Understanding the Linux Kernel, Visually

Hanzhi Liu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
hanzhiliu@smail.nju.edu.cn

Yanyan Jiang
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
jyy@nju.edu.cn

Chang Xu
State Key Laboratory for Novel Software Technology, Nanjing University
Nanjing, China
changxu@nju.edu.cn

Abstract

Understanding the Linux kernel is challenging due to its large and complex program state. While existing kernel debugging tools provide full access to kernel states at arbitrary levels of detail, developers often spend a significant amount of time sifting through redundant information to find what is truly useful. Additionally, the textual results provided by traditional debuggers are often insufficient for expressing high-dimensional information in a readable manner.

This paper presents Visualinux, the first debugging framework that can simplify the program state of the Linux kernel to a level that can be visually understood with low programming complexity and effort. Visualinux includes a domain-specific language for specifying simplifications of a kernel object graph, an SQL-like domain-specific language for customizing the simplified object graph, and a panel-based interactive debugger. Evaluation results show that Visualinux can visualize various complex kernel components and efficiently assist developers in diagnosing sophisticated kernel bugs.

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

Keywords: Linux Kernel, Software Visualization, Debugging

ACM Reference Format:
Hanzhi Liu, Yanyan Jiang, and Chang Xu. 2025. Understanding the Linux Kernel, Visually. In *Twentieth European Conference on Computer Systems (EuroSys '25), March 30–April 3, 2025, Rotterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3689031.3696095

EuroSys '25, March 30–April 3, 2025, Rotterdam, Netherlands

ACM ISBN 979-8-4007-1196-1/25/03
https://doi.org/10.1145/3689031.3696095

1 Introduction

For years, developers have used GDB [11, 14, 28] and log/trace tools [19, 22, 23] to debug the Linux kernel. These tools provide mechanisms to examine kernel states at various levels of detail. However, considering that the human brain can only handle a limited amount of information at a time, developers are often overwhelmed by the abundance of information. They must take additional steps to *simplify* the information and isolate the truly useful data.

Unfortunately, the low-level debugging mechanisms (or even modern debuggers like drgn [2], which allows dynamic introspection of the running kernel with Python scripts) require significant human or programming effort to overcome the complexity of the kernel state and extract a specific subset of the information for a particular debugging objective, such as diagnosing the root cause of a failure or understanding the implementation of a data structure. Even worse, much of this effort is wasted once a debugging session is terminated.

Motivating example. The red-black tree for virtual memory area management was finally retired in the Linux kernel 6.1. Its successor, the maple tree [5], is a read-lock-free range-based B-tree that scales better on multi-processor systems.

We struggled to understand this new data structure: a textual interface naturally falls short of displaying high-dimensional information (e.g., an interconnected, complex data structure) in a readable way. We wrote scripts to unwrap the `union` type of nodes and parse the compressed pointers. Nevertheless, comprehending the list of tree nodes with indirect pointers remained mentally challenging.

The obstacle to understanding the maple tree is rooted in the complexity of the Linux kernel: There are millions of live objects at runtime, with complex objects having hundreds of fields (e.g., `task_struct`, which represents a process). Additionally, objects are often referenced through containers. Simply "printing" these objects as text produces an overwhelming amount of information for debugging purposes. Such complexity leads to the following question:

> *What kind of mechanism can help developers effectively and efficiently customize a view of the kernel object graph for the purpose of understanding kernel states?*

Approach. We respond to this question by observing that the kernel state is essentially a graph, where kernel objects

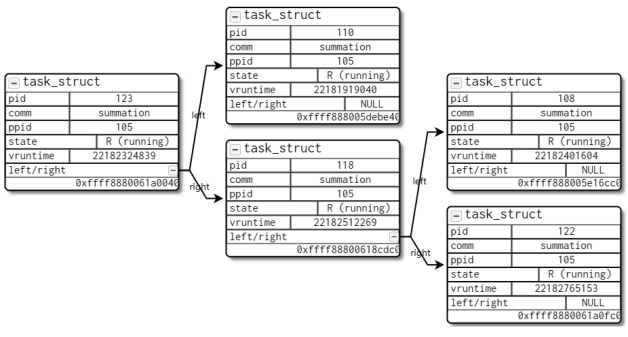
are pointer-connected nodes. A human debuggers' (implicit) goal is to *simplify* this graph to meet their debugging objectives. This paper argues that such simplified views can be created through two layers of simplification:

1. **ViewCL**, the View *Construction* Language, which allows for the creation of object graphs (plots) at customizable levels of abstraction. ViewCL employs three fundamental operators—*prune*, *flatten*, and *distill*—to programmatically reduce the kernel states.
2. **ViewQL**, the View *Query* Language, which enables customization of a simplified object graph in a developer-friendly manner. ViewQL provides the ability to exclude specific irrelevant object types or fields, eventually producing a human-readable plot.

The following ViewCL program defines a plot of the red-black tree managing the Completely Fair Scheduler (CFS) [4] run queue on the first processor:

```
1  // Declare a Box for a task_struct object
2  define Task as Box<task_struct> [
3      Text pid, comm
4      Text ppid: parent.pid
5      Text<string> state: ${task_state(@this)}
6                          // ${...} evaluates a C expression
7                          // @this refers to the Box itself
8                          // <...> denotes the display format
9      Text se.vruntime
10 ]
11
12 // cpu_rq(0) is the run queue of the first processor
13 root = ${cpu_rq(0)->cfs.tasks_timeline}
14
15 // RBTree is our predefined container data structure
16 // @root refers to the definition in Line 13
17 sched_tree = RBTree(@root).forEach |node| {
18     // For each node yield (create) a box of Task whose
19     // associated object is @node.se.run_node
20     yield Task<task_struct.se.run_node>(@node)
21 }
22
23 // Plot the object graph rooted at @sched_tree
24 plot @sched_tree
```

In ViewCL, a `Box` encodes the fields of interest to the developer (Lines 1–10), and `${...}` evaluates a C expression, which can be used for declaring fields (Line 5) or retrieving data from the kernel state (Line 13). We also implement container data structures like `RBTree` as part of the "standard library." When provided with a closure in `forEach`, it collects the generated boxes in the loop and constructs an object graph (Lines 15–21). At a GDB breakpoint, this program produces a simplified plot of the run queue:



For systems under heavy workloads, the plot might still be too large to read. To simplify it further, one can use the following ViewQL program, which operates on a generated plot and toggles the display-related attributes of the boxes:

```
1  // Select all tasks
2  task_all = SELECT task_struct FROM *
3
4  // Select tasks that has pid or ppid of 2
5  task_2 = SELECT task_struct
6      FROM all_tasks
7      WHERE pid == 2 OR ppid == 2
8
9  // Mark tasks that are not process #2 or its direct
10 // children with attribute "collapsed: true"
11 UPDATE task_all \ task_2 WITH collapsed: true
```

ViewQL follows most of the SQL syntax, which `SELECT`s a set of objects and `UPDATE`s their key-value attributes. The example above focuses on process #2 and its direct children in the run queue by updating all other tasks in the run queue as "collapsed".

