KernelRCA: Facilitating Root Cause Analysis of Memory Corruptions in Linux Kernel with Contextual Causality Chain

Kangzheng Gu
Fudan University

Yifan Zhang
Fudan University

Yuan Zhang
Fudan University

Min Yang
Fudan University

Abstract

Continuous fuzzing infrastructure has found a large number of bugs. In this case, automatic root cause analysis (RCA) has been proposed to reduce the expensive manual effort to understand the root cause of a bug. However, existing root-cause representations are designed as isolated forms. Analysts still need to manually infer the integrated bug-triggering procedure including calling context and data dependency, which is very difficult for OS kernels due to their complexity.

In this paper, we propose *contextual causality chain (CC-chain)*, a novel root-cause representation to intuitively reflect the integrated bug-triggering procedure of memory corruptions in the Linux kernel. CC-chain shows the bug-contributing instructions to explain corresponding unexpected behaviors that lead to a bug, as well as calling contexts and data dependencies among these instructions to help analysts rapidly understand how a bug happens. To automatically construct the CC-chain, we design a root cause analysis system *KernelRCA* including selective tracing, contextual information recovery, and chain-style root cause analysis. KernelRCA successfully diagnoses 54 various kinds of real-world memory corruptions in the Linux kernel and performs better than existing crash reports and KASAN reports. A user study shows that KernelRCA's reports significantly facilitate bug understanding and fixing for human analysts.

1 Introduction

Security of the operating system (OS) kernels is critical for computer systems. To discover vulnerabilities in OS kernel, kernel fuzzing [5, 25, 35, 38] is one of the most popular approaches. Industrial fuzzing infrastructure like Syzkaller [5] has reported thousands of kernel bugs automatically. However, bug-handling tasks such as bug localization and bug fixing are still time-consuming [26, 34] and usually require experienced analysts. As a result, some bugs cannot be handled in time, which may threaten the security of operating systems.

One representative method of automatic debugging is automatic root cause analysis (RCA) [13, 33, 44, 48]. Root cause analysis takes bug-related artifacts as inputs, like PoC or memory snapshot, and outputs a series of *root-cause representations* depicting the bug's root cause, which help analysts rapidly understand and fix the bug.

To facilitate bug understanding, the design of root-cause representations is critical. It should describe the representative features of the bug. Existing works have explored two types of representative features to depict the root cause. Predicate [11, 13, 30, 33, 41] is one of the important features which is a set of boolean expressions describing bug-triggering conditions. According to bug-triggering conditions, analysts can identify the path leading to the bug and infer the bug-triggering procedure, which characterizes the execution path and program interactions involved in triggering a bug. Another representative feature is the program element [16, 32, 46, 48]. Program elements are the buggy code snippets including bug-related instructions, basic blocks, source code statements, etc. Analysts can review the code around the buggy snippets to link related code snippets together to infer the bug-triggering procedure. Recent works like ARCUS [44] consider conditions and code snippets both to make up a root-cause representation combining the advantages of predicates and program elements.

However, these root-cause representations only capture isolated bug features, leaving a gap in the ability to infer the integrated bug-triggering procedure. For user-space programs, inferring the integrated bug-triggering procedure might be easy since they usually have a uniform program entry and explicit function calls. Isolated representations can be easily linked together to recover an integrated procedure. But it is quite difficult for the kernel. First, the kernel is designed in a multi-task schema. Different tasks have different entries, while they may run simultaneously and access some shared data. Second, the kernel follows an extensible design to support various subsystems, devices, and algorithms. As the kernel is typically written in non-object-oriented languages like C, extensibility is primarily achieved through indirect calls. These two characteristics of the kernel largely increase the complexity of control flow and data dependency in the kernel,

making inferring the integrated bug-triggering procedure from the isolated root-cause representations much more difficult.

To address this problem, we propose *contextual causality chain* (*CC-chain* for short), a novel root-cause representation to help analysts easily understand the integrated bug-triggering procedure in the Linux kernel. The contextual causality chain contains two components, the causality chain and the contextual information. The causality chain is a chain of representative instructions for bug-related unexpected kernel behaviors describing how they cause the bug. The contextual information demonstrates the control flow and data dependency between these instructions to depict how unexpected behaviors on the chain are interconnected. With the help of the CC-chain, analysts would get rid of guessing or inferring the dynamic bug-triggering procedure from isolated root-cause representations, since in the CC-chain the procedure has been intuitively presented.

To automatically construct the CC-chain, we design a root-cause analysis system *KernelRCA*. Due to the widespread use of kernel fuzzing in bug detection, KernelRCA primarily focuses on diagnosing fuzzer-found bugs. It takes a C-language fuzzer-found PoC and the kernel source code as inputs and outputs a CC-chain as the root cause representation. Please note that KernelRCA is not limited to fuzzer-found bugs, as long as a bug-triggering PoC is available.

The overall idea is to trace the bug-triggering PoC and diagnose the root cause by analyzing the traces. There are three technical challenges. First, tracing the whole kernel execution is unaffordable [47]. But we have to collect sufficient information for root cause analysis. Thus, we leverage a selective tracing strategy to focus on the kernel tasks directly or indirectly triggered by the PoC, which avoids tracing the irrelevant kernel tasks. We further introduce a trace normalization process to reduce the trace size and provide critical semantics for root cause analysis. Second, we need to design an efficient approach to recover the contextual information from the raw trace. Some straightforward approaches like recording a full call stack will introduce high overhead. And combining call stacks of different instructions requires extra effort. In KernelRCA, we adopt a linear emulation-like approach to efficiently recover the calling context and data dependencies, by incrementally analyzing instruction semantics rather than performing full execution emulation. Third, we need to recognize the bug-contributing instructions from the large number of executed instructions. A simple way is to report all the instructions having control and data relationships with the crash site. But it may introduce many false positives. Instead, we leverage the contextual information to model the typical causes so that we can precisely identify the bug-contributing instructions. Finally, we report a CC-chain including a chain of bug-contributing instructions with descriptions as well as contextual information including the calling contexts and data dependencies.

The evaluation shows that KernelRCA effectively diagnoses 54 out of 65 real-world memory corruptions in the Linux kernel. We employ a distance metric to show that results of KernelRCA are closer to the root cause than kernel crash/KASAN reports. The ablation study shows the selective tracing reduces about 95% of the instructions unnecessary for analysis, and contextual information helps to precisely identify about 1% bug-contributing instructions among all the instructions having the data-dependency relationship with the crash site. Then, a user study is conducted to demonstrate the helpfulness of the CC-chain for human analysts in understanding and fixing bugs. Additionally, we show several case studies to explain why the CC-chain could help understand and fix a bug. Our contributions can be summarized as:

- We design a novel root-cause representation *contextual causality chain* to intuitively depict the integrated bug-triggering procedure.
- We implement a practical tool *KernelRCA* to automatically generate the *contextual causality chain* for memory corruptions in Linux kernel, which is open-sourced at `https://github.com/seclab-fudan/KernelRCA`.
- Evaluation shows that the *KernelRCA* can effectively generate *contextual causality chain* which facilitates understanding kernel memory corruptions.

