminates explanations confined to `io_uring`'s local bookkeeping, but still does not explain why user-space access persists.

The participant therefore revisits the workload and source code and shifts attention to the cross-subsystem connection between the buffer and the process address space: the target buffer is explicitly `mmap`ed into user space. This motivates *H5*, namely whether buffer unregistration removes the buffer from the `io_uring` context but fails to tear down the user mapping. Comparing the process address space before and after buffer unregistration supports *H5* by showing that the corresponding virtual memory region remains mapped. This identifies a lifetime mismatch between buffer unregistration and the surviving user memory mapping, but the mapping alone does not show whether the backing page remains valid. The participant then asks *Q4* and requests a combined view of the `io_uring` context, the process address space, and the backing physical pages across the same interval. As shown in Figure 9, the combined view answers *Q4* by revealing the key inconsistency: when the buffer is unregistered (`is_mapped` is set to false) and released (the buffer ring releases its reference to the physical page, and the page's reference count drops to zero), the virtual memory region still maps to that page.

This plot clarifies the state-level diagnosis: the bug involves a cross-subsystem lifetime mismatch between buffer management and memory mapping. Finally, the participant answers *Q5* by returning to GDB and the kernel source code, confirming that the unregistration path calls `folio_put` on the page, which retires the page into the allocator without tearing down the user mapping. In other words, buffers registered with `PBUF_RING_MMAP` can still be unregistered through a generic path that does not account for the user mapping. As a result, a freed

Just Accepted

physical page in kernel space remains reachable from a user-level process, leading to a kernel-to-user resource leak.

Summary. The case study took about 70 minutes in total, and the participant was never stuck for more than 15 minutes, so no explicit hints were needed. From the recorded trajectory under our controlled conditions, we draw two findings about how bug diagnosis proceeds in an unfamiliar subsystem and where Visualinux contributes most.

Finding 1: To diagnose an unfamiliar subsystem, the first challenge is building a working mental model. Before the participant can form productive hypotheses from the symptom, the first questions are structural: which kernel object corresponds to the user-visible `io_uring` instance, how registered buffers are organized inside that object, and how the context is connected to the process through `ring_fd`. These questions do not yet explain the corruption, but they establish a working model of the subsystem and of what its normal state should look like, which is necessary for identifying what later goes wrong. At this stage, Visualinux helps because its plots make the relevant objects and their relations easier to understand than raw debugger output.

Finding 2: Visualinux helps most when the task is to understand object organization and cross-subsystem correlations. When the participant needs to understand how registered buffers are organized, how the `io_uring` context is connected to process state, and how the mapping and backing page evolve across buffer unregistration, raw GDB output becomes difficult to follow. At those points, the participant turns to Visualinux plots for a more coherent view of the subsystem state. The most decisive evidence comes from a combined cross-subsystem view that reveals a released page still reachable through a surviving user mapping. This division of labor is consistent with the retrospective analyses in Section 5.4: GDB controls execution and verifies code-level causes, whereas Visualinux helps developers understand relationships across multiple objects and subsystem boundaries.

Discussion. Although this case study abstracts away the participant's familiarity with Visualinux, the recorded trajectory suggests that the main effort lies in deciding what state to inspect, which in turn depends on forming useful debugging hypotheses. This is consistent with the design rationale in Sections 3.1 and 3.2: once common views are available in ViewCL, practical use of Visualinux should largely reduce to lightweight ViewQL refinement or direct natural-language requests.

5.6 Performance

We evaluate the performance of Visualinux on two representative debugging scenarios:

(1) *GDB (QEMU)*: Attach GDB to a local QEMU instance running an x86_64 kernel with a root disk image and two virtual CPUs in the Tiny Code Generator (TCG) mode. The kernel version is 6.1.25. This is a typical setting for debugging kernel functionality and data structures.
(2) *KGDB (Raspberry Pi 400)*: Attach GDB to a remote host running KGDB on a Raspberry Pi 400 (Ubuntu 22.04 AArch64). The kernel version is 6.1.0-rpi8. This is a typical setting for debugging real embedded systems.

We implement a workload (~500 LOC) that creates five processes (each process creates two threads), with each thread repeatedly performing IPC operations, mapping/unmapping files and anonymous pages, etc. We evaluate all plot generation tasks in Table 2 and collect runtime statistics for the most time-consuming step of ViewCL program execution[4].

Table 5 shows the performance evaluation results. The major performance bottleneck is evaluating the C expressions (e.g., `node.mr64.slot`) embedded in ViewCL. In the localhost GDB/QEMU setting, the latency is generally acceptable for interactive use across all evaluated figures.

In the remote KGDB setting on Raspberry Pi 400, Visualinux is noticeably slow for most figures, especially larger plots (e.g., Fig 3-6), though smaller plots remain tolerable for interactive debugging. This slowdown comes

[4]ViewQL query evaluation and front-end rendering incur negligible overhead.

Just Accepted

Table 5. Performance results of plotting the representative ULK figures. The table shows object graph construction overhead in milliseconds for two debugging scenarios: GDB with QEMU emulation and KGDB on Raspberry Pi 400. For each scenario, we report total execution time, per-object overhead, and per-kilobyte overhead. The last 3 rows are results of the additional plots mentioned in Section 5.1.