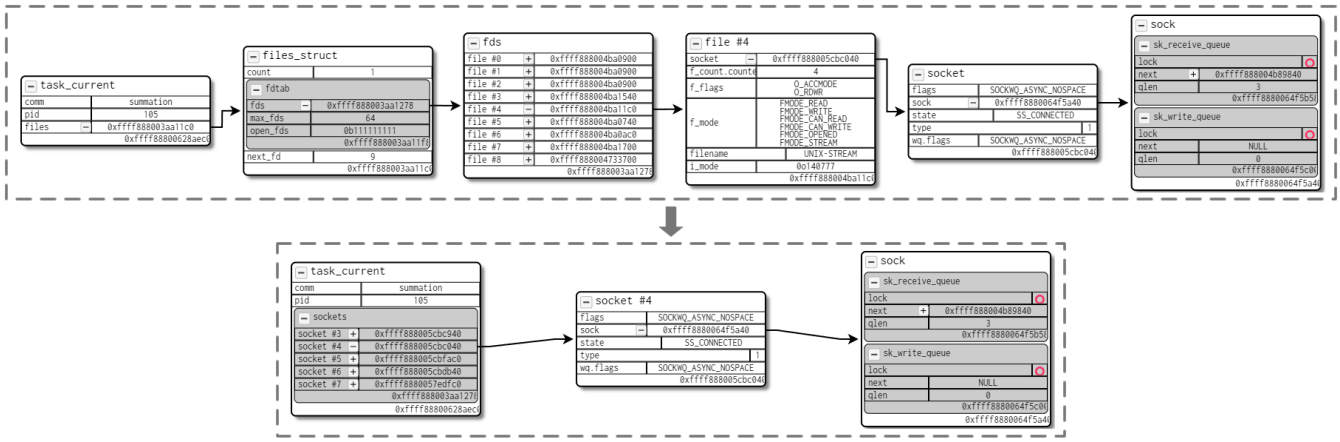
The separation of ViewCL and ViewQL, rather than using a single language, explicitly divides the tasks of extracting the kernel state for visualization (ViewCL) and creating a visually tractable plot (ViewQL). With commonly used Linux kernel data structures and containers being predefined in ViewCL, most developers can work exclusively with ViewQL, often without even being aware of ViewCL's existence. The simplicity of ViewQL also enables large language models to customize plots based on user's textual descriptions, without requiring knowledge of either ViewCL or ViewQL.

Contributions. This paper presents a novel attempt to bridge the gap between a complex system's runtime state and the limited attention span of the human brain. We proposed the first programmatic approach that enables program state visualization of the Linux kernel. Specifically, we designed two domain-specific languages (DSLs), ViewCL and ViewQL, which work together to break down the large, complex kernel state into simpler state "views" that are visually tractable.

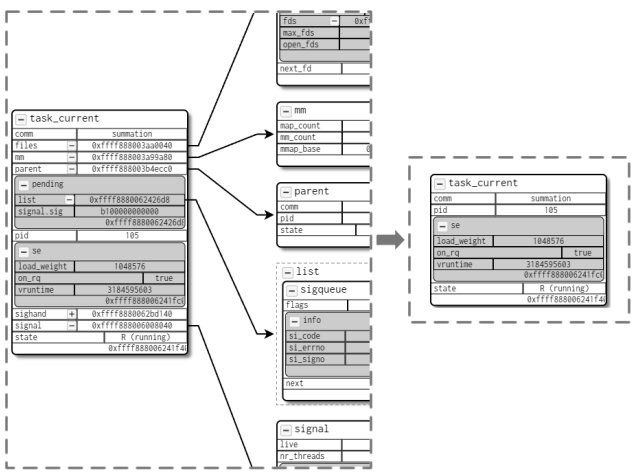
We implement these ideas in a tool, Visualinux, which functions as a detached front-end for GDB [14]. Visualinux offers developers a user-friendly, push-button interface to manage multiple views. It is capable of porting representative figures from the well-known (though now obsolete) textbook *Understanding the Linux Kernel* [35] to the latest Linux 6, and we demonstrate its effectiveness through case studies involving the diagnosis of real-world Linux kernel CVEs [67, 68]. Visualinux is publicly available at:

https://icsnju.github.io/visualinux

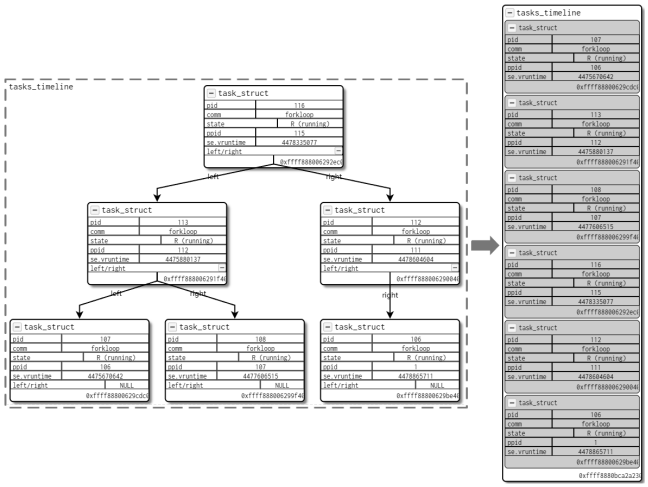
Paper organization. The rest of this paper is organized as follows. Section 2 presents the design of Visualinux DSLs and the interactive debugger, followed by concrete examples in Section 3. Implementation details and evaluation are discussed in Section 4 and Section 5, respectively. Sections 6 and 7 summarize the related work and conclude the paper.



(a) Flatten



(b) Prune



(c) Distill

Figure 1. Illustration of the mechanisms for simplifying the kernel state.

2 Unraveling the Kernel State

Visualinux is built upon Daniel Jackson's *small scope hypothesis* [51], which suggests that a *small portion* (sufficiently small to be visually processed by the human brain) of the state suffices for understanding and debugging a specific part of the system. Therefore, our first step towards practical kernel debugging is an efficient mechanism to simplify the full kernel state for a specific debugging objective.

2.1 Problem Analysis

The kernel state is essentially a graph, where kernel objects are nodes and nodes are connected by pointers. We observe that developers understand (simplify) a state in terms of a debugging objective by repeatedly applying the following three operations to simplify a running state:

1. **Prune.** To produce small states, developers prune (remove) fields, objects, and relations that are irrelevant to the debugging objective. Figure 1b 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 be long while intermediate objects are irrelevant to the debugging objective. Developers *flatten* (or compress) a path as a direct link between objects. For example, Figure 1a flattens the long path from a `task_struct` to its associated sockets, skipping intermediate objects in three different kernel modules.
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 not specifically involving the tree structure, a flattened linear list of intervals may read better. Therefore, developers intentionally *distill* a kernel data structure into a compact format that retains objects' logical relations. Figure 1c provides such an example.

Given a root node (e.g., a pointer to a `task_struct`) as the starting point of visualization, we observe that applying these operations yields a visually understandable view of the kernel state. For years, 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] are designed to enhance the versatility and efficiency of this procedure. Visualinux follows this approach.

2.2 Plotting an Object Graph

This paper encodes the mechanism of *prune*, *flatten*, and *distill* as a domain-specific language, ViewCL, to formally express the kernel state simplification procedure under a specific debugging objective. The (simplified) core syntax of ViewCL is listed below, with further implementation details discussed in Section 4:

Program	P	::=	b^*
BoxDecl	b	::=	Box *box* $\{ v^+ \}$
View	v	::=	*view* $\{ i^* \}$
Item	i	::=	`Text`(*expr*)
		|	*box*(*expr*)
		|	`Link`(*box*(*expr*))

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 to plot an object. The concept of `View` is inspired by database system views [82]. 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. Every `Link` is always a member of a `Box` pointing to another `Box`. A pointer can naturally correspond to a `Link`.