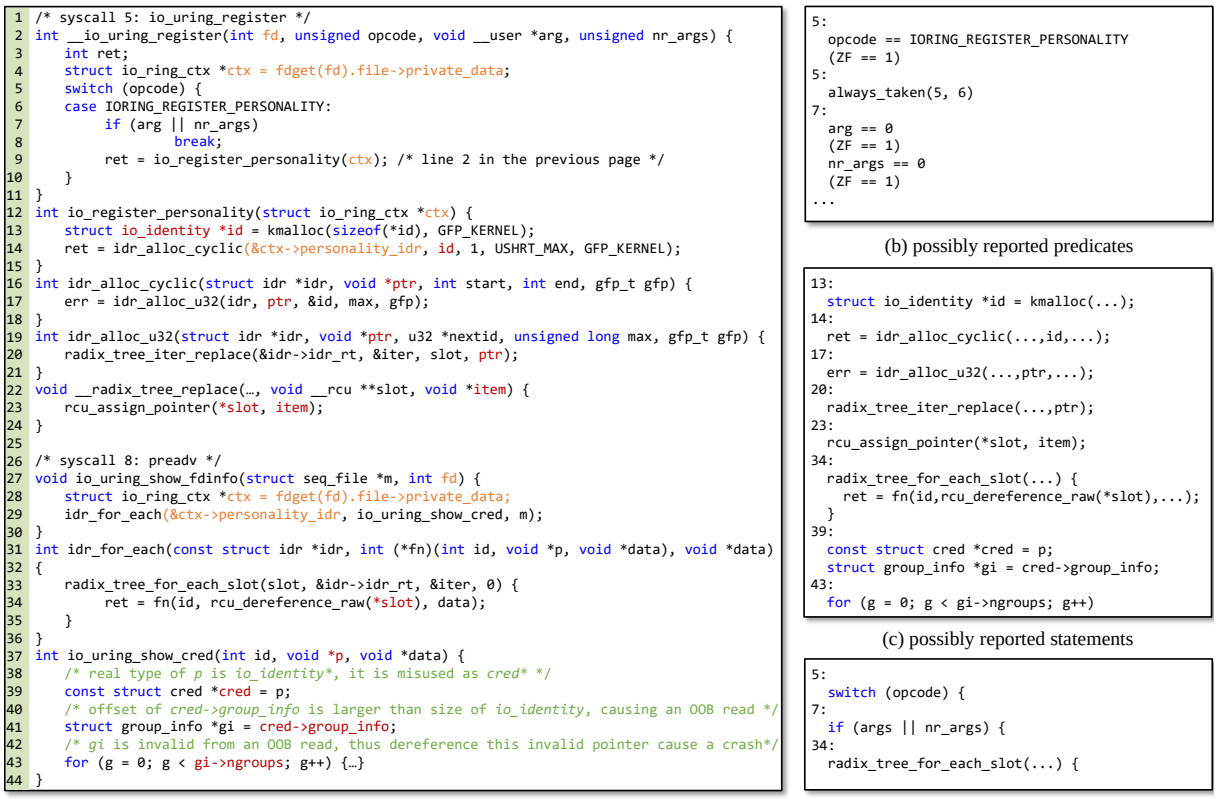
2 Problem Understanding

2.1 Motivating Example

We take a type-confusion bug that causes memory corruption as an example to discuss the advantages of CC-chain compared to existing root-cause representations. The relevant code of this bug is shown in Figure 1(a) involving two syscalls. In the PoC, the fifth syscall `sys_io_uring_register` allocates an object of type `io_identity` and saves it into the pointer `id` at line 13. Then, pointer `id` is saved into a radix tree via lines 14, 17, 20, and 23. The eighth syscall `sys_preadv` gets this radix tree at line 29, reads the `id` from the radix tree, and passes it as an argument to the function `io_uring_show_cred` at line 34. The function `io_uring_show_cred` misuses the type of the pointer `p` (i.e. `id`) as `cred *` at line 39 and fetches another pointer `gi` from `p` at line 41. As the pointer `p` is used as `cred *` but is actually `io_identity *`, the `gi` read from `p` is illegal. Finally, at line 43. dereference of illegal pointer `gi` causes a memory corruption.

2.2 Classic Root-Cause Representations

Automatic root cause analysis helps analysts understand the bug by root-cause representations. A good root-cause representation should be intuitive enough to help analysts rapidly understand what happens during the bug-triggering procedure. Analysts would first review the root cause representation and



```
1  /* syscall 5: io_uring_register */
2  int __io_uring_register(int fd, unsigned opcode, void __user *arg, unsigned nr_args) {
3      int ret;
4      struct io_ring_ctx *ctx = fdget(fd).file->private_data;
5      switch (opcode) {
6      case IORING_REGISTER_PERSONALITY:
7          if (arg || nr_args)
8                  break;
9          ret = io_register_personality(ctx); /* line 2 in the previous page */
10     }
11 }
12 int io_register_personality(struct io_ring_ctx *ctx) {
13     struct io_identity *id = kmalloc(sizeof(*id), GFP_KERNEL);
14     ret = idr_alloc_cyclic(&ctx->personality_idr, id, 1, USHRT_MAX, GFP_KERNEL);
15 }
16 int idr_alloc_cyclic(struct idr *idr, void *ptr, int start, int end, gfp_t gfp) {
17     err = idr_alloc_u32(idr, ptr, &id, max, gfp);
18 }
19 int idr_alloc_u32(struct idr *idr, void *ptr, u32 *nextid, unsigned long max, gfp_t gfp) {
20     radix_tree_iter_replace(&idr->idr_rt, &iter, slot, ptr);
21 }
22 void __radix_tree_replace(…, void __rcu **slot, void *item) {
23     rcu_assign_pointer(*slot, item);
24 }
25
26 /* syscall 8: preadv */
27 void io_uring_show_fdinfo(struct seq_file *m, int fd) {
28     struct io_ring_ctx *ctx = fdget(fd).file->private_data;
29     idr_for_each(&ctx->personality_idr, io_uring_show_cred, m);
30 }
31 int idr_for_each(const struct idr *idr, int (*fn)(int id, void *p, void *data), void *data)
32 {
33     radix_tree_for_each_slot(slot, &idr->idr_rt, &iter, 0) {
34         ret = fn(id, rcu_dereference_raw(*slot), data);
35     }
36 }
37 int io_uring_show_cred(int id, void *p, void *data) {
38     /* real type of p is io_identity*, it is misused as cred* */
39     const struct cred *cred = p;
40     /* offset of cred->group_info is larger than size of io_identity, causing an OOB read */
41     struct group_info *gi = cred->group_info;
42     /* gi is invalid from an OOB read, thus dereference this invalid pointer cause a crash*/
43     for (g = 0; g < gi->ngroups; g++) {…}
44 }
```

(a) simplified source code of syzbot#5ba5

```
5:
  opcode == IORING_REGISTER_PERSONALITY
  (ZF == 1)
5:
  always_taken(5, 6)
7:
  arg == 0
  (ZF == 1)
  nr_args == 0
  (ZF == 1)
...
```

(b) possibly reported predicates

```
13:
  struct io_identity *id = kmalloc(...);
14:
  ret = idr_alloc_cyclic(...,id,...);
17:
  err = idr_alloc_u32(...,ptr,...);
20:
  radix_tree_iter_replace(...,ptr);
23:
  rcu_assign_pointer(*slot, item);
34:
  radix_tree_for_each_slot(...) {
    ret = fn(id,rcu_dereference_raw(*slot),...);
  }
39:
  const struct cred *cred = p;
  struct group_info *gi = cred->group_info;
43:
  for (g = 0; g < gi->ngroups; g++)
```

(c) possibly reported statements

```
5:
  switch (opcode) {
7:
  if (args || nr_args) {
34:
  radix_tree_for_each_slot(...) {
```

(d) possibly reported branches

Figure 1: Motivating Example

code to infer which path the program has executed and how data have been passed to understand the bug-triggering procedure. Then, analysts can recognize important program behaviors and corresponding bug-contributing instructions to learn the root cause. Thus, a good root-cause representation needs to reflect the runtime control relationships and data dependencies of bug-contributing instructions that represent important program behaviors leading to the bug.