#	Figure	GDB (QEMU)			KGDB (rpi-400)		
		Total	**Per Obj**	**Per KB**	**Total**	**Per Obj**	**Per KB**
1	Fig 3-4. process tree	59.6	0.45	17.4	7,025.1	23.73	913.7
2	Fig 3-6. PID hash tables	155.3	0.17	23.2	20,904.3	9.14	870.2
3	Fig 4-5. IRQ descriptors	39.4	0.35	17.2	4,309.6	16.38	924.6
4	Fig 6-1. dynamic timers	181.2	0.15	18.7	8,096.4	6.71	825.4
5	Fig 7-1. CFS scheduler runqueue	13.7	0.22	14.1	33.6	8.41	1,202.3
6	Fig 8-2. buddy system and pages	60.2	0.12	16.8	3,213.8	6.63	894.2
7	Fig 8-4. slab allocator	12.8	0.42	13.3	894.9	29.97	945.3
8	Fig 9-2. process address space	48.6	0.29	32.0	1,402.1	8.44	930.3
9	Fig 11-1. signal handler	45.6	0.31	26.7	1,392.5	9.94	852.4
10	Fig 12-3. process fd array	10.9	0.13	87.5	24.7	0.34	1,411.7
11	Fig 13-3. device driver	326.0	0.87	40.9	7,602.3	20.43	952.5
12	Fig 14-3. block device	80.3	1.11	36.7	1,972.1	27.38	903.3
13	Fig 15-1. file page cache	84.7	0.37	42.8	86.0	1.17	931.9
14	Fig 16-2. file memory mapping	10.1	0.12	31.4	36.2	0.51	1,101.0
15	Fig 17-1. page reverse mapping	62.4	0.31	24.7	2,322.6	11.55	921.4
16	Fig 17-6. swap area management	34.3	0.33	51.6	559.0	5.53	840.6
17	Fig 19-1/2. IPC management	19.0	0.52	12.5	1,078.8	41.46	861.7
18	workqueue example	11.9	0.33	10.5	1,779.5	26.16	810.1
19	from process to VFS	40.2	0.31	22.1	659.3	8.13	819.0
20	socket connection	20.8	0.18	42.5	17.4	0.25	1,062.5

from both the slower embedded processor and KGDB communication delays: retrieving an object is approximately 50 times slower than in the GDB/QEMU setting, and even retrieving a `uint64` via KGDB takes about 5 ms. These results suggest that Visualinux is best suited to focused debugging objectives, where developers inspect a small, task-relevant substate rather than a complete view of a large system module.

6 Discussion

6.1 Methodology Behind Visualinux

The key insight underlying Visualinux is that program state *simplification* has long been an implicit yet fundamental practice in kernel debugging and understanding. Treating the kernel state as an object graph, developers simplify it for a specific debugging objective by repeatedly applying the following three operations:

(1) **Prune.** To produce small states, developers prune (remove) fields, objects, and relations that are irrelevant to the debugging objective. Figure 10b shows an example where most fields in the `task_struct` are removed for brevity, except for the process identity and scheduling-related information.
(2) **Flatten.** An indirect connection between objects can span many hops, yet the intermediate objects are irrelevant to the debugging objective. Developers *flatten* (or compress) such a path as a direct link between objects. For example, Figure 10a flattens the long path from a `task_struct` to its associated sockets, skipping intermediate objects across three subsystems.

Just Accepted

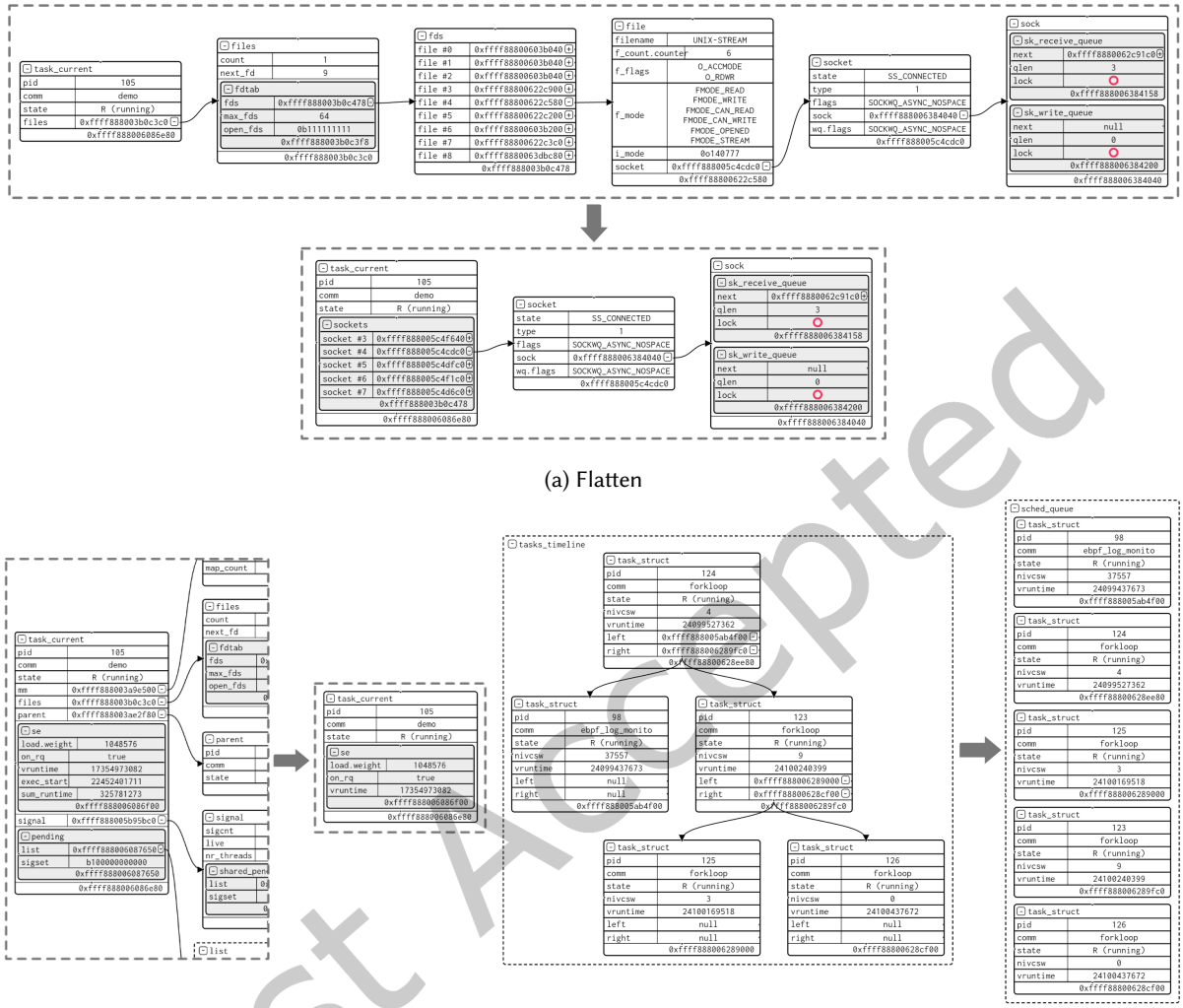


Fig. 10. Illustration of the fundamental operations for program state simplification.