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

```
define Task as Box<task_struct> {
    // Default view: only pid and comm (command)
    :default [
        Text pid, comm
    ]

    // Scheduler view: extends default with vruntime
    :default => :sched [
        Text se.vruntime
    ]

    // Scheduler-rq view: displays the task's run queue
    :sched => :sched_rq [
        // A link named "runqueue" to @rq defined below
        Link runqueue -> @rq
    ] where {
        rq = ... // 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 the `pid`, `comm`, and `se.vruntime` items.

Given a `Box` type and an expression indicating the root object, Visualinux will traverse the runtime object graph through the links (i.e., pointer relations) recursively until all reachable objects are processed. A corresponding box will be created for every object reached. We also provide ViewCL programs (with built-in views) for both generic data structures (linked lists, red-black trees, etc.) and frequently used objects (`task_struct`, run queue, files, etc.), such that customized simplification can usually be achieved within modifying a few lines of ViewCL code.

The *prune*, *flatten*, and *distill* operations in Section 2.1 are implemented as follows in ViewCL:

1. The definitions of `Box` and `View` encode the *prune* operation. Observing that *most* of the state data is irrelevant to a specific debugging objective, ViewCL specifies what to plot, i.e., the complement of objects to be pruned.
2. ViewCL introduces dot-connected fields to *flatten* a data path. For example, we can use `Text(parent.pid)` or `Link(files.fdtab.fds)` to bypass the intermediate links between objects.
3. ViewCL implements *distill* by providing converter functions. These functions take a container object, 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.

2.3 Customizing a Simplified Object Graph

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`). We can further refine and customize the visualization of G using 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 under the filtering condition from `WHERE`.
- `UPDATE` assigns attributes to the set of selected boxes. Attributes determine how each box is displayed, e.g., selecting a view (by setting the *view* attribute) or making a box invisible (by setting *trimmed* to `true`).

In contrast to ViewCL's focus on constructing an object graph, ViewQL is designed to allow developers to easily customize the object graph without needing to understand

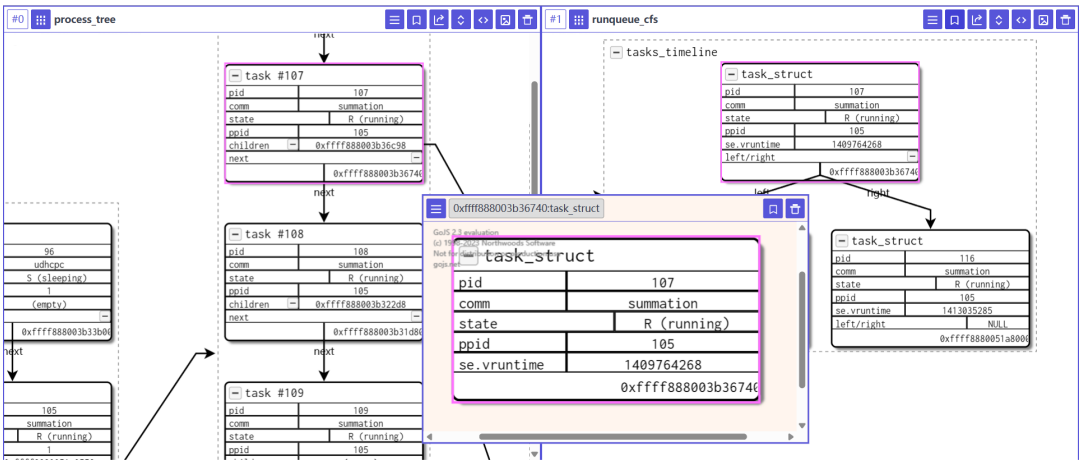


Figure 2. The front-end of Visualinux debugger with two primary panes and a secondary pane. Through the "focus" operation, the developer can easily find a specified `task_struct` in two process management structures for different purposes.

ViewCL. ViewQL offers flexible last-mile customization for developers' own debugging objectives.

Suppose that a ViewCL program implements three views: `default`, `show_mm`, and `full` for `task_struct`. One can configure the process tree to display memory mapping for all user threads (tasks associated with address space) by:

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

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

One can also further configure the memory mapping to display only the writable memory areas:

```
// Select memory areas that do not have a writable flag
// is_writable is a ViewCL-defined field
non_writable_vmas = SELECT vm_area_struct
    FROM *
    WHERE is_writable != true

// Set these VMAs' "collapsed" attribute, for displaying
// only a small click-to-expand button
UPDATE non_writable_vmas WITH collapsed: true
```

2.4 A Visual Interactive Debugger

Developers may need to simultaneously inspect different parts of a kernel state, or even different perspectives of a single data structure (e.g., a process tree as an ordered list, or simultaneously as a raw data structure in the run queue). The visual interactive debugger provides a paneling mechanism for creating linked or detached views over two types of panes (each pane displays an object graph)[1]:

1. *Primary pane*, which displays a ViewCL-extracted object graph, where ViewQL programs can be applied to further customize.
2. *Secondary pane*, which displays a focused object picked by the developer from another primary or secondary pane.

Starting from a single primary pane displaying an object graph, panes can be managed as a pane tree using the following operations:

1. *Split* (vertically or horizontally) an existing primary pane to create a new primary pane.
2. *Select* (by clicking objects or using ViewQL) a set of objects from a pane to create a new secondary pane for displaying them.
3. *Refine* a set of objects displayed in a pane by applying a ViewQL program.

Our debugger also allows developers to simultaneously explore multiple plots displayed in different panes. For instance, a "focus" operation searches for a specified object on all displayed object graphs. This is useful when the developer obtains an object from one data structure and wants to figure out how it is simultaneously managed by other data structures. Figure 2 shows an example, where the developer searches for a process in a parent tree (left) and a scheduling tree (right).

Finally, the simplicity of ViewQL (only `SELECT` and `UPDATE` without nested queries) enables large language models to synthesize ViewQL programs from natural-language descriptions, and developers can directly "talk" to the debugger to obtain a designated plot. Taking the previous example of displaying `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, plus our predefined ViewQL examples for in-context learning, yields the following correctly synthesized ViewQL program for view customization:

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

[1]We borrowed this idea from tmux [26], a widely used terminal multiplexer that provides pane-based window management.

3 Debugging with Visualinux

To diagnose sophisticated bugs, developers typically debug interactively in an iterative "guess-and-check" fashion [29, 59]. Given a reproducible failure, a developer formulates hypotheses about the root cause based on their experience, observes the program state to test these hypotheses, and uses the results to narrow down the problem scope. This iterative process continues until the root cause is identified.

Visualinux enhances this procedure by extending GDB with a set of CLI-like commands that can be invoked on breakpoints. Using these commands, developers can evaluate ViewCL programs over the current program state to generate object graphs, which are sent to the visualizer. They can also flexibly manipulate the graphs via ViewQL programs (or even using natural language with the help of LLM), such as switching views for a group of objects that have multiple views, and collapsing a group of visually distracting objects.

3.1 Example (1): Live Visualization of an Under-documented Data Structure

The Linux kernel 6.1 introduces 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 is a range-based B-tree for tracking gaps and storing ranges using read-copy-update (RCU) [24]. It is one of the most complex generic data structures in today's Linux kernel.

Unfortunately, like many newly developed components, thorough documentation and tutorials for the maple tree are lacking. Developers may find it challenging to understand its design and implementation details. Existing documentation [5, 21, 50] provides introductory concepts, design principles, and APIs but lacks details of the data structure, such as how it maintains node relations, performs data compaction to achieve high efficiency, or leverages the RCU mechanism to support lock-free updates.

At the time of writing, the authors are not aware of any existing diagrammatic representation of this novel 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 is straightforward declarations of object fields, and most of the GDB Python code is Python implementation of helper functions for parsing the data structure.

The simplified ViewCL program is shown in Figure 3. The `where` clause (e.g., Lines 19–62) of a box creates a local scope, allowing variables to be defined and referenced in the box items. For instance, `@slots` is defined on Line 34 and referenced as an embedded object field on Line 17.

A maple node (Line 13) is a complex `union` that consists of multiple levels of indirection, containing either 10 or 16 "slots" depending on the node type. We use a switch-case