The most popular root-cause representations are program elements (e.g., buggy instructions, basic blocks) [21,40,44,48] and predicates (boolean expressions capturing bug-triggering conditions) [13, 33, 41]. Both of them share a common feature that is isolated. They only present some isolated program states or code snippets. The analyst still needs to infer the missing control flow and data dependency outside root-cause representations to understand the whole bug-triggering procedure. It is relatively easy for user-space programs since they mostly have uniform entry and explicit function calls. However, it is difficult for the kernel. As most of the existing work cannot be directly applied to the kernel, we infer the possible root-cause representations they may generate for our motivating example to explain why they are not enough for understanding kernel bugs.

Predicates. Predicates are boolean expressions describing conditions of program states. Existing works like [13, 33, 41] usually consider three types of predicates: (1) Register and memory predicates describing the value constraint of a register or memory unit, e.g. `RAX < 10`. (2) Flag predicates describing the state of a flag register, e.g. `ZF == 1` representing the zero flag is set. (3) Control flow predicates describing the executed program path, e.g. `always_taken_to(A, B)` meaning the next basic block executed after A is always B.

Possible predicates for the motivating example are listed in Figure 1(b). As the bug is triggered under a certain path rather than a certain range of variables' values (like out-of-bound access), the most relevant predicates are about branch variables and control flow like `opcode==IORING_REGISTER_PERSONALITY`, `ZF==1`, and `always_taken(5, 6)` at line 5. However, such predicates are less relevant to the root cause of the bug. The root cause is that the pointer `id` with type `io_identity *` saved at line 23 is misused as `cred *` at line 39. Analysts need to manually infer the data flow of how the pointer `id` is propagated. However, inferring the propagation of `id` is difficult using the predicates, since the predicates do not reflect the data dependency crossing the syscall boundary between lines 23 and 34 and the indirect call boundary between line 34 and line 37. Inferring such data dependencies scattered across different

code snippets is quite difficult when no predicates directly imply these data dependencies. Not to mention inferring the entire bug-triggering procedure.

Program Elements. Crash and sanitizer reports are widely used for Linux kernel debugging. Crash reports typically show the last executed instructions that triggered an error, while sanitizers like KASAN [7] highlight bug-related instructions based on the bug type. However, they often lack clarity regarding root causes. First, they focus on runtime exceptions, which may not reflect the root cause. Second, the reported instructions are isolated. For example, in a use-after-free (UAF) bug, KASAN reports the allocation, free, and use sites. Despite accompanying call stacks, analysts often struggle to correlate them, understand their control flow and data dependencies, and pinpoint the root cause.

Spectrum-based methods like [21, 40] may report the statements shown in Figure 1(c) that would occur in the buggy execution but not in the normal execution. Inferring the bug-triggering procedure from statements is non-trivial. First, inferring the calling context is difficult. A general-purpose data-structure operator like `radix_tree_iter_replace` at line 20 might be called under various contexts. And the statement like line 39 might be executed in an indirect call (line 34). Identification of their calling contexts is labor-intensive. Second, inferring data dependencies is also hard. For example, if multiple general-purpose data-structure operators like `rcu_assign_pointer` and `rcu_dereference_raw` are called between different syscalls, it is difficult for analysts to confirm whether they access the same `slot` unless they trace back the source of the data structure and discover that they are accessed through the same `fd`. It is worth noting that the statements shown in Figure 1(c) represent an ideal, comprehensive set. Practically, spectrum-based methods do not always produce comprehensive results, as they rely on statistical likelihood to identify bug-inducing code and thus may yield false negatives. Consequently, their outputs can make uncovering the root cause more difficult.

Hypothesis-based methods like [44, 48] may report the divergent branches (equivalent to basic blocks) between normal and bug-triggering paths, shown in Figure 1(d). Unfortunately, none of these branches should be blamed for this bug. The code that should be blamed is line 39 which wrongly converts the type of pointer `p`, which is far from the reported branches. In this case, analysts need to further infer the entire bug-triggering procedure to understand the root cause.

2.3 Contextual Causality Chain

Since inferring the missing dynamic procedure from the isolated root-cause representations is difficult, we design a new root-cause representation *contextual causality chain (CC-chain)* that can intuitively show the integrated bug-triggering procedure and explain what has happened. The CC-chain of the motivating example is shown in Figure 2.

Different from isolated root-cause representations, the CC-chain is a structured and uniform representation of the integrated bug-triggering procedure. Formally, a CC-chain is a tuple $\mathcal{C} = (G, B)$, where $G = (V_I \cup V_C \cup V_V, E_C \cup E_D)$ is a heterogeneous directed acyclic graph, and B is a set of *unexpected behaviors*. V_I, V_C, and V_V denote the sets of instruction nodes, call-frame nodes, and value nodes, respectively. Intuitively, Figure 2 illustrates the CC-chain using source code lines in place of instructions and variables in place of registers and memory locations. The edges in E_C construct the *calling context*. A frame-instruction edge (f, i) indicates that instruction $i \in V_I$ is executed under the call frame $f \in V_C$. A frame-frame edge (f_1, f_2) indicates that the call frame $f_2 \in V_C$ is dynamically generated under the caller frame $f_1 \in V_C$. The edges in E_D construct the *data dependency*, describing input-output relationships across instructions. A data-dependency edge (v_{out}, v_{in}) denotes that the value v_{in} used by some instruction i_1 depends on the value v_{out} produced by another instruction i_2. Each unexpected behavior $b \in B$ is correlated to a specific instruction $i \in V_I$, meaning that the execution of i triggers b under its calling context and input values. By following data-dependency edges, the bug-contributing instructions in V_I with unexpected behaviors can be connected into a *causality chain*, which explains how unexpected behaviors are related and reveals the root cause of the bug.

The CC-chain of the motivating example clearly shows the integrated bug-triggering procedure. In the fifth syscall, the `id` is allocated and propagated to the `*slot`. When it is written to `*slot`, its type is `io_identity*`. Then, the eighth syscall reads the same `*slot` and misuses its type as `cred*`, causing a read-write type confusion. Then, the misused pointer `cred` is dereferenced, causing an out-of-bound read and getting an invalid pointer `gi`. Finally, the dereference of the invalid pointer `gi` causes a crash. Some dynamic procedures are difficult to infer with classic root-cause representations, such as the indirect control flow between lines 34 and 39, and the cross-syscall data dependency between line 23 and line 34. Oppositely, the CC-chain clearly shows them. CC-chain also clearly shows the root cause, i.e, type confusion, as it is the source of all the subsequent unexpected behaviors. According to the CC-chain, the developer could correct the type of `p` to fix this bug like in Figure 3.