(3) **Distill.** Data structures can be difficult to read in their node-pointer forms. For example, memory-mapped regions are organized as a tree for faster queries. However, for debugging objectives that do not directly concern the tree structure, a flattened linear list of intervals is more readable. Therefore, developers intentionally *distill* a kernel data structure into a compact format that retains the objects' logical relations. Figure 10c provides such an example.

Starting from a root object, applying these operations yields a visually clear view of the kernel state. For decades, kernel developers have *implicitly* followed this procedure to simplify kernel states for debugging. The Linux kernel includes GDB scripts to (textually) visualize kernel data structures [11, 20], and tools like drgn [2, 13]

Just Accepted

are designed to enhance the versatility and efficiency of this procedure. Visualinux follows this established approach by encoding the three operations into the core semantics of ViewCL:

(1) The declarative nature of ViewCL encodes the *prune* operation. Since *most* state data is irrelevant to a specific debugging objective, ViewCL specifies what should remain in the plot rather than enumerating everything to remove.
(2) ViewCL introduces dot-connected fields to *flatten* a data path. For example, field definitions like `Text(parent.pid)` or `Link(files.fdtab.fds)` naturally bypass the task-irrelevant intermediate links between objects.
(3) ViewCL implements *distill* by providing converter functions. These functions take a container data structure, such as a red-black tree, hash list, or extensible array, and convert it into conceptually simpler abstract data types: either an ordered sequence or an unordered set.

The design of ViewQL is guided by the same simplification principles, but applies them at the display level after an object graph has been constructed. For instance, developers can remove irrelevant objects by setting *trimmed*, compress local details by setting *collapsed*, and keep attention on task-relevant regions of the graph through selection and set operations. Thus, ViewQL complements ViewCL by providing a finer-grained refinement layer for adapting an existing state view to the current debugging objective.

6.2 Limitations and Future Work

Visualinux is designed as a complement to, rather than a replacement for existing kernel debugging tools. While it provides practical visualization of runtime states and state changes across execution, Visualinux fundamentally depends on memory snapshots, which must be obtained through other debugging tools such as GDB or crash dumps. Additionally, Visualinux's effectiveness is limited in scenarios involving non-deterministic bugs or concurrency issues, as it assumes trace reproducibility, which is precisely the property that such bugs often lack.

Another limitation is the maintenance overhead introduced by Visualinux. Although our layered abstraction (including the natural-language interface) makes state visualization accessible to most kernel developers, creating comprehensive ViewCL programs for the vast array of kernel components remains a substantial long-term undertaking. While this task is considerably simpler than using traditional scripting approaches, it still requires substantial effort from tool maintainers and kernel experts to build a comprehensive library that evolves alongside the Linux kernel.

Finally, while Visualinux is specifically designed and implemented for the Linux kernel, the underlying insights and methodologies are not inherently limited to this domain. The fundamental debugging practice of state simplification applies similarly to understanding complex program states in other system software. We plan to explore generalizing our approach to create debugging tools for broader software systems.

7 Related Work

Interactive debugging. Various automated debugging solutions have been proposed, including delta debugging [87, 109, 120], statistical debugging [37, 69, 78, 92, 116], causal tracing [56, 68, 82, 111], and reverse debugging [48, 60, 108, 119]. These solutions aim to improve the efficiency of fault reproduction or root cause diagnosis. However, past research indicates that automated debugging tools offer limited assistance when developers face challenging debugging tasks [94, 113], especially those beyond the capabilities of the tools. This paper focuses on interactive debugging [34, 46, 72, 73]. Since full automation remains a distant goal, we believe that efficient approaches to interactive debugging are indispensable for developers.

Whyline [72, 73] allows developers to ask why and why-not questions about program output and generates answers using various program analyses, providing an interactive way to trace data dependencies. Hypothesizer [34] records program behaviors and user actions during bug reproduction and generates possibly relevant hypotheses about program state and behavior, continuously narrowing down the problem scope through interaction.

Just Accepted

Although helpful for program understanding, these approaches focus primarily on explaining observed outputs and behaviors, and thus offer limited assistance with bugs whose key evidence lies in complex data structures and object relations. Moreover, they do not readily transfer to complex systems like the Linux kernel, where developers must reason about a massive live state with low-level data representations and weak interactive observability.

Visualization-based interactive debugging. Past research has shown that program visualization can effectively help developers understand program state and behavior [50, 65, 79, 83, 95, 99, 100, 104]. Multiple research projects have focused on constructing visual interactive debuggers, as one of the core purposes of interactive debugging is program understanding [15, 39, 49, 64, 74, 91, 93, 121]. Additionally, interactive tools have been designed specifically for understanding algorithms, data structures, and call graphs [1, 9, 75].

However, traditional visualization-based debugging techniques are not suitable for complex software systems with excessively large program states, such as the Linux kernel. The lack of suitable abstractions in these techniques forces developers to work with complete diagrams of entire massive objects and handle long, deep pointer chains. Directly applying such techniques to the Linux kernel would thus be impractical.