```
1  // ${...} evaluates a C expression
2  // @{...} refers to a ViewCL object
3  // @this denotes the box itself
4
5  define VMArea as Box<vm_area_struct> [
6    // Display 64-bit numbers as hexadecimal
7    Text<u64:x> vm_start, vm_end
8    // ...
9    Link vm_file -> @...
10   Link anon_vma -> @...
11 ] where { ... }
12
13 define MapleNode as Box<maple_node> [
14   Text<u64:x> @last_ma_min, @last_ma_max
15
16   // ViewCL container objects
17   Container slots: @slots
18   Container pivots: @pivots
19 ] where {
20   // Variables defined in the where clause can be
21   // referenced in the parental scope
22
23   is_leaf = ${mte_is_leaf(@this)}
24   node = ${mte_to_node(@this)}
25   pivots = ...
26
27   // These values are used in MapleNode items; ViewCL
28   // collects the ma_min and ma_max in the last loop
29   // interation (Lines 42-43)
30   last_ma_min = @ma_min
31   last_ma_max = @ma_max
32
33   // Plot the maple node slots according to its node type
34   slots = switch ${mte_node_type(@this)} {
35     case ${maple_leaf_64}, ${maple_range_64}:
36       // Plot the slots as an array
37       Array(@node.mr64.slot).forEach |item| {
38         // Generate a box for each item in @node.mr64.slot
39         yield Box [
40           Link slot -> @slot_to_plot
41         ] where {
42           ma_min = ${ma_calc_min(...)}
43           ma_max = ${ma_calc_max(...)}
44
45           slot = switch ${ma_slot_check(...)} {
46             // A live maple tree slot
47             case ${true}:
48               switch @is_leaf {
49                 // Leaf node: this is a VMA
50                 case ${true}: VMArea(@item)
51                 // Non-leaves: recursive construction
52                 case ${false}: MapleNode(@item)
53               }
54             // Unavailable slot, e.g., pending-free
55             case ${false}: NULL
56           }
57         }
58       }
59     case ${maple_arange_64}: ...
60     otherwise: ...
61   }
62 }
63
64 define MapleTree as Box<maple_tree> [
65   Link ma_root -> @ma_root
66 ] where {
67   type = ${mte_node_type(@this.ma_root)}
68   ma_min = ${0}
69   ma_max = ${mt_node_max(@type)}
70   ma_root = switch ${xa_is_node(@this.ma_root)} {
71     case ${true}: MapleNode(@this.ma_root)
72     case ${false}: VMArea(@this.ma_root)
73   }
74 }
```

Figure 3. ViewCL program for plotting maple trees.

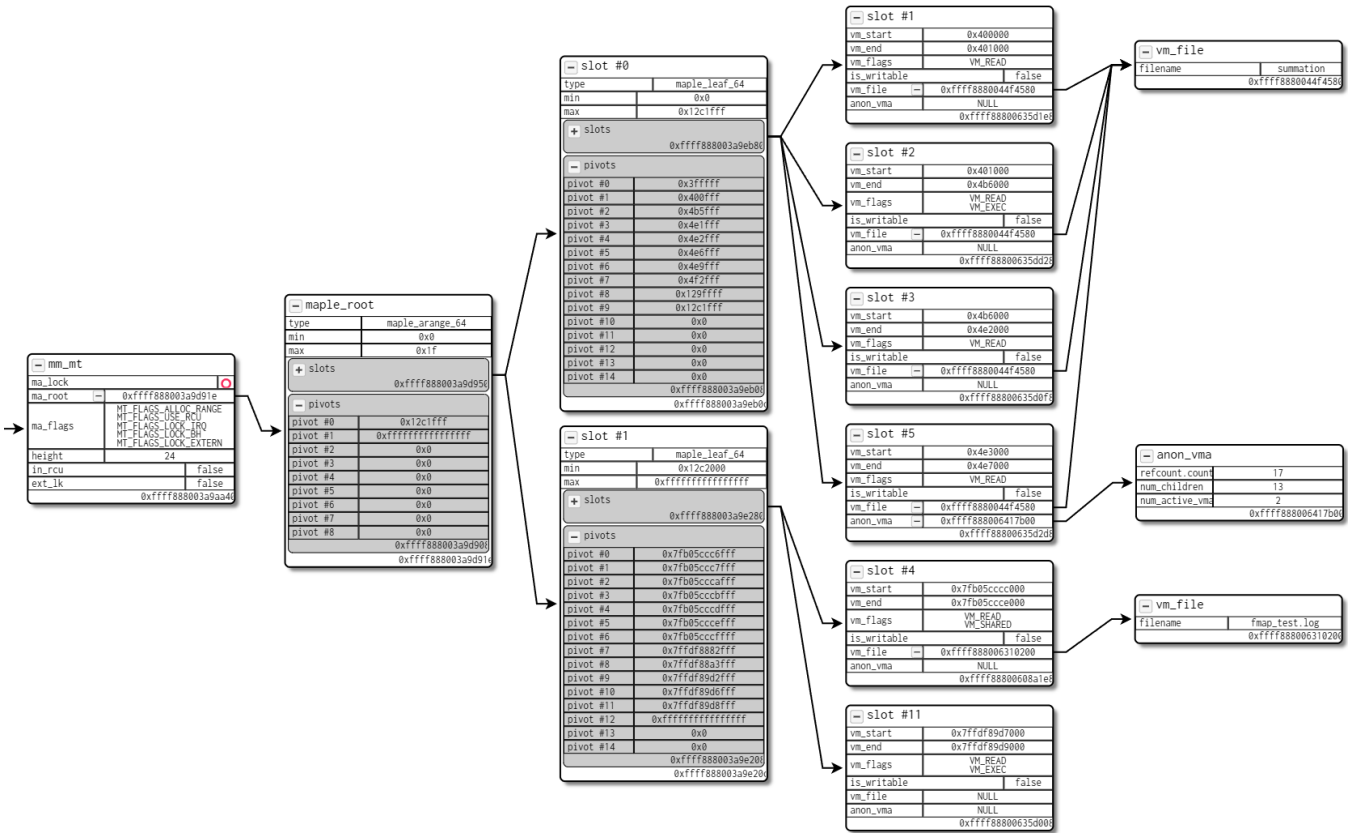


Figure 4. Visualizing a maple tree representation of the memory areas of a process.

statement to unwrap the enumeration type, which enables the display of different types of nodes with distinct boxes (Lines 34–35, 59–60) and conditional object generation (Lines 45–56). Variables in the `where` clause can be accessed in the parental scope. The items defined in Line 14 refer to Lines 30–31, which, in turn, reference Lines 42–43.

One can also use ViewQL to further simplify and customize the plot. The following ViewQL program collapses the large but useless lists of slot pointers and trims all writable memory areas to help focus on the read-only ones (assuming that this is the debugging objective), yielding the plot in Figure 4, which is readable as a data structure describing the address space of a process.