2.4 Technical Challenges

Overhead of kernel tracing can be unaffordable. A straightforward approach is to record every executed instruction. But it would be unaffordable for kernel even with advanced hardware tracing techniques [47]. Fortunately, not all the instructions are necessary for root cause analysis. We can focus on the most possible instructions contributing to the bug and shrink the tracing range. As the bug is triggered by a PoC containing a series of system calls, we can focus on the kernel instructions executed by the PoC, including instructions

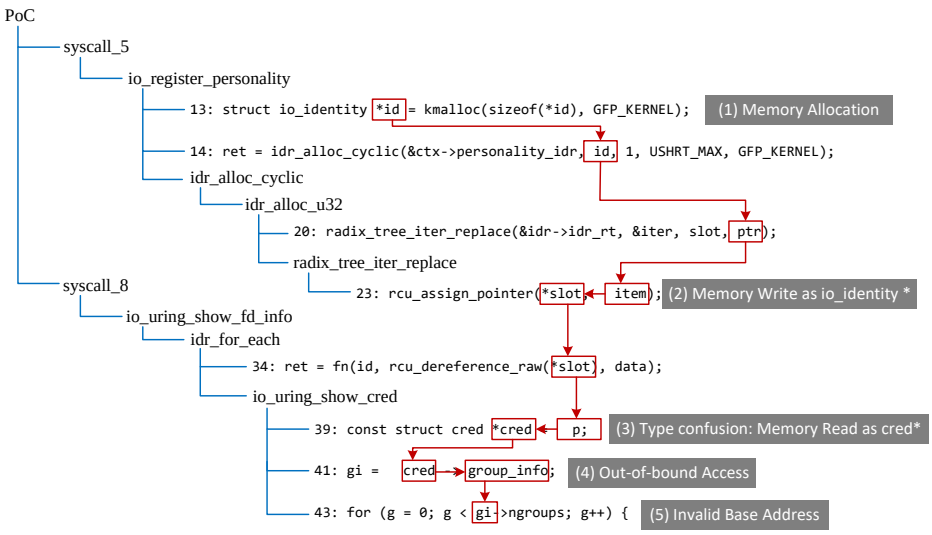


Figure 2: Contextual Causality Chain for the Motivating Example. The calling context is shown in blue. The data dependencies are shown in red. The causality chain contains all the statements and descriptions in gray boxes.

```
1  static int io_uring_show_cred(int id, void *p, void *data)
2  {
3  -   const struct cred *cred = p;
4  +   struct io_identity *iod = p;
5  +   const struct cred *cred = iod->creds;
```

Figure 3: Fix of Motivating Example.

directly executed in the invoked syscalls and indirectly executed in other tasks created by these syscalls. Such a selective tracing strategy can largely reduce the instructions to analyze. After selective tracing, we introduce a trace normalization process to further reduce trace size and provide important semantics for root cause analysis. First, we remove top-half interrupt handlers, i.e., fast interrupt code executed immediately upon an interrupt, as their execution is less relevant to buggy logic. Second, we model rich-semantic functions, i.e., frequently invoked kernel functions implementing high-level semantics (e.g., memory allocation), by abstracting their instruction-level execution into semantic operations.

Amount of context information to be recovered is huge. As raw instructions and memory accesses in the trace reflect little semantics to diagnose the root cause, we need to recover semantical contextual information like calling context and data dependency. Before learning the root cause, it is unclear which instructions contribute to the bug. Thus, all instructions' contextual information must be recovered. A naive approach to recover calling context is storing the entire call stack for each instruction. However, deep kernel call stacks cause high storage overhead and require extra processing to merge them into an integrated calling context for the CC-chain. For data dependency, the key challenge is identifying memory aliasing across kernel tasks. While concrete memory access addresses are recorded, matching all reads to their corresponding writes is time-consuming due to their large quantity.

To efficiently recover the calling context and data dependency, we design an emulation-like method. We analyze the recorded instructions sequentially and maintain a "current state" after analyzing each instruction. The "current state" is not merely the machine state captured in the trace. More importantly, it maintains a call tree of stack frames and a set of value nodes of registers and memories with the latest instructions modifying them. These structures encode the calling context and data dependencies associated with every instruction. By maintaining the call tree and the set of value nodes, we can recover contextual information in linear time and space complexity.

Precisely identifying bug-contributing instructions is difficult. Bug-contributing instructions must be related to the crash site. If a crashing instruction accesses memory with an illegal address, the instruction for calculating that address can be the root cause or a clue to other bug-contributing instructions. But even with this rule, too many instructions remain. Reporting all of them would confuse analysts. Generally, only a few instructions truly contribute to the bug. Since we have already recovered contextual information, we use it to identify the instructions that introduce and connect unexpected kernel behaviors. We apply an iterative algorithm to construct a chain of instructions exhibiting the most unexpected behaviors. This chain represents how the root cause is generated and leads to other unexpected behaviors until crashing.

3 Method

3.1 Overview

In this paper, we propose a root cause analysis system KernelRCA to generate the CC-chain for Linux memory corruptions. KernelRCA contains three components: selective tracing, contextual information recovery, and chain-style root cause analysis. The overall workflow is shown in Figure 4.

In selective tracing, we focus on three representative kinds of kernel tasks including system call, softirq, and queued work. We trace the executed instructions and memory accesses in these tasks that are directly and indirectly emitted by the PoC. Then, we introduce trace normalization that removes the irrelevant interrupts and models some rich-semantic functions to further reduce trace size and provide semantics for root cause analysis. After selective tracing, the number of instructions and memory accesses to analyze is largely reduced.

In contextual information recovery, we design an emulation-like approach to recover the calling context and data dependency from the raw trace. For calling context, we build a call tree by simulating the creation and destruction of the stack frame. For data dependency, we build a data dependency graph of the input-output dependencies among registers and memory. To be specific, we maintain a "current state" during emulation, including the latest created stack frame nodes and value nodes. Additionally, we attach semantic tags to the value node to provide semantics for root cause analysis.

In chain-style root cause analysis, we first recognize unexpected behaviors during the buggy execution. We model common unexpected behaviors of spatial, temporal, and semantic issues. The contextual information is used to recognize unexpected behaviors and corresponding instructions. Then, we perform an iterative algorithm to find a data dependency path on which the instructions raise the most unexpected behaviors. These instructions make up the causality chain in the CC-chain. Finally, we attach the contextual information and the descriptions of unexpected behaviors to instructions in the causality chain to generate the entire CC-chain.

3.2 Selective Tracing

3.2.1 Task-based Tracing

Root cause analysis requires gathering enough information about the bug-triggering procedure. Some bug-specific root cause analysis is designed for specific bug types like use-after-free [19] or data-racing [23]. They can efficiently gather necessary information based on the characteristics of their target bug types. For example, FREEWILL [19] leverages static analysis to identify the ref-count variables so that it can only trace the statements modifying these ref-counts. Unlike bug-specific root cause analysis, we aim to design a general-purpose approach. So, we are unaware of what information is needed when tracing. In this case, a straightforward approach is to record every executed instruction and memory access, which can be ensured to recover any intermediate kernel states precisely. However, even with advanced hardware features like Intel-PT, recording every kernel instruction still has unaffordable overhead [47].

To reduce the tracing overhead, we propose a selective tracing strategy. Linux kernel is a multi-task system that switches between various tasks. Most of the running tasks are irrelevant to the buggy behavior. Due to the buggy behavior being triggered by PoC, the task needed to trace should be directly or indirectly emitted by the PoC. In this paper, we consider major kinds of tasks in Linux kernel: *syscall (i.e. system call)*, *queued work*, and *softirq*. Syscall works like a function call. User-space programs actively invoke syscalls to request kernel functionalities. Queued work is a kind of delayed execution mechanism. If a task takes a long time to finish and is not emergent, the kernel can insert the task handlers into a work queue. The queued task will be executed later in a daemon process. Softirq is the bottom-half interrupt handler. When an interrupt arises, the kernel only performs the most urgent jobs immediately in the top half, and defer remaining jobs to the bottom half to ensure real-time performance.

We maintain a task set to determine which task needs to be traced. First, the main process of the PoC is added to the set when the PoC executes the `main` function. Then, we monitor whether a task in the task set emits a new task by instrumenting task management functions in Table 1, including creating a new process, enqueueing a queued work, and raising a softirq. The emitted tasks are then added to the task set. When a task is finished, we remove it from the task set. We use the PID, address of the `struct work`, and softirq index to identify these tasks respectively. During the kernel execution, when a task in the task set is scheduled, the tracing is enabled. And when it is scheduled out, the tracing is paused.

Table 1: Task management functions of kernel tasks.