Interactive debugging for kernels. Kernel debugging has been studied for many years [41, 42, 60, 68, 71, 106]. However, most existing research focuses on automated debugging techniques and has overlooked the importance of interactive debugging.

During the long-term evolution of the Linux kernel, a number of tools have been proposed to help developers debug and understand the kernel from various perspectives, including interactive inspection [2, 11, 27], runtime validation [16–18, 24], logging [22], and tracing [19, 26]. Visualinux focuses particularly on visualizing program state, taking a different approach from scripting tools like GDB scripts [11, 27] and drgn [2]. It also offers high-level data structure abstractions not present in existing state analysis tools [10, 12].

drgn [2, 13] is a modern, programmable kernel debugger that enables developers to debug the kernel in a Pythonic way. It offers several Linux-specific helper functions to simplify access to various kernel data structures, such as IDRs and maple trees [13]. drgn also supports writing reusable debugging scripts. However, as a textual tool, drgn does not provide intuitive illustrations of nested data structures. It offers limited assistance in understanding the kernel state, which is where Visualinux excels.

Visualinux is not the first work to utilize visualization for understanding the Linux kernel. However, previous approaches focus on various domains such as traces [6, 26], call graphs [3], and source code organization [7, 102], which are orthogonal to Visualinux.

High-level query languages for debugging. Visualinux is not the first solution to leverage a high-level query language to help developers express and answer debugging questions at a higher level of abstraction. However, many existing approaches focus on designing abstractions for execution traces [51, 61, 63, 77, 82, 86, 96]. They help developers query temporal event streams, whereas Visualinux focuses on simplifying and comparing runtime state snapshots.

Object query languages [58, 59, 98, 112] access runtime objects through relational interfaces. PiCO QL [59], for instance, defines a relational representation of accessible Linux kernel data structures and allows developers to customize views for system resources. Our goal is broader program understanding rather than relational querying alone. Accordingly, Visualinux differs from this line of work by combining a declarative language for object graph construction with a separate relational interface for selective visualization.

Differencing approaches. Differencing techniques have been extensively studied across various domains. Code differencing is applied to code evolution analysis [36, 53–55, 70, 97, 110, 115, 117], refactoring detection [103, 107], and automatic program repair [118]. Model differencing is used for comparing software designs [84, 85, 114]. Log differencing analyzes execution patterns and system behaviors [35, 38, 43, 62].

Just Accepted

While these approaches provide valuable insights into program evolution and change analysis, their scope fundamentally differs from Visualinux. To our knowledge, this paper proposes the first runtime state differencing approach that addresses the challenge of understanding instantaneous changes during system execution. Existing object differencing libraries target structured data comparison [30–32] and are not designed for object graphs extracted from live Linux kernel states, where complex data relationships include cross-references and long pointer chains.

8 Conclusion

This paper presents Visualinux, a debugging framework that enables practical visual debugging for the Linux kernel. By separating object graph construction, state view customization, and state diff generation, Visualinux provides an interactive visualization interface that helps developers comprehend kernel objects and their runtime evolution. Its GDB integration and chat-based interface also make it readily applicable to real-world debugging tasks.

References

[1] 2011. *Data Structure Visualization.* https://www.cs.usfca.edu/~galles/visualization/
[2] 2019. *A kernel debugger in Python: drgn.* https://lwn.net/Articles/789641/
[3] 2021. *Kernel_visualization.* https://github.com/x2c3z4/kernel_visualization
[4] 2022. *CFS Scheduler - The Linux Kernel documentation.* https://docs.kernel.org/6.1/scheduler/sched-design-CFS.html
[5] 2022. *Introducing Maple Trees [LWN.net].* https://lwn.net/Articles/845507/
[6] 2022. *SchedViz.* https://github.com/google/schedviz
[7] 2023. *Interactive map of Linux kernel.* https://makelinux.github.io/kernel/map/
[8] 2023. *StackRot (CVE-2023-3269): Linux kernel privilege escalation vulnerability.* https://github.com/lrh2000/StackRot
[9] 2023. *visualising data structures and algorithms through animation - VisuAlgo.* https://visualgo.net
[10] 2024. *crash(8) — Linux manual page.* https://man7.org/linux/man-pages/man8/crash.8.html
[11] 2024. *Debugging kernel and modules via GDB.* https://www.kernel.org/doc/html/latest/process/debugging/gdb-kernel-debugging.html
[12] 2024. *Documentation for Kdump - The Kexec-based Crash Dumping Solution.* https://www.kernel.org/doc/html/latest/admin-guide/kdump/kdump.html
[13] 2024. *Drgn documentation.* https://drgn.readthedocs.io/en/latest/index.html
[14] 2024. *GDB: The GNU Project Debugger.* https://sourceware.org/gdb/
[15] 2024. *GDBGUI.* https://github.com/cs01/gdbgui
[16] 2024. *Kernel Address Sanitizer (KASAN).* https://www.kernel.org/doc/html/latest/dev-tools/kasan.html
[17] 2024. *Kernel Electric-Fence (KFENCE).* https://www.kernel.org/doc/html/latest/dev-tools/kfence.html
[18] 2024. *Kernel Memory Leak Detector.* https://www.kernel.org/doc/html/latest/dev-tools/kmemleak.html
[19] 2024. *Linux Tracing Technologies.* https://www.kernel.org/doc/html/latest/trace/index.html
[20] 2024. *linux/scripts/gdb at master - torvalds/Linux.* https://github.com/torvalds/linux/tree/master/scripts/gdb
[21] 2024. *Maple Tree - The Linux Kernel documentation.* https://docs.kernel.org/core-api/maple_tree.html
[22] 2024. *Message logging with printk.* https://www.kernel.org/doc/html/latest/core-api/printk-basics.html
[23] 2024. *RCU Concepts - The Linux Kernel documentation.* https://docs.kernel.org/RCU/rcu.html
[24] 2024. *Runtime locking correctness validator.* https://www.kernel.org/doc/html/latest/locking/lockdep-design.html
[25] 2024. *tmux/tmux: tmux source code.* https://github.com/tmux/tmux
[26] 2024. *Traceshark.* https://github.com/cunctator/traceshark
[27] 2024. *Using KGDB, KDB and the kernel debugger internals.* https://www.kernel.org/doc/html/latest/dev-tools/kgdb.html
[28] 2025. *A guide to io_uring [LWN.net].* https://lwn.net/Articles/932568/
[29] 2025. *A thorough introduction to eBPF.* https://lwn.net/Articles/740157/
[30] 2025. *deep-diff.* https://github.com/flitbit/diff
[31] 2025. *java-object-diff.* https://github.com/SQiShER/java-object-diff
[32] 2025. *jsondiffpatch.* https://github.com/benjamine/jsondiffpatch
[33] 2025. *Memory Management Documentation - The Linux Kernel Documentation.* https://docs.kernel.org/mm/index.html
[34] Abdulaziz Alaboudi and Thomas D. Latoza. 2023. Hypothesizer: a hypothesis-based debugger to find and test debugging hypotheses. In *Proceedings of the 36th Annual ACM Symposium on User Interface Software and Technology* (San Francisco, CA, USA) *(UIST '23)*. Association for Computing Machinery, New York, NY, USA, Article 73, 14 pages. doi:10.1145/3586183.3606781