```
// 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 is_writable == true
UPDATE writable_vmas WITH trimmed: true
```

```
1  // CPU #0                          // CPU #1
2  mm_read_lock(&mm->mmap_lock)       mm_read_lock(&mm->mmap_lock)
3  expand_stack()                     find_vma_prev()
4    mas_store_prealloc()               mas_walk()
5      ...                                ...
6        mas_free()                         node = rcu_deref_check()
7          ma_free_rcu()                    // node pointer fetched
8            call_rcu(&mt_free)
9            // node is dead
10 mm_read_unlock(&mm->mmaplock)
11 ... // wait for RCU period
12 rcu_do_batch()
13   mt_free_rcu()                    mas_prev()
14     kmem_cache_free()                ...
15     // node is freed                   rcu_deref_check(node..)
16                                        // UAF occurs
17                                    mm_read_unlock(&mm->mmaplock)
```

Figure 5. A simplified kernel function call trace that can trigger the use-after-free bug of CVE-2023-3269. CPU #0 holds a read lock (Line 2) and attempts to free a node (Lines 6–8). The actual free operation is deferred by RCU (Lines 12–14) until after a grace period. However, this leads to a use-after-free error on Line 15 when CPU #1 is concurrently reading the same data structure.

3.2 Example (2): Understanding a Security Breach

Figure 5 presents a simplified kernel trace of CVE-2023-3269 [68], a high-severity vulnerability caused by concurrent use-after-free (UAF). The root cause of the vulnerability is a CPU holding `mm_read_lock` that frees an object. The actual

free is deferred until after the RCU grace period, causing a subtle concurrent UAF. It took the Linux kernel developers nearly two weeks to reach a consensus on the bug and provide a patch [8]. Manifesting this bug involves interactions between multiple kernel mechanisms: address spaces (including the newly introduced maple tree), memory management, and lock-free synchronization.

Visualinux can provide an intuitive visualization of both the maple tree and the RCU waiting list to help developers understand how corrupted states caused the bug. During interactive debugging, developers can also customize which state information of these data structures should be displayed, such as held locks, bit flags, and the reference count, based on their interest.

Suppose that a developer can consistently observe a UAF error on a maple tree node. A data structure plot (like Figure 4) would reveal that the UAF always occurs when dereferencing a leaf node. This indicates that the internal structure of maple tree may be irrelevant to the UAF. Correspondingly, the developer can slightly 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 mm_maple_tree -> @mm_mt
      ]
+     :default => :show_addrspace [
+         Link mm_addr_space -> @mm_as
+     ]
  } where {
      mm_mt = MapleTree(@this.mm_mt)
+     mm_as = Array.selectFrom(@mm_mt, VMArea)
  }
```

A `MapleTree` maintains an ordered set. `Array.selectFrom` traverses such a set and produces a sorted list of the objects in the set. 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 list of memory-mapped regions.

The developer can then inspect the state transition in detail by pausing the CPU right before the memory dereference point and trying to capture another CPU's `free` of this memory. With Visualinux, the developer can "pin" a specific maple tree node using a natural-language instruction:

> *Find me all vm_area_struct whose address is not* `<fetched_node_address>`*, and collapse them.*

Large language models (specifically, DeepSeek-V2 [41] in this paper) can provide the following ViewQL program for further pruning the states by making unrelated memory areas invisible for visual clarity:

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

The developer can then observe this node being transferred to the RCU waiting list and later being freed after the RCU grace period, capturing the use-after-free bug. If locks are chosen for visualization, their displayed status can also clearly indicate the lock held incorrectly.

Visualinux enables efficient interactive debugging in two key aspects. First, ViewCL provides simplified representation of a huge data structure, and developers can visually map the nodes to their conceptual model of a maple tree. Second, after fetching the potentially freed node, the developer can ask Visualinux to focus on it through a few lines of ViewQL code (or even natural language). This helps the developer further simplify the state information and focus on the subset related to the debugging objective.

4 Visualinux Implementation

We implement Visualinux prototype for interactive debugging. Visualinux can debug the Linux kernel via GDB, whether for virtual machines or a remote physical machine running KGDB. It comprises two main components: the GDB extension, which is integrated into the GDB host to support ViewCL, and a visualizer front-end that implements ViewQL and the paneling mechanism. The GDB extension includes ~4,000 lines of Python code, along with ~500 lines of GDB scripts that expose kernel functions invisible to the debugger in ViewCL, such as static inline functions. The visualizer is implemented in roughly 2,000 lines of TypeScript, managing the extracted object graphs, interpreting the ViewQL, and displaying the panes.

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

1. *vplot* extracts object graphs according to a given ViewCL program. It can also synthesize naive ViewCL code for trivial debugging objectives.
2. *vctrl* controls the panes and the views displayed on them. For instance, it can split an existing pane to create a new one, or apply a ViewQL request to an object graph.
3. *vchat* relies on LLM to provide a natural language interface: it converts the given message to another command (either *vplot* or *vctrl*) with well-formed arguments.

4.1 ViewCL and Interpreter

Though ViewCL has provided a programmatic way to perform kernel state simplification, it is still non-trivial to define a specific plot of a given Linux kernel state. The Linux kernel-specific mechanisms for heterogeneous resource management make the state extraction more complicated, such as `container_of`-based data structure management and cpu-local pointer translators. The Linux kernel also utilizes various C programming tricks to simulate object-oriented or functional programming features, such as multipurpose `void` pointers, unions of complex objects, and function pointer

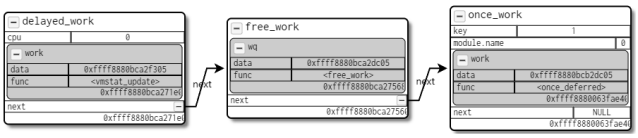


Figure 6. Visualization of a work list of workqueue `mm_percpu_wq` in the Linux kernel. Work lists are heterogeneous lists constructed through the `container_of` mechanism, and the types of their nodes are determined by a function pointer field. The next pointer is the list abstraction extracted through Visualinux, which is not the real one in kernel implementation.

Table 1. Text decorators supported by ViewCL

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

sets as operation interfaces. We additionally support the following features to improve the usability of ViewCL.

Handling data compaction. The Linux kernel widely adopts data compaction to improve data locality. For example, multiple short integers can be packed together into a single integer, and the low-order bits of an aligned pointer can be filled with bit flags. ViewCL allows the mixing of C expressions with ViewCL variables to handle compact data, e.g., bit fields in a page table entry.

Handling embedded containers. In the Linux kernel, container structures (such as linked lists and red-black trees) are typically nested within objects and organized using the `container_of` macro. Figure 6 illustrates an example of a workqueue in the Linux kernel, which maintains a sequence of asynchronous tasks scheduled and executed in the background. For embedded containers, the objects (nodes) themselves are unaware of the type information of the data objects; this information must be inferred through the access method of the linked list.

ViewCL provides predefined containers to assist in describing data structures organized by the `container_of` relation. As demonstrated in Section 1, developers can easily define an `RBTree` and customize the member objects. Given that `container_of`-based data structures are widespread in the Linux kernel, this mechanism significantly enhances the practicality of ViewCL.

Handling polymorphic objects. ViewCL supports switch-case statements to handle conditional object connections. For instance, for multipurpose pointers and unions of large objects, the exact type of object is decided at runtime. Moreover, the combination of `Container` and switch-case enables simple handling of significantly complex structures such as the heterogeneous list in Figure 6.

Text decorators. ViewCL provides decorators to specify the text format, such as the size and base of an integer. It is also possible to customize display formats, such as the function name of a function pointer, or the macro names of bit flags. We also specifically support EMOJI to intuitively visualize a stateful value, e.g. whether a spinlock is held. Table 1 shows all available text decorators of ViewCL.

Implementation. ViewCL programs are interpreted by Visualinux. Starting from a root box object specified at `plot`, Visualinux recursively evaluates all expressions and items by resolving the innermost `${...}` (evaluating a C expression by GDB, in which macros and GDB Python functions are resolved) or `@...` (referring to a ViewCL variable, e.g., another Python box object).

4.2 ViewQL and Visualizer

The visualizer is implemented as an independent front-end. It refreshes the panes and their visualized contents upon receiving an HTTP POST request from `v`-commands executed in GDB: either *vplot* with extracted object graphs, or *vctrl* with ViewQL programs or other pane operations.