Task	Function	Description
Process/Work	__schedule	Schedule In & Out
Process	_do_fork	Create
Process	do_exit	Exit
Work	insert_work	Create
Work	process_one_work	Start & End
Softirq	__raise_softirq_irqoff	Create
Softirq	__do_softirq	Start & End

We run the Linux kernel in a dynamic analysis framework S2E [14] to perform an instruction-level tracing. For each traced instruction, we record the values of commonly used registers before the instruction's execution. Our prototype is implemented on x86_64 thus we record general-purpose registers (e.g. RAX, R15), conditional registers, segment registers, and the program counter RIP. For memory accesses, we record the accessed address, size, and memory value.

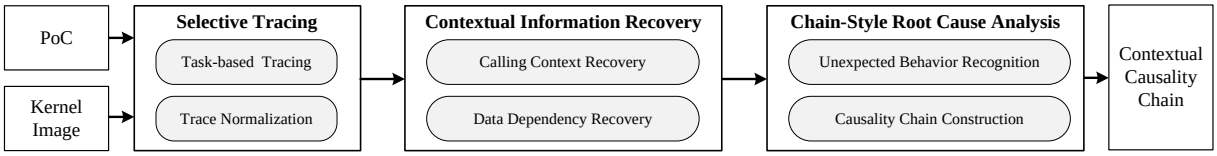


Figure 4: Workflow of KernelRCA.

3.2.2 Trace Normalization

Although selective tracing has filtered out irrelevant tasks, many instructions in the trace are still redundant. Besides, the trace only consists of raw instructions and provides little semantics useful for root cause analysis like memory management operations, e.g. allocation and free. Thus, we design a trace normalization process to further reduce the trace size and provide critical semantics.

First, we are not interested in the user-space instructions since we are diagnosing kernel bugs. Removing user-space instructions is trivial. User-space instructions can be distinguished by their address. The highest bytes of an address of user-space instruction is `0x0000`, while kernel-space is `0xFFFF`.

Second, the top-half interrupt can be removed since the bug hardly happens in it. We performed a simple study on 100 bugs and found that only 4 bugs involved top-half interrupt handlers. We use different strategies to recognize and remove the instructions of interrupts breaking user-space and kernel-space executions. Interrupt instructions that break user-space execution can be identified by their addresses, as they belong to kernel-space. However, this approach is ineffective for interrupts that break kernel-space execution. Due to the nature of the interrupt that breaks the control flow, we use a CFI-like method to distinguish interrupt instructions. Specifically, we compare whether the recorded next instruction matches the expected next instruction for every traced instruction. For a non-control instruction, the expected next instruction is right after the current instruction. For the conditional jump, the possible next instructions are right after the jump or the jump target. For `call`, we try to resolve the callee's address. For `return`, we maintain a stack of return addresses to infer the return target. Once the next instruction recorded in the trace does not match the expected next instruction, it indicates an interrupt happened. Until the expected next instruction is met, all the entries in the trace will be removed from the trace.

Third, we model some rich-semantic functions to provide important semantics and avoid the overhead of instruction-level analysis. Since using function modeling will lose the opportunity to diagnose the internal behavior of the function, we carefully choose the functions to model as few as possible. These modeled functions either provide critical semantics for root cause analysis or are seemingly irrelevant to the root cause. We systematically consider four types of functions. (1) Three scheduling functions: `__schedule`, `__do_softirq`, and `process_one_work`. They are modeled to remove their instructions to split the traces of different tasks. (2) We model exported functions (`EXPORT_SYMBOL`) that allocate or free memory in `mm/slab.c`, `mm/slub.c`, `mm/slob.c`, `mm/percpu.c`, and `mm/page_alloc.c`, since memory allocation and free semantics are important for root cause analysis. (3) We replace the sanitizer functions in KASAN with a blank model, as they are orthogonal to the functionality we want to diagnose. And (4) we assume functions involving floating-point arithmetic are irrelevant to the root cause of memory corruptions since they hardly participate in address computation. They will also be replaced with a blank model. It simplifies the implementation of data-dependency recovery by excluding floating-point operations.

We identify the instructions in the modeled functions similarly to how we recognize interrupts. We replace the whole function with a single function entry in the trace. A function contains the function name and necessary information to emulate the function behavior. For instance, we parse the size and allocated address from the trace for `kmalloc` and save them in the function entry.

3.3 Contextual Information Recovery

The root cause is an unexpected program behavior that causes the program to fall into an unintended state. Selective tracing only records raw instructions and memory accesses, which have little semantic information to diagnose the root cause. Semantic information like the calling context of each instruction and data dependency between instructions is useful to recognize unexpected program behavior. Besides, such information can help analysts better understand the integrated bug-triggering procedure. However, before knowing the root cause, we cannot know which instructions contribute to the bug and what contextual information is needed to recover. So we must recover the contextual information as much as possible. With the large amount of executed instructions, an efficient approach is required.

We design an emulation-like approach to recover the contextual information with linear time and spatial complexity. The basic idea is to analyze every instruction sequentially and emulate its functionality to maintain the stack frame and rebuild the data dependency. Additionally, we need special treatments to adapt the characteristics of the kernel.

3.3.1 Calling Context Recovery

It seems to be easy to recover the calling context of each instruction. A straightforward way is to create a new stack frame on the emulated stack at a call instruction and destroy the topmost stack frame at a return instruction. However, it has two problems. First, every instruction should have a calling context. Simply saving the whole call stack for every instruction will introduce huge overhead. Second, faithfully recovering the "real" calling context may not be a proper choice. For example, a softirq handler may execute in a normal process when the process changes its interruptiblity by calling `local_bh_enable`. In this case, the instruction's calling context in the softirq may look like Figure 5(a) that the softirq handler is a subroutine of the syscall. However, the softirq may have no relation with the syscall and can be triggered elsewhere. Such a calling context would mislead the root cause analysis algorithm and analysts that the syscall and the softirq have some relationship. Thus, each task should be treated separately when recovering the calling context like Figure 5(b). This issue arises from the interleaved execution of different tasks. Not only softirqs but also process scheduling and work queue scheduling encounter this issue (Figure 5(c,d,e,f)).

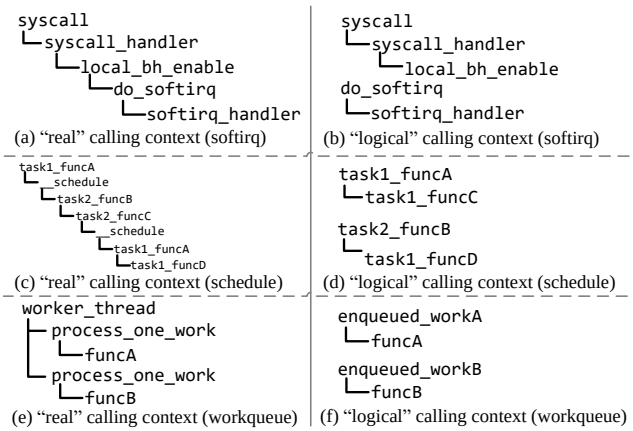


Figure 5: "Real" Calling Context v.s. "Logical" Calling Context.

To solve these two problems, we build a call tree as in Figure 2 for each task individually. The trace is divided by tasks after trace normalization. We separately analyze the traces for each task. We first create a root node representing the root call frame for each task. Every instruction in the task is sequentially scanned. For a call instruction, we create a new node representing a new call frame, point its parent to the topmost call frame on the stack, and push the new node into the stack. For a return instruction, we pop the topmost node on the stack. With the call tree, we can only save a single node for each instruction instead of the entire call stack, since the call stack can be recovered by iterating the parents of its

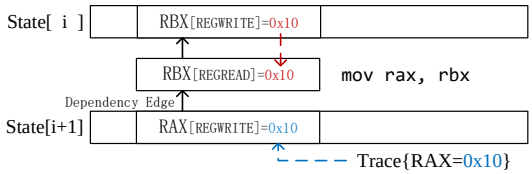


Figure 6: Construction of Data Dependency Graph.

node, which largely reduces the storage overhead. Another advantage is that recovering the calling context relationship between two instructions is convenient since their stack frame nodes share some common parents.