Just Accepted

[35] Hen Amar, Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2018. Using finite-state models for log differencing. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 49–59. doi:10.1145/3236024.3236069

[36] Taweesup Apiwattanapong, Alessandro Orso, and Mary Jean Harrold. 2004. A differencing algorithm for object-oriented programs. In *Proceedings of the 19th IEEE International Conference on Automated Software Engineering (ASE '04)*. IEEE Computer Society, USA, 2–13.

[37] Joy Arulraj, Po-Chun Chang, Guoliang Jin, and Shan Lu. 2013. Production-run software failure diagnosis via hardware performance counters *(ASPLOS '13)*. Association for Computing Machinery, New York, NY, USA, 101–112. doi:10.1145/2451116.2451128

[38] Lingfeng Bao, Nimrod Busany, David Lo, and Shahar Maoz. 2019. Statistical log differencing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 851–862. doi:10.1109/ASE.2019.00084

[39] David B. Baskerville. 1985. *Graphic presentation of data structures in the DBX debugger.* Technical Report UCB/CSD-86-260. EECS Department, University of California, Berkeley.

[40] Kaiwan N Billimoria. 2021. *Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization.* Packt Publishing.

[41] Tegawendé F. Bissyandé, Laurent Réveillère, Julia L. Lawall, and Gilles Muller. 2012. Diagnosys: automatic generation of a debugging interface to the Linux kernel. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE '12)*. Association for Computing Machinery, New York, NY, USA, 60–69. doi:10.1145/2351676.2351686

[42] Martin J. Bligh and Mathieu Desnoyers. 2010. Linux kernel debugging on Google-sized clusters. https://api.semanticscholar.org/CorpusID:262330426

[43] Kirill Bogdanov and Neil Walkinshaw. 2009. Computing the structural difference between state-based models. In *2009 16th Working Conference on Reverse Engineering*. 177–186. doi:10.1109/WCRE.2009.17

[44] Jeff Bonwick. 1994. The slab allocator: an object-caching kernel. In *USENIX Summer 1994 Technical Conference (USENIX Summer 1994 Technical Conference)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenix-summer-1994-technical-conference/slab-allocator-object-caching-kernel

[45] D. Bovet and M. Cesati. 2005. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly Media.

[46] Brian Burg, Richard Bailey, Amy J. Ko, and Michael D. Ernst. 2013. Interactive record/replay for web application debugging. In *Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology* (St. Andrews, Scotland, United Kingdom) *(UIST '13)*. Association for Computing Machinery, New York, NY, USA, 473–484. doi:10.1145/2501988.2502050

[47] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. 2009. A review of overview+detail, zooming, and focus+context interfaces. *Comput. Surveys* 41, 1, Article 2 (Jan. 2009), 31 pages. doi:10.1145/1456650.1456652

[48] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: reverse debugging of failures in deployed software. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*. USENIX Association, Carlsbad, CA, 17–32. https://www.usenix.org/conference/osdi18/presentation/weidong

[49] Jeffrey K. Czyz and Bharat Jayaraman. 2007. Declarative and visual debugging in Eclipse. In *Proceedings of the 2007 OOPSLA Workshop on Eclipse Technology EXchange* (Montreal, Quebec, Canada) *(eclipse '07)*. Association for Computing Machinery, New York, NY, USA, 31–35. doi:10.1145/1328279.1328286

[50] Sharon Ellershaw and Michael Oudshoorn. 1998. Program visualization - the state of the art. (06 1998).

[51] Úlfar Erlingsson, Marcus Peinado, Simon Peter, and Mihai Budiu. 2011. Fay: extensible distributed tracing from kernels to clusters. In *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles* (Cascais, Portugal) *(SOSP '11)*. Association for Computing Machinery, New York, NY, USA, 311–326. doi:10.1145/2043556.2043585

[52] DeepSeek-AI et al. 2024. DeepSeek-V2: a strong, economical, and efficient mixture-of-experts language model. arXiv:2405.04434 [cs.CL] https://arxiv.org/abs/2405.04434

[53] Jean-Rémy Falleri, Floréal Morandat, Xavier Blanc, Matias Martinez, and Martin Monperrus. 2014. Fine-grained and accurate source code differencing. In *Proceedings of the 29th ACM/IEEE International Conference on Automated Software Engineering* (Vasteras, Sweden) *(ASE '14)*. Association for Computing Machinery, New York, NY, USA, 313–324. doi:10.1145/2642937.2642982

[54] Beat Fluri, Michael Wursch, Martin Pinzger, and Harald Gall. 2007. Change distilling: tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 33, 11 (2007), 725–743. doi:10.1109/TSE.2007.70731