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

- *view* (`string`) determines which views of boxes are displayed. If absent, the view is set to `default`.
- *trimmed* (`bool`) objects and their descendants are removed from the graph.
- *collapsed* (`bool`) objects are displayed as small buttons; clicking this button will remove the "collapsed" attribute.
- *direction* (`string`) is used for containers, indicating whether they are plotted horizontally (default) or vertically.

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 like `is_inside`.

Implementation. Visualinux serializes the box representation of objects in an in-memory database and transpiles ViewQL programs to NoSQL queries for execution. We also implement persisting the state of panes and plots for reuse across debugging sessions.

The visualizer can also synthesize a ViewQL program from the user's natural language description by invoking a large language model. Specifically, a text message `desc` in *vchat* will be pasted into the following prompt consisting of descriptions, guidance, and concrete examples:

```
A kernel object graph is ... The vertices are dented by
Box (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.
The VQL only has 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'm intended to {{desc}}. Synthesize a ViewQL program.
```

Since ViewQL inherits language features from SQL, which are well-understood by large language models, in-context learning should not be a major difficulty. DeepSeek-V2 [41] correctly synthesizes all 10 ViewQL programs in the case study in Section 5.2.

5 Evaluation

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

1. Can Visualinux handle data structures and mechanisms in the Linux kernel?
2. Can Visualinux assist in diagnosing real-world kernel bugs that require an understanding of the kernel state?
3. What is the performance overhead associated with kernel state simplification using Visualinux?

5.1 Evolving *Understanding the Linux Kernel*

The classic textbook *Understanding the Linux Kernel* [35], 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). The book covers 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, the ULK book has become outdated today: 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. Such rapid development of the Linux kernel makes no textbook can keep pace with the latest developments [32, 76, 79].

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

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 is not capable of generating a high-level overview diagram 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 within a single line of ViewCL code, as described in Section 4.1.
3. The 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 the readers.

Table 2 summarizes the results: Visualinux can indeed "revive" ULK, demonstrating the ability to handle various complex data structures and Linux kernel components. Code shared between plots is calculated repeatedly to evaluate for each independent plot. We also observe that a significant

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

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

† ULK plots these diagrams as separate figures due to space limits. We can generate both separate and merged interactive figures.

‡ 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.

Table 3. Debugging objectives for ViewQL usability evaluation. Descriptions are simplified due to space limitations.

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

proportion of core kernel mechanism implementations have changed significantly since Linux 2.6.11: 17 out of 21 (81%) figures for Linux 2.6.11 have lost their timeliness in the latest kernel, with 14 out of 17 (82%) have undergone significant implementation changes.

These discrepancies between textbooks and the actual kernel implementation hinders the new generation of developers from understanding the Linux kernel. With Visualinux, developers can gain a visual (or even interactive) representation of the system structure of the latest kernel with minimal programming effort, making it easier to intuitively understand the kernel state. All generated plots are available on our website.

5.2 Case Study (1): Beyond *Understanding the Linux Kernel*

Visualinux enables textbook writers to plot illustrative figures from real kernel states. For example, ULK does not cover the network system, 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, we demonstrate the connection between the read/write buffer queues through the process file descriptor table within a few lines of ViewCL code, which also efficiently flattens multiple irrelevant passed-through objects.

The interactive nature of ViewQL also provides a flexible way to support changing debugging objectives, which helps developers inspecting the kernel state on demand. We constructed hypothetical debugging objectives for 10 selected diagrams in ULK as listed in Table 3. All these debugging objectives involve fewer than 10 lines of ViewQL code and can be successfully generated by DeepSeek-V2 [41] from a natural-language description.