3.3.2 Data Dependency Recovery

The data dependency we discuss is the input-output dependency between the operands of instructions. The output of an instruction depends on its inputs, while the inputs may depend on other instructions' outputs. We first recover intra-task data dependency. Each instruction's functionality would be emulated sequentially in a task's trace. During emulation, we maintain a "current state" including a register map and a memory map. Each map contains a series of value nodes. Each value node records the concrete value, data size, timestamp when the value node is created, the location where the value is saved (register name/memory address), and a dependency list recording the value nodes that this node depends on.

We use the `mov rax, rbx` to explain the data dependency recovery (Figure 6). Suppose that the current timestamp is *i*. Instruction emulation creates input nodes based on its input operands, e.g., a new RBX value node. Its value, data size, and timestamp are obtained from the RBX node in the current state. Then, the RBX node in the current state is added to the new node's dependency list. Next, we create a new value node of RAX to represent the instruction's output, whose value and size are obtained from the trace. After emulation, the timestamp is incremented and assigned to the output nodes. According to the semantics of the `mov` instruction, the value of RAX depends on the value of RBX. So we add the input RBX node to the dependency list of the output RAX node. Given the complexity of x86_64 instructions, we lift them to VEX IR [6] and emulate the IR to extract operand dependencies. Finally, the current state is updated by replacing the old RAX node with the new one. Besides, we create a semantic tag of how the value is created (shown in Table 2), which provides semantics for root cause analysis. Emulating memory access follows a similar way. We attach the *ADDR* tag to the address node of memory access.

For the modeled function, we create the value nodes and data dependency according to the function model. For example, the output of `kmalloc` is a value node of RAX whose value is the allocated address from the trace and the semantic tag is *MEMALLOC*. We also create value nodes for each allocated

Table 2: Semantic Tags of a VNode.

Semantic Tag	Description
UNKNOWN	Value whose source is unknown
TEMP	Value with no special semantic
CONST	Constant value
ADDR	Value representing a memory address
MEMALLOC	Address or memory bytes of memory allocation
MEMREAD	Values read from memory
MEMWRITE	Values write to memory
MEMFREE	Address or memory bytes of freed memory
REGREAD	Values read from register
REGWRITE	Values write to register

memory byte in the memory map in the current state.

After recovering the intra-task data dependency, we continue to recover the inter-task data dependency. Different tasks would communicate through accessing the same memory. Such memory accesses cause inter-task data dependency. First, we find all the value nodes whose semantic tag is *MEMREAD* but the dependency list is empty. It means the source of the memory read is out of the task since all the intra-task data dependency has been recovered. Then, we search the *MEMWRITE* value nodes from the final states in other tasks' scheduling slices before the memory read. If a *MEMWRITE* node A is the latest node writing this memory and has the same address and value as the *MEMREAD* node B, node A is the source of B. We add A to the dependency list of node B.

Note that another research topic, dynamic slicing [10, 45], also builds data dependency graphs from execution traces. Dynamic slicing aims to locate variable-influencing statements from data dependency, whereas KernelRCA focuses on recognizing unexpected behaviors. As their purpose differs from ours, the data dependency graph of dynamic slicing cannot be directly used. First, dynamic slicing typically constructs dependencies based on program variables, where each variable is represented by a single node, even if the statement is executed multiple times. This requires additional effort to distinguish dependencies in different contexts. In contrast, our data dependency graph creates a new value node for each executed instruction with its own context. Second, dynamic slicing often uses language-specific features to infer data dependencies [27], while KernelRCA observes them directly from the binary trace. Third, we attach semantic tags to value nodes to support root cause analysis.

3.4 Chain-style Root Cause Analysis

To diagnose the root cause, we first recognize the possible unexpected behaviors. We first use contextual information to recognize unexpected behaviors. Then we propose a chain-style analysis to find unexpected behaviors that may be the root cause. It finds the longest chain that contains the most unexpected behavior. The insight is that the root cause may first cause a series of unexpected behaviors and eventually result in observable phenomena such as crashes.

3.4.1 Unexpected Behavior Recognition

Based on our observation, three kinds of unexpected behaviors may cause memory corruption: spatial issues, temporal issues, and semantic issues.

Spatial Issues. In most cases, the kernel should only execute kernel-space instructions and access kernel-space addresses. We check whether the accessed address is a valid kernel address for each memory access and indirect branch. If the concrete value of a value node in the data dependency graph with semantic tag *ADDR* is not a valid kernel-space address, we think an unexpected behavior is raised by the instruction of this value node, and attach a description that "Invalid Base Address" to this instruction if it is a memory access or "Invalid Code Address" if it is an indirect branch. The only exception is that a user-space memory is accessed in I/O functions like `copy_from_user`. Then we find the source value node of the address and attach a description that "Source of Base Address" to its instruction. Further analysis is performed when needed: addresses in $[-1024, -1]$ from return values indicate error code misuse; legal-looking addresses from negative integers suggest integer overflows; for allocated addresses, we check if the access offset exceeds the allocation size to detect out-of-bounds accesses.

Temporal Issues. The dynamically allocated memory regions can only be accessed in its life-cycle from allocation to free. In most cases, an allocated memory region should be initialized before usage. We mainly discuss use-before-initialization (UBI) and use-after-free (UAF) in temporal issues. We infer the life cycle of memory regions from the modeled memory allocation and free functions.

To detect UBI, we check all the value nodes with semantic tag *MEMREAD*. We first extract the value node representing the accessed address. Then, we recognize whether the address belongs to the heap or stack. If the source of the address node is memory allocation, it is a heap address. If the address node is calculated from a stack pointer register, it is a stack address. For a heap *MEMREAD* node that does not depend on a *MEMWRITE* node after the memory allocation is a use-before-initialization. The instruction of this *MEMREAD* node is regarded to raise an unexpected behavior of heap UBI. We attach descriptions "Source of Memory Allocation" and "Heap UBI" to the instructions of *MEMALLOC* node and the *MEMREAD* node respectively. For stack memory access, we also check whether it depends on a *MEMWRITE* node. If so, we increasingly check whether the *MEMWRITE* node can be the valid initialization. The valid stack initialization requires the corresponding *MEMWRITE* node to write on a living stack frame, i.e. its own stack frame or one of the parent frames of the *MEMREAD* node. It can be checked by comparing the corresponding stack frame nodes on the call tree. Otherwise, we find a stack UBI and attach the descriptions "Source of

Stack Pointer" and "Stack UBI" respectively for the node of the stack pointer and *MEMREAD*.

To detect UAF, we first recognize the dangling pointers. When a *ADDR* node becomes the parameter of a memory-free function, it becomes a dangling pointer. All the copies of this pointer living in memory and registers also become dangling pointers. Then, we check the value nodes with semantic tags *MEMREAD* and *MEMWRITE*. If a node's address is derived from a dangling pointer, the instruction is marked as a UAF. We label the free and access instructions as "Freed Memory" and "Heap UAF." Stack UAFs are treated as stack UBIs, since stacks should be reinitialized before reuse.