[55] Beat Fluri, Michael Würsch, Martin Pinzger, and Harald Gall. 2025. A retrospective of ChangeDistiller: tree differencing for fine-grained source code change extraction. *IEEE Transactions on Software Engineering* 51, 3 (2025), 852–857. doi:10.1109/TSE.2025.3538326

[56] Rodrigo Fonseca, George Porter, Randy H. Katz, and Scott Shenker. 2007. X-Trace: a pervasive network tracing framework. In *4th USENIX Symposium on Networked Systems Design & Implementation (NSDI 07)*. USENIX Association, Cambridge, MA. https://www.usenix.org/conference/nsdi-07/x-trace-pervasive-network-tracing-framework

[57] Free Software Foundation. 2020. *GNU Diffutils: Comparing and Merging Files.* https://www.gnu.org/software/diffutils/ Documents the unified diff format used by Git..

Just Accepted

[58] Marios Fragkoulis, Diomidis Spinellis, and Panos Louridas. 2015. An interactive SQL relational interface for querying main-memory data structures. *Computing* 97, 12 (01 Dec 2015), 1141–1164. doi:10.1007/s00607-015-0452-y

[59] Marios Fragkoulis, Diomidis Spinellis, Panos Louridas, and Angelos Bilas. 2014. Relational access to Unix kernel data structures. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 12, 14 pages. doi:10.1145/2592798.2592802

[60] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *2020 USENIX Annual Technical Conference (USENIX ATC 20)*. USENIX Association, 281–292. https://www.usenix.org/conference/atc20/presentation/ge

[61] Simon F. Goldsmith, Robert O'Callahan, and Alex Aiken. 2005. Relational queries over program traces. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 385–402. doi:10.1145/1094811.1094841

[62] Maayan Goldstein, Danny Raz, and Itai Segall. 2017. Experience report: log-based behavioral differencing. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. 282–293. doi:10.1109/ISSRE.2017.14

[63] Zhenyu Guo, Dong Zhou, Haoxiang Lin, Mao Yang, Fan Long, Chaoqiang Deng, Changshu Liu, and Lidong Zhou. 2011. G2: a graph processing system for diagnosing distributed systems. In *2011 USENIX Annual Technical Conference (USENIX ATC 11)*. USENIX Association, Portland, OR. https://www.usenix.org/conference/usenixatc11/g2-graph-processing-system-diagnosing-distributed-systems

[64] David Hanson and Jeffrey Korn. 1997. A simple and extensible graphical debugger. In *USENIX 1997 Annual Technical Conference (USENIX ATC 97)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/usenix-1997-annual-technical-conference/simple-and-extensible-graphical-debugger

[65] Jane Hoffswell, Arvind Satyanarayan, and Jeffrey Heer. 2018. Augmenting code with in situ visualizations to aid program understanding. In *Proceedings of the 2018 CHI Conference on Human Factors in Computing Systems* (Montreal QC, Canada) *(CHI '18)*. Association for Computing Machinery, New York, NY, USA, 1–12. doi:10.1145/3173574.3174106

[66] Liam Howlett. 2021. *The Maple Tree, A Modern Data Structure for a Complex Problem*. https://blogs.oracle.com/linux/post/the-maple-tree-a-modern-data-structure-for-a-complex-problem

[67] Daniel Jackson. 2012. *Software abstractions: logic, language, and analysis*. The MIT Press.

[68] Dae R. Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. 2023. Diagnosing kernel concurrency failures with AITIA. In *Proceedings of the Eighteenth European Conference on Computer Systems* (Rome, Italy) *(EuroSys '23)*. Association for Computing Machinery, New York, NY, USA, 94–110. doi:10.1145/3552326.3567486

[69] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: a technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 344–360. doi:10.1145/2815400.2815412

[70] Miryung Kim and David Notkin. 2009. Discovering and representing systematic code changes. In *2009 IEEE 31st International Conference on Software Engineering*. 309–319. doi:10.1109/ICSE.2009.5070531

[71] Samuel T. King, George W. Dunlap, and Peter M. Chen. 2005. Debugging operating systems with time-traveling virtual machines. In *2005 USENIX Annual Technical Conference (USENIX ATC 05)*. USENIX Association, Anaheim, CA. https://www.usenix.org/conference/2005-usenix-annual-technical-conference/debugging-operating-systems-time-traveling

[72] Amy J. Ko and Brad A. Myers. 2004. Designing the Whyline: A Debugging Interface for Asking Questions about Program Behavior. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Vienna, Austria) *(CHI '04)*. Association for Computing Machinery, New York, NY, USA, 151–158. doi:10.1145/985692.985712

[73] Amy J. Ko and Brad A. Myers. 2008. Debugging reinvented: asking and answering why and why not questions about program behavior. In *Proceedings of the 30th International Conference on Software Engineering* (Leipzig, Germany) *(ICSE '08)*. Association for Computing Machinery, New York, NY, USA, 301–310. doi:10.1145/1368088.1368130

[74] Tim Kräuter, Harald König, Adrian Rutle, and Yngve Lamo. 2022. The visual debugger tool. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 494–498. doi:10.1109/ICSME55016.2022.00066

[75] Thomas D. LaToza and Brad A. Myers. 2011. Visualizing call graphs. In *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. 117–124. doi:10.1109/VLHCC.2011.6070388

[76] Lucas Layman, Madeline Diep, Meiyappan Nagappan, Janice Singer, Robert Deline, and Gina Venolia. 2013. Debugging revisited: toward understanding the debugging needs of contemporary software developers. In *2013 ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*. 383–392. doi:10.1109/ESEM.2013.43

[77] Geoffrey Lefebvre, Brendan Cully, Christopher Head, Mark Spear, Norm Hutchinson, Mike Feeley, and Andrew Warfield. 2012. Execution mining. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments* (London, England, UK) *(VEE '12)*. Association for Computing Machinery, New York, NY, USA, 145–158. doi:10.1145/2151024.2151044