The following LLM-generated ViewQL program changes the visualization such that all superblocks are displayed vertically. It further removes all superblocks that are not attributed to any block device from the visualization, suppose that we intentionally check 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
```

Note that while Visualinux can provide plots consistent with the latest kernel implementation and a flexible mechanism to interactively understand them, it cannot replace textbooks like ULK, but is an effective supplement to them. This is because textbooks also explain the logic of the code in detail, which is beyond the scope of Visualinux.

5.3 Case Study (2): Interactive Debugging

StackRot: Maple tree and RCU waiting list. The first case is our motivating example, CVE-2023-3269 [68] (StackRot), which is a vulnerability in the memory management system that can be exploited to execute arbitrary kernel code. As demonstrated in Section 3.2, Visualinux can visualize the two complex data structures (with irrelevant information eliminated) and show the memory area being moved to the RCU waiting list, revealing the root cause of the vulnerability.

Dirty Pipe: Page cache shared between file and pipe. CVE-2022-0847 [67] (Dirty Pipe) is a local privilege escalation vulnerability caused by the `flag` field of `pipe_buffer` not being initialized within certain functions. This glitch became a flaw due to an update in the Linux kernel 5.8, which did not involve `pipe_buffer` at all.

This issue arises when the `splice` system call moves data between two file descriptors using zero-copy for the file descriptor pointing to a pipe (where the pipe data is stored in the page cache). Unfortunately, the `flags` in the pipe buffer lack proper initialization in `copy_page_to_iter_pipe()`, and a `CAN_MERGE` flag will incorrectly mark the pipe buffer as writable, causing the pipe to directly modify the shared page cache and corrupt file contents.

This vulnerability involves multiple kernel components, including process management, the file system, pipes, and the page cache. Most objects and fields in these components are irrelevant to the root cause of the bug and should be excluded from display. With the debugging objective of "a zero-copy page is being shared across file descriptors" in mind, we can implement Visualinux program to put relevant information together: pipe buffer, bug-inducing flags, and the page. With approximately 60 lines of ViewCL code, we remove irrelevant objects (including those on the critical

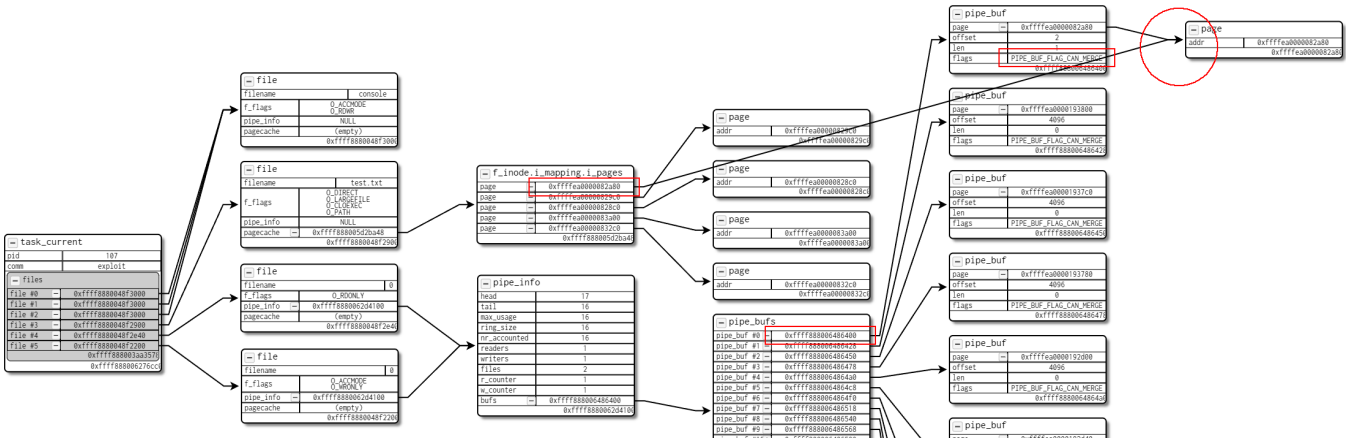


Figure 7. The Dirty Pipe-related object graph. ViewCL plots the page caches of all files and all pipes from the file table of the current thread. ViewQL trims all pages except for the shared ones. We mark the only shared page and the erroneous pipe buffer flag with red borders for clarity.

path of the reference graph), such as the file struct and inode mappings, from the debugger's view.

Using the following ViewQL program, we can further locate all page caches shared between a file and a pipe. This step of simplification is critical in narrowing down the object graph to a visually tractable size:

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

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

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

As shown in Figure 7, only one page is displayed in the rightmost column (for the buggy workload we have debugged), and we can clearly observe that the task of pid 107 owns a page from `test.txt`, which is shared with a pipe with a `CAN_MERGE` flag. Such sharing is prohibited in the kernel and thus indicates an internal logical error.

This figure evolves as the debugging process proceeds: one will observe such a page with flawed flags being added to the pipe buffer. Debugging with such dynamic plots is expected to be more effective than reading textual logs produced by debugging scripts.

5.4 Performance

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

1. *GDB (QEMU)*: Attach GDB to a localhost QEMU emulator running an x86_64 kernel (a root disk image) consisting of 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.

Table 4. Performance results of plotting the representative ULK figures.

#	Figure	GDB (QEMU)			KGDB (rpi-400)		
1	Fig 3-4	59.6	0.45	17.4	7,025.1	23.73	913.7
2	Fig 3-6	155.3	0.17	23.2	20,904.3	9.14	870.2
3	Fig 4-5	39.4	0.35	17.2	4,309.6	16.38	924.6
4	Fig 6-1	181.2	0.15	18.7	8,096.4	6.71	825.4
5	Fig 7-1	13.7	0.22	14.1	33.6	8.41	1202.3
6	Fig 8-2	60.2	0.12	16.8	3,213.8	6.63	894.2
7	Fig 8-4	12.8	0.42	13.3	894.9	29.97	945.3
8	Fig 9-2	48.6	0.29	32.0	1,402.1	8.44	930.3
9	Fig 11-1	45.6	0.31	26.7	1,392.5	9.94	852.4
10	Fig 12-3	10.9	0.13	87.5	24.7	0.34	1411.7
11	Fig 13-3	326.0	0.87	40.9	7,602.3	20.43	952.5
12	Fig 14-3	80.3	1.11	36.7	1,972.1	27.38	903.3
13	Fig 15-1	84.7	0.37	42.8	86.0	1.17	931.9
14	Fig 16-2	10.1	0.12	31.4	36.2	0.51	1101.0
15	Fig 17-1	62.4	0.31	24.7	2,322.6	11.55	921.4
16	Fig 17-6	34.3	0.33	51.6	559.0	5.53	840.6
17	Fig 19-1/2	19.0	0.52	12.5	1,078.8	41.46	861.7
18	workqueue	11.9	0.33	10.5	1,779.5	26.16	810.1
19	proc2vfs	40.2	0.31	22.1	659.3	8.13	819.0
20	socketconn	20.8	0.18	42.5	17.4	0.25	1062.5

Each | x y z | represents a visualization overhead, where x denotes the total cost in milliseconds, y indicates the cost per object in milliseconds, and z specifies the cost per kilobyte of data structure in milliseconds.

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 calling the operating system for IPCs, mapping/unmapping files and anonymous pages, etc. We visualize all figures in Table 2 and collect the runtime statistics for the most time-consuming step of ViewCL program execution[2].

Table 4 shows the performance evaluation results. The major performance bottleneck of Visualinux is evaluating the C expressions (e.g., `node.mr64.slot`) in ViewCL, and the latency is acceptable for a human debugger for most figures for both evaluated scenarios.

We also observe that KGDB on a slower device like Raspberry Pi 400 is significantly slower than GDB-QEMU on localhost: retrieving an object is approximately 50 times slower, with even retrieving a `uint64` via KGDB costing approximately 5ms. This is due to both slower processors in the embedded system and KGDB delays, which makes plotting large data structures (e.g., Fig 3-6) that frequently invoke C-expression evaluation slow. However, if we focus our debugging objective on smaller data structures, the performance of Visualinux would be considered acceptable for both platforms.

6 Related Work

Interactive debugging. Various automated debugging solutions have been proposed, including delta debugging [66, 85, 91], statistical debugging [30, 53, 61, 70, 89], causal tracing [42, 52, 63, 86], and reverse debugging [37, 45, 84, 90]. 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 [72, 88], especially those that are beyond the capabilities of the tools. This paper focuses on interactive debugging [29, 36, 55, 56]. Since full automation is still a distant goal, we believe that practical approaches to interactive debugging is indispensable for developers.

Whyline [55, 56] 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 [29] records program behaviors and user actions during bug reproduction, based on which it generates possibly-relevant hypotheses about program state and behavior, continuously narrowing down the problem scope through interaction. Although helpful for program understanding, existing research does not solve problems from the perspective of program state and offer limited assistance in diagnosing bugs involving complex data structures. Moreover, none of these approaches are applicable to complex systems like the Linux kernel.

Visualized interactive debugging. Past research has confirmed that program visualization can effectively help developers understand program state and behavior [39, 49, 62, 64, 73, 77, 78, 81]. Multiple research projects have focused on constructing visualized interactive debuggers since one of the core purposes of interactive debugging is to understand the program [15, 31, 38, 48, 57, 69, 71, 92]. Additionally, there are interactive tools designed specifically for understanding algorithms, data structures, and call graphs [1, 9, 58].

However, traditional visualization-based debugging techniques are not suitable for complex software systems with excessively large and complex 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 and deep pointer chains. Therefore, directly applying such techniques to the Linux kernel would be impractical.

Interactive debugging for kernels. Kernel debugging has been studied for many years [33, 34, 45, 52, 54, 83]. However, most of the existing research focuses on automated debugging techniques and 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 diagnose the kernel and understand the kernel from various perspectives, including interactive debugging [2, 11, 28], runtime validation [16–18, 25], logging [22], and tracing [19, 27]. Visualinux has a particular focus on visualizing a program state, but takes a different approach from scripting tools like GDB scripts [11, 28] 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 high-dimensional data structures. It offers limited assistance in understanding the kernel state, which is where Visualinux excels.

Visualinux is also not the first work that utilizes visualization to assist in understanding the Linux kernel. However, previous approaches focus on various domains such as traces [6, 27], call graphs [3] and source code organization [7, 80], 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 bridge the gap between human and software that hinders debugging. However, many existing approaches focus on designing and implementing abstractions for execution traces to solve problems from a temporal perspective [40, 46, 47, 60, 63, 65, 74], which are orthogonal to Visualinux.

Object query languages [43, 44, 75, 87] access runtime objects through relational interfaces. PiCO QL [44], for instance, defines a relational representation of accessible Linux

[2]ViewQL and front-end rendering incur negligible overhead.

kernel data structures and allows developers to customize views for system resources. This paper focuses on program understanding, not just bug diagnosis, so our solution is quite different from existing research: Visualinux combines a declarative model of the kernel state with a relational interface for object graph manipulation, intentionally decomposing the challenges to facilitate easier kernel understanding.

7 Conclusion