Semantic Issues. The semantic issue is caused by improperly using the memory content. Here we take type confusion as an example. Type confusion means a value in memory is misused as a different and inconvertible type of its original type. To identify the type confusion, we infer the data type of value nodes after constructing the data dependency graph. For a *REGREAD* node, we directly retrieve its type in the debug information. For *MEMREAD* and *MEMWRITE* nodes, we extract the base register and offset of the memory access. Using the base register and offset, we can also retrieve the data type. Due to compiler optimization, some register and memory type information is not present in the debug information. We use a type propagation to infer the missing data type. For a value node with a known type, we recursively propagate its type along the dependency edge forward and backward.

We build a type lattice to check type convertibility. Inherited structures can be converted to the type of their first member if it's also a structure. A partial order is added between such types. All basic types are treated as mutually convertible. If the types a pair of *MEMREAD* and *MEMWRITE* nodes are not convertible, type confusion is raised by their instructions. These instructions are also attached with the description "Type Confusion" including their types.

Although sanitizers like KASAN can also detect unexpected behaviors such as out-of-bound accesses and use-after-free, our approach differs significantly. Sanitizers prioritize runtime efficiency and rely on shallow indicators (e.g., corrupted flags), limiting their ability to capture deeper, implicit anomalies. For instance, sanitizers only trigger when a bad address is dereferenced, but cannot detect the unexpected behavior when the bad address is generated. In contrast, KernelRCA's unexpected behavior detection leverages recovered dependencies and calling contexts, allowing it to find more types of unexpected behaviors and pinpoint them more accurately.

3.4.2 Causality Chain Construction

We model root cause analysis as finding the longest instruction chain in the data dependency graph from unexpected behaviors to the crash site, where each instruction is either an unexpected behavior or bridges between them. We find the longest chain because this helps uncover the deepest cause. As the data dependency graph is a directed acyclic graph, we apply an iterative algorithm to find this chain.

We initialize a count of unexpected behaviors to 0 for all the instructions. Then, we iteratively update the count along the dependency edges.

$$\text{count}(u) = \max_{v \in \text{pred}(u)} \{\text{count}(v) + \text{UB}(v)\}$$

where $\text{UB}(v)$ is the number of unexpected behaviors of instruction v and $\text{pred}(u)$ is the set of instructions whose output is the input of u. When the count of the crash site no longer increases, the root-cause chain is found. We attach corresponding contextual information and descriptions of unexpected behaviors to the chain, which make up the entire CC-chain to explain the integrated bug-triggering procedure and facilitate understanding of the root cause.

4 Evaluation

4.1 Evaluation Setup

KernelRCA is implemented with about 8K LoC of C/C++ and 2K LoC of Python. We implement selective tracing on S2E [14], a dynamic analysis framework based on QEMU [12]. Although some hardware features can help trace the executed instructions [17, 47], they hardly support tracing the memory access on most architectures. We leave hardware-assistant tracing to future work. We use the disassembler *Capstone* [1] to parse the kernel binary, *libvex* [3] to lift the binary to IR, *libdwarf* [2] to parse the debug information. We run KernelRCA on a server running Ubuntu 18.04 and having an Intel Xeon 6254 CPU and 377 GB of memory.

To build the evaluation dataset, we collect the publicly disclosed bugs on Syzbot [4] with the following criteria. First, the kernel version of the bug lies between v5.0 and v5.14 on the upstream.[1] Second, the bug has a C-language PoC. Third, the bug title includes keywords in Table 5 indicating a memory corruption. Also, we require an available patch of the bug to validate the correctness of root cause analysis. We obtained 306 bugs with these criteria. We randomly select 100 of them and spend 3 weeks reproducing these bugs. Among these 100 bugs, 65 are successfully reproduced. The causes of reproduction failures are discussed in §5. The reproduced 65 bugs are used as our dataset (Table 9). The dataset includes various types of memory corruptions with varying levels of complexity. The corresponding patches range from 1 to 150 LoC, while the PoCs range from 29 to 7,176 LoC.

[1] KernelRCA is not limited to specific kernel versions; the version restriction exists solely for the convenience of S2E instrumentation.

4.2 Effectiveness of Root Cause Analysis

Baselines. Most root cause analyses are developed for user-space programs. They rely on infrastructures that are less compatible with the Linux kernel, making their methods inapplicable in our scenario. Instead, we adopt widely used, scalable approaches in kernel debugging practice: crash/KASAN reports and Syzbot's cause bisection [8]. Crash/KASAN reports were introduced in §2.2. Syzbot's cause bisection can automatically identify the bug-inducing commit that is also a kind of "root cause". We compare KernelRCA against these methods to show its effectiveness.

Criteria. Objectively evaluating root-cause analysis is challenging, since the root cause is a semantic concept. We follow the previous works [13,33] using the patch as ground truth. In this case, we first verify the correctness of patches. We begin by testing whether each patch suppresses the bug when executing the PoC. Then, we manually review the patch and its associated commit message to ensure that it indeed resolves the root cause of the bug. All patches in our dataset were confirmed to be correct through this process.

First, *to judge the correctness of CC-chain*, we manually compare kernel behaviors before and after applying the bug-fixing patch. A CC-chain is considered correct only if it captures the key instructions that trigger the root cause, as reflected in the semantic changes introduced by the patch. Partial matches that miss essential causes are marked as incorrect. Two experts independently conduct the evaluation and cross-validate the results to ensure reliability.

Second, *to compare with baselines*, we calculate a root-cause distance. The distance measures how far the reported instructions are from the ground-truth instructions. The ground-truth instructions are the instructions deleted/inserted by the patch. We use the following as the reported instructions for each method: (1) all instructions in the CC-chain for KernelRCA; (2) the crashing instruction for crash reports, the allocation, free, and access instructions for KASAN-UAF reports, and the allocation and access instructions for KASAN-OOB reports; and (3) all instructions modified by the bug-inducing commit identified by Syzbot's cause bisection. For each bug, we calculate the minimal interval between every reported and ground-truth instructions in the selective execution trace as the root-cause distance, since considering irrelevant tasks' execution is meaningless. This distance is similar to "ΔRoot Cause" in [44]. They measure it at the basic block level, whereas we measure it at the instruction level.

Correctness. KernelRCA correctly diagnoses 54 among 65 bugs, covering a spectrum of crash types and bugs from simple to complex. These results are shown in Table 3. The CC-chains exhibit patterns shown in Table 4. Most chains provide more information about the root cause than the crash report. For example, the chain 01 and 04 are representative patterns for the general protection fault and null pointer dereference. The bug report only tells that they access an invalid address.

Table 3: Correctly diagnosed bugs. Crash Type is defined in Table 5. The number over 1,000 is abbreviated. BID=Bug ID. CT=Crash Type. D.C.=root-cause distance of CC-chain. D.B.=root-cause distance of the Syzbot bisection result. D.R.=root-cause distance of crash/KASAN report.