[78] Ben Liblit, Mayur Naik, Alice X. Zheng, Alex Aiken, and Michael I. Jordan. 2005. Scalable statistical bug isolation *(PLDI '05)*. Association for Computing Machinery, New York, NY, USA, 15–26. doi:10.1145/1065010.1065014

Just Accepted

[79] Tom Lieber, Joel R. Brandt, and Rob C. Miller. 2014. Addressing misconceptions about code with always-on programming visualizations. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (Toronto, Ontario, Canada) *(CHI '14)*. Association for Computing Machinery, New York, NY, USA, 2481–2490. doi:10.1145/2556288.2557409

[80] Hanzhi Liu, Yanyan Jiang, and Chang Xu. 2025. Understanding the Linux kernel, visually. In *Proceedings of the Twentieth European Conference on Computer Systems* (Rotterdam, Netherlands) *(EuroSys '25)*. Association for Computing Machinery, New York, NY, USA, 1044–1060. doi:10.1145/3689031.3696095

[81] Robert Love. 2010. *Linux Kernel Development (Developer's Library), 3rd Edition*. Addison-Wesley Professional.

[82] Jonathan Mace, Ryan Roelke, and Rodrigo Fonseca. 2015. Pivot tracing: dynamic causal monitoring for distributed systems. In *Proceedings of the 25th Symposium on Operating Systems Principles* (Monterey, California) *(SOSP '15)*. Association for Computing Machinery, New York, NY, USA, 378–393. doi:10.1145/2815400.2815415

[83] J.I. Maletic, A. Marcus, and M.L. Collard. 2002. A task oriented view of software visualization. In *Proceedings First International Workshop on Visualizing Software for Understanding and Analysis*. 32–40. doi:10.1109/VISSOF.2002.1019792

[84] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. ADDiff: semantic differencing for activity diagrams. In *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering* (Szeged, Hungary) *(ESEC/FSE '11)*. Association for Computing Machinery, New York, NY, USA, 179–189. doi:10.1145/2025113.2025140

[85] Shahar Maoz, Jan Oliver Ringert, and Bernhard Rumpe. 2011. CDDiff: semantic differencing for class diagrams. In *ECOOP 2011 – Object-Oriented Programming*, Mira Mezini (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 230–254.

[86] Michael Martin, Benjamin Livshits, and Monica S. Lam. 2005. Finding application errors and security flaws using PQL: a program query language. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA) *(OOPSLA '05)*. Association for Computing Machinery, New York, NY, USA, 365–383. doi:10.1145/1094811.1094840

[87] Ghassan Misherghi and Zhendong Su. 2006. HDD: hierarchical delta debugging. In *Proceedings of the 28th International Conference on Software Engineering* (Shanghai, China) *(ICSE '06)*. Association for Computing Machinery, New York, NY, USA, 142–151. doi:10.1145/1134285.1134307

[88] MITRE. 2022. *CVE-2022-0847*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2022-0847

[89] MITRE. 2023. *CVE-2023-3269*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2023-3269

[90] MITRE. 2025. *CVE-2024-0582*. https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2024-0582

[91] T.G. Moher. 1988. PROVIDE: a process visualization and debugging environment. *IEEE Transactions on Software Engineering* 14, 6 (1988), 849–857. doi:10.1109/32.6163

[92] Karthik Nagaraj, Charles Killian, and Jennifer Neville. 2012. Structured comparative analysis of systems logs to diagnose performance problems. In *9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. USENIX Association, San Jose, CA, 353–366. https://www.usenix.org/conference/nsdi12/technical-sessions/presentation/nagaraj

[93] Rainer Oechsle and Thomas Schmitt. 2002. JAVAVIS: automatic program visualization with object and sequence diagrams using the Java debug interface (JDI). In *Software Visualization*, Stephan Diehl (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 176–190.

[94] Chris Parnin and Alessandro Orso. 2011. Are automated debugging techniques actually helping programmers. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis* (Toronto, Ontario, Canada) *(ISSTA '11)*. Association for Computing Machinery, New York, NY, USA, 199–209. doi:10.1145/2001420.2001445

[95] Blaine A. Price, Ronald M. Baecker, and Ian S. Small. 1993. A principled taxonomy of software visualization. *J. Vis. Lang. Comput.* 4 (1993), 211–266.

[96] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. 2022. Debugging the OmniTable way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*. USENIX Association, Carlsbad, CA, 357–373. https://www.usenix.org/conference/osdi22/presentation/quinn

[97] S. Raghavan, R. Rohana, D. Leon, A. Podgurski, and V. Augustine. 2004. Dex: a semantic-graph differencing tool for studying changes in large code bases. In *20th IEEE International Conference on Software Maintenance, 2004. Proceedings.* 188–197. doi:10.1109/ICSM.2004.1357803

[98] Christoph Reichenbach, Yannis Smaragdakis, and Neil Immerman. 2012. PQL: a purely-declarative java extension for parallel programming. In *Proceedings of the 26th European Conference on Object-Oriented Programming* (Beijing, China) *(ECOOP'12)*. Springer-Verlag, Berlin, Heidelberg, 53–78. doi:10.1007/978-3-642-31057-7_4

[99] G.-C. Roman and K.C. Cox. 1993. A taxonomy of program visualization systems. *Computer* 26, 12 (1993), 11–24. doi:10.1109/2.247643

[100] Gruia-Catalin Roman and Kenneth C. Cox. 1992. Program visualization: the art of mapping programs to pictures. In *Proceedings of the 14th International Conference on Software Engineering* (Melbourne, Australia) *(ICSE '92)*. Association for Computing Machinery, New York, NY, USA, 412–420. doi:10.1145/143062.143157

[101] Rami Rosen. 2014. *Linux Kernel Networking: Implementation and Theory*. Apress Berkeley, CA.