This paper presents Visualinux, the first debugging framework that simplifies the Linux kernel state to a level that can be visually understood with low programming complexity and effort. Built upon two domain-specific languages for kernel state simplification and view customization, Visualinux provides developers with an intuitive perspective to visually understand the Linux kernel objects. Its integration with GDB and combination with large language models also make it easy to use in real-world debugging tasks.

Acknowledgments

We are grateful to the anonymous reviewers and our shepherd, Anton Burtsev, for their constructive feedback on this paper. This work was supported in part by National Key R&D Program (Grant #2022YFB4501801) of China, National Natural Science Foundation of China (Grants #62025202, #62272218), the Leading-edge Technology Program of Jiangsu Natural Science Foundation (Grant #BK20202001), and the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. Yanyan Jiang (jyy@nju.edu.cn), the corresponding author, was supported by the Xiaomi Foundation.

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. *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://docs.kernel.org/dev-tools/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. *Perf Wiki.* https://perf.wiki.kernel.org/index.php/Main_Page
[24] 2024. *RCU Concepts - The Linux Kernel documentation.* https://docs.kernel.org/RCU/rcu.html
[25] 2024. *Runtime locking correctness validator.* https://www.kernel.org/doc/html/latest/locking/lockdep-design.html
[26] 2024. *tmux/tmux: tmux source code.* https://github.com/tmux/tmux
[27] 2024. *Traceshark.* https://github.com/cunctator/traceshark
[28] 2024. *Using KGDB, KDB and the kernel debugger internals.* https://www.kernel.org/doc/html/latest/dev-tools/kgdb.html
[29] 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. https://doi.org/10.1145/3586183.3606781
[30] 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. https://doi.org/10.1145/2451116.2451128
[31] 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.
[32] Kaiwan N Billimoria. 2021. *Linux Kernel Programming: A comprehensive guide to kernel internals, writing kernel modules, and kernel synchronization.* Packt Publishing.
[33] 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. https://doi.org/10.1145/2351676.2351686
[34] Martin J. Bligh and Mathieu Desnoyers. 2010. Linux Kernel Debugging on Google-sized clusters. https://api.semanticscholar.org/CorpusID:262330426
[35] D. Bovet and M. Cesati. 2005. *Understanding the Linux Kernel, 3rd Edition.* O'Reilly Media.
[36] 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. https://doi.org/10.1145/2501988.2502050
[37] 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

[38] 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. https://doi.org/10.1145/1328279.1328286

[39] Sharon Ellershaw and Michael Oudshoorn. 1998. Program Visualization - The State of the Art. (06 1998).

[40] 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. https://doi.org/10.1145/2043556.2043585

[41] 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

[42] 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

[43] 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. https://doi.org/10.1007/s00607-015-0452-y

[44] 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. https://doi.org/10.1145/2592798.2592802

[45] 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

[46] 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. https://doi.org/10.1145/1094811.1094841

[47] 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

[48] 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

[49] 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. https://doi.org/10.1145/3173574.3174106

[50] 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

[51] Daniel Jackson. 2012. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press.

[52] 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. https://doi.org/10.1145/3552326.3567486

[53] 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. https://doi.org/10.1145/2815400.2815412

[54] 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

[55] 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. https://doi.org/10.1145/985692.985712

[56] 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. https://doi.org/10.1145/1368088.1368130

[57] 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. https://doi.org/10.1109/ICSME55016.2022.00066

[58] 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. https://doi.org/10.1109/VLHCC.2011.6070388

[59] 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. https://doi.org/10.1109/ESEM.2013.43

[60] 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. https://doi.org/10.1145/2151024.2151044

[61] 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. https://doi.org/10.1145/1065010.1065014

[62] 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. https://doi.org/10.1145/2556288.2557409

[63] 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. https://doi.org/10.1145/2815400.2815415

[64] 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. https://doi.org/10.1109/VISSOF.2002.1019792

[65] 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. https://doi.org/10.1145/1094811.1094840

[66] 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. https://doi.org/10.1145/1134285.1134307

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

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

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

[70] 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

[71] 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.

[72] 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. https://doi.org/10.1145/2001420.2001445

[73] 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.

[74] 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

[75] 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. https://doi.org/10.1007/978-3-642-31057-7_4

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

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

[78] 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. https://doi.org/10.1145/143062.143157

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

[80] 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. https://doi.org/10.1177/1473871617743239

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

[82] 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. https://doi.org/10.1145/500080.500091

[83] 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. https://doi.org/10.1145/1376789.1376792

[84] 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. https://doi.org/10.1145/2039239.2039251

[85] 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. https://doi.org/10.1145/3468264.3468625

[86] 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

[87] 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. https://doi.org/10.1007/11785477_3

[88] 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. https://doi.org/10.1109/ICSME.2016.67

[89] 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. https://doi.org/10.1145/1629575.1629587

[90] 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

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

[92] Andreas Zeller and Dorothea Lütkehaus. 1996. DDD—a Free Graphical Front-End for UNIX Debuggers. *SIGPLAN Not.* 31, 1 (jan 1996), 22–27. https://doi.org/10.1145/249094.249108

A Artifact Appendix

A.1 Abstract

This artifact consists of:

1. The implementation of Visualinux, which comprises a GDB extension and a visualization front-end.
2. A Docker image containing scripts and ViewCL source code for reproducing the evaluation results.

A.2 Description & Requirements

A.2.1 How to access. Visualinux is publicly available at:

https://icsnju.github.io/visualinux

The paper's corresponding version is also archived at https://doi.org/10.5281/zenodo.13710732.

A.2.2 Hardware Dependencies. The machine should be capable of compiling and debugging the Linux kernel (approximately 6GB of disk space is required). KGDB experiments were conducted on a Raspberry Pi 400 PC kit.

A.2.3 Software Dependencies. A Unix-like operating system is required with the environment necessary to compile and debug the Linux kernel. This artifact further requires GDB to be compiled with Python support (this is usually the default option for today's GDB releases), and Node.js for launching the visualizer front-end. Specifically:

- GCC 11+
- GDB 12+
- QEMU 6+
- Python 3.10+
- Node.js 18+
- make
- gcc-multilib
- flex
- bc
- bison
- xz-utils
- libelf-dev
- libssl-dev

The evaluation results (data structures) in the paper correspond to Linux kernel 6.1.X. For other kernel versions, the ViewCL scripts may subject to minor changes. We conducted our experiments on Ubuntu Server 22.04 LTS.

A.3 Evaluation Workflow

A.3.1 Major Claims.

- (C1): Visualinux can handle various data structures and components in the Linux kernel, demonstrated by "reviving" the figures in the classic textbook ULK (Table 2).
- (C2): Visualinux provides a flexible way to help achieve debugging objectives that vary in practice (Table 3).
- (C3): Visualinux can assist in diagnosing real-world kernel bugs that require an understanding of kernel state, demonstrated by case studies of two well-known CVEs (Figures 4 and 7).
- (C4): Visualinux has an acceptable performance overhead (Table 4).

A.3.2 Experiments. Instructions for reproducing the evaluation results are available in the public repository. The reproduction goals and estimated time costs are listed below.

Preparation. The kernel must be compiled first, which requires 1 human-minute and ~60 compute-minutes. Booting the kernel in GDB-QEMU requires 1 human-minute and 1 compute-minute.

Textbook Revival. This experiment generates representatives figures from the textbook ULK that achieve the hypothetical debugging objectives (i.e. the corresponding ViewQL code applied). C1 and C2 together require 1 human-minute and 3 compute-minutes.

CVE Case Studies. This experiment generates plots of data structures involved in the studied CVEs, where irrelevant objects are collapsed. To reproduce just the plots, C3 requires only 1 human-minute and 1 compute-minute. However, to fully reproduce the scenarios that expose the vulnerabilities, much more effort is required to prepare the specific kernel versions.

Performance. This experiment re-evaluates the performance of generating the ULK figures. C4 (GDB/QEMU) requires 1 human-minute and 3 compute-minutes. C4 (KGDB/rpi-400) requires 1 human-minute and 10 compute-minutes (assuming that a Linux environment on rpi-400 has been deployed).