[102] Jianjun Shi, Weixing Ji, Jingjing Zhang, Zhiwei Gao, Yizhuo Wang, and Feng Shi. 2019. KernelGraph: understanding the kernel in a graph. *Information Visualization* 18, 3 (2019), 283–296. doi:10.1177/1473871617743239

Just Accepted

[103] Danilo Silva, João Paulo da Silva, Gustavo Santos, Ricardo Terra, and Marco Tulio Valente. 2021. RefDiff 2.0: a multi-language refactoring detection tool. *IEEE Transactions on Software Engineering* 47, 12 (2021), 2786–2802. doi:10.1109/TSE.2020.2968072

[104] J.T. Stasko and C. Patterson. 1992. Understanding and characterizing software visualization systems. In *Proceedings IEEE Workshop on Visual Languages*. 3–10. doi:10.1109/WVL.1992.275790

[105] Michael Stonebraker. 1975. Implementation of integrity constraints and views by query modification. In *Proceedings of the 1975 ACM SIGMOD International Conference on Management of Data* (San Jose, California) *(SIGMOD '75)*. Association for Computing Machinery, New York, NY, USA, 65–78. doi:10.1145/500080.500091

[106] Henrik Stuart, René Rydhof Hansen, Julia L. Lawall, Jesper Andersen, Yoann Padioleau, and Gilles Muller. 2007. Towards easing the diagnosis of bugs in OS code. In *Proceedings of the 4th Workshop on Programming Languages and Operating Systems* (Stevenson, Washington) *(PLOS '07)*. Association for Computing Machinery, New York, NY, USA, Article 2, 5 pages. doi:10.1145/1376789.1376792

[107] Nikolaos Tsantalis, Ameya Ketkar, and Danny Dig. 2022. RefactoringMiner 2.0. *IEEE Transactions on Software Engineering* 48, 3 (2022), 930–950. doi:10.1109/TSE.2020.3007722

[108] Ana-Maria Visan, Kapil Arya, Gene Cooperman, and Tyler Denniston. 2011. URDB: a universal reversible debugger based on decomposing debugging histories. In *Proceedings of the 6th Workshop on Programming Languages and Operating Systems* (Cascais, Portugal) *(PLOS '11)*. Association for Computing Machinery, New York, NY, USA, Article 8, 5 pages. doi:10.1145/2039239.2039251

[109] Guancheng Wang, Ruobing Shen, Junjie Chen, Yingfei Xiong, and Lu Zhang. 2021. Probabilistic delta debugging. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 881–892. doi:10.1145/3468264.3468625

[110] Yuxin Wang, Adam Welc, Lazaro Clapp, and Lingchao Chen. 2023. Last Diff Analyzer: multi-language automated approver for behavior-preserving code revisions. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (San Francisco, CA, USA) *(ESEC/FSE 2023)*. Association for Computing Machinery, New York, NY, USA, 1693–1704. doi:10.1145/3611643.3613870

[111] Lingmei Weng, Peng Huang, Jason Nieh, and Junfeng Yang. 2021. Argus: debugging performance issues in modern desktop applications with annotated causal tracing. In *2021 USENIX Annual Technical Conference (USENIX ATC 21)*. USENIX Association, 193–207. https://www.usenix.org/conference/atc21/presentation/weng

[112] Darren Willis, David J. Pearce, and James Noble. 2006. Efficient object querying for Java. In *Proceedings of the 20th European Conference on Object-Oriented Programming* (Nantes, France) *(ECOOP'06)*. Springer-Verlag, Berlin, Heidelberg, 28–49. doi:10.1007/11785477_3

[113] Xin Xia, Lingfeng Bao, David Lo, and Shanping Li. 2016. "Automated debugging considered harmful" considered harmful: a user study revisiting the usefulness of spectra-based fault localization techniques with professionals using real bugs from large systems. In *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 267–278. doi:10.1109/ICSME.2016.67

[114] Zhenchang Xing and Eleni Stroulia. 2005. UMLDiff: an algorithm for object-oriented design differencing. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering* (Long Beach, CA, USA) *(ASE '05)*. Association for Computing Machinery, New York, NY, USA, 54–65. doi:10.1145/1101908.1101919

[115] Zhenchang Xing and Eleni Stroulia. 2007. API-evolution support with diff-catchup. *IEEE Transactions on Software Engineering* 33, 12 (2007), 818–836. doi:10.1109/TSE.2007.70747

[116] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I. Jordan. 2009. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 117–132. doi:10.1145/1629575.1629587

[117] Chunhua Yang and E. James Whitehead. 2019. Pruning the AST with hunks to speed up tree differencing. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. 15–25. doi:10.1109/SANER.2019.8668032

[118] Deheng Yang, Xiaoguang Mao, Liqian Chen, Xuezheng Xu, Yan Lei, David Lo, and Jiayu He. 2023. TransplantFix: graph differencing-based code transplantation for automated program repair. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering* (Rochester, MI, USA) *(ASE '22)*. Association for Computing Machinery, New York, NY, USA, Article 107, 13 pages. doi:10.1145/3551349.3556893

[119] Cristian Zamfir, Baris Kasikci, Johannes Kinder, Edouard Bugnion, and George Candea. 2013. Automated debugging for arbitrarily long executions. In *14th Workshop on Hot Topics in Operating Systems (HotOS XIV)*. USENIX Association, Santa Ana Pueblo, NM. https://www.usenix.org/conference/hotos13/session/zamfir

[120] A. Zeller and R. Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200. doi:10.1109/32.988498

[121] Andreas Zeller and Dorothea Lütkehaus. 1996. DDD—a free graphical front-end for UNIX debuggers. *SIGPLAN Not.* 31, 1 (jan 1996), 22–27. doi:10.1145/249094.249108

Received 1 November 2025; revised 7 April 2026; accepted 22 May 2026

Just Accepted