BID	CT	Chain	D.C.	D.B.	D.R.	BID	CT	Chain	D.C.	D.B.	D.R.
2c91	GPF	01	19	0	3K	6922	GPF	01	0	∞	18
8c77	GPF	01	3	∞	17	8e59	GPF	01	0	∞	0
91f7	GPF	01	2	∞	3	c5c6	GPF	01	176	∞	362K
f451	GPF	01	33K	∞	46K	a8e5	GPF	01	202	∞	18K
1bc7	GPF	01	0	0	11	a0f5	GPF	01	8	8	9
691c	UPR	01	15	∞	18	15bf	GPF	01	0	0	0
02dd	GPF	01	0	∞	0	d35e	KNP	01	0	∞	0
defb	NPD	02	0	∞	1K	2656	NPD	02	1	∞	1K
8dba	GPF	03	2	12	32	449f	GPF	03	8	1	11
57c3	GPF	04	21K	0	22K	c50f	GPF	04	0	∞	0
396f	GPF	04	450	∞	52K	d5cd	GPF	04	0	∞	0
5999	GPF	04	0	∞	0	b44a	GPF	04	0	13	0
6c6a	GPF	04	0	∞	0	2364	GPF	04	0	0	45
3674	GPF	04	0	0	11K	1fbc	GPF	04	0	189	0
4187	GPF	04	20K	71	62K	f9cf	GPF	04	0	177K	16K
ae31	GPF	04	0	∞	0	990d	GPF	04	0	35	47
7b96	KNP	04	0	0	0	12c0	KNP	05	2K	∞	304K
a139	UAF	06	841	690	2K	6312	KNP	06	0	∞	21K
94ed	UAF	06	21	∞	1,160K	4b1e	OOB	07	3	∞	1K
694d	OOB	07	0	167K	91	3aac	OOB	07	0	251K	296
3f6d	OOB	07	161K	∞	4,904K	2e1c	OOB	07	26K	0	26K
f1d7	OOB	07	6	∞	13	1f0c	OOB	07	15K	0	15K
24a2	OOB	07	0	∞	0	16a7	OOB	07	36	∞	106K
3d67	OOB	07	1	115	9	1708	OOB	07	2	∞	20
578c	UPR	08	9	∞	813K	c580	GPF	09	2	85	2
afa3	UPR	09	3K	∞	343K	f56b	GPF	10	0	∞	1
b66d	GPF	11	0	4K	0	5ba5	GPF	12	0	106K	13K

Instead, the CC-chain reveals the deeper cause that the invalid address comes from a UBI memory access or a non-address constant value, implying the potential fix that supplements proper initialization or checks the validity of the address. For a few bugs like 5ba5b3f, we find a complex CC-chain with multiple causes. This chain clearly shows the root cause of type confusion that raises other unexpected behaviors like OOB access, which eliminates the interference of non-root causes.

In these correct cases, six of their root-cause chains contain false positives, i.e., fake unexpected behaviors. Three fake UBIs are caused by the initialization out of the PoC-triggered task. One fake UBI is caused by the KernelRCA's unawareness of the implicit initialization in memory allocation functions like `kzalloc`. Two fake type confusions are caused by the imprecise type information in the debug information.

We analyze the failing reasons for the remaining 11 bugs. Five bugs involve data dependency that is generated out of the PoC-emitted tasks, e.g. accessing an allocated but uninitialized memory region that already existed before the PoC runs. Three bugs involve some complex instructions like `add rdi,[r12+0x1a8]` to calculate a memory address. In order to discover deep causes, KernelRCA needs to parse and track the base address. However, it fails to determine whether the

Table 4: Observed CC-chains. (Causes in brackets sometimes occur)

No.	Root Cause Description in CC-chain
01	MEMALLOC → Heap UBI → Invalid Base Address → Invalid Memory Access
02	MEMALLOC → Heap UBI → Invalid Code Pointer
03	Stack UBI → (Invalid Base Address →) Invalid Memory Access
04	CONST → Invalid Base Address → Invalid Memory Access
05	CONST → Invalid Code Pointer
06	MEMFREE → Dangling Pointer → Use-After-Free
07	Out-of-bound Access → Invalid Memory Access
08	Overflowed Integer → Invalid Memory Access
09	Error Code as Address → (Invalid Base Adress →) Invalid Memory Access
10	MEMALLOC → Invalid Base Address → Invalid Memory Access
11	Type Confusion → Invalid Base Address → Invalid Memory Access
12	Type Confusion → Out-of-bound Access → Invalid Base Address → Invalid Memory Access

base address is the original `rdi` or the memory `[r12+0x1a8]`, which prevents further analysis. One bug involves implicit data dependency between branch variables and in-branch variables, which has not been supported yet. One bug meets out-of-memory during root cause analysis since it involves large loops that generate too many intermediate states. One bug involves type confusion but the debug information lacks the type information of a critical register.

Comparison with Crash/KASAN Report. On average, KernelRCA's root-cause distance is only 45.6% of the crash/KASAN report, indicating KernelRCA's result is much closer to the root cause. KernelRCA achieves a root-cause distance of less than 30 in 41 cases, compared to only 26 for the crash/KASAN reports, demonstrating its advantage in precision. Moreover, KernelRCA achieves a zero root-cause distance in 26 cases, directly pointing to the patch location, while only 14 cases do so in the crash/KASAN reports.

Comparison with Syzbot's Cause Bisection. In 41 cases, KernelRCA achieves shorter root-cause distances than Syzbot. Syzbot outperforms KernelRCA in only 7 cases. In the cases where Syzbot performs better, the root causes of the bugs are more logical, rather than being limited to memory issues. Due to KernelRCA's instruction-level analysis, it can pinpoint root causes more precisely than Syzbot's coarse-grained, commit-level bisection. Notably, in 30 cases, the root-cause distance of the bug-inducing commit reported by Syzbot is ∞. Among these, 18 cases result from Syzbot failing to complete the bisection. We manually analyze the remaining 12 cases and find that Syzbot has identified unrelated commits. These results show the robustness of KernelRCA, especially in scenarios where Syzbot's analysis fails or misleads. Overall, KernelRCA offers a more accurate and practical solution for root-cause diagnosis in kernel debugging.

4.3 Ablation Study

4.3.1 Overhead

The measurement of KernelRCA's overhead is shown in Table 8. The average trace size is 1.39 GB, while the largest and smallest are 11.50 GB and 0.10 GB. KernelRCA needs 97 seconds to diagnose a bug on average. The tracing takes most of the time. Other stages are much faster since only a small fraction of instructions would be analyzed after selective tracing and trace normalization.

4.3.2 Effectiveness of Selective Tracing

We use a "raw" setting as the baseline that neither enables task-based tracing nor trace normalization to trace the kernel. Then we subsequently enable task-based tracing and trace normalization. The number of instructions in the trace to analyze is shown in Table 6. The raw traces contain 100K to 100,000K of instructions. Task-based tracing averagely reduces the trace size to 0.32× of raw traces. Our evaluation is performed on a low-noise system with few background processes. Task-based tracing would be more effective in a noisy system. Combining task-based tracing and trace normalization, about 95% of the instructions are abandoned for root-cause analysis.

4.3.3 Effectiveness of Contextual Information for Root Cause Analysis

Contextual information contributes to precisely identifying the instructions associated with unexpected behaviors. We compare the number of instructions before and after using the contextual information to identify the unexpected behaviors. The results are shown in Table 7. Tens of thousands of instructions may have data dependency with the crash site. Such a large quantity may confuse the analysts. After considering the contextual information, bug-contributing instructions can be precisely identified. On average, the contextual information helps to filter out about 99% irrelevant instructions from all the data-dependent instructions of the crash site. For 43 bugs, the number of bug-contributing instructions is fewer than 10, which is quite precise for human analysts.

4.4 User Study

Design. We design a questionnaire to evaluate KernelRCA's helpfulness in kernel debugging practices. The questionnaire has three parts. The first part collects the characteristics of participants, including their working experience and areas. The second part contains several bugs in our dataset. Participants first review the crash/KASAN report and assess their understanding of the root cause. Then, they review the CC-chain and reassess their comprehension. We develop an automatic tool to visualize the CC-chain, which is publicly available in our artifact. The understanding of the root cause is quantified into five levels. Participants select a level based on their comprehension. In total, we selected 18 cases covering all crash types and CC-chain types. For balance, we designed six different questionnaires, each containing three distinct cases. Each questionnaire was distributed as evenly as possible. The

third part collects participants' overall opinions of how KernelRCA and its components help understand the bug, and whether KernelRCA helps fix it. An example of the questionnaire is available at `https://github.com/seclab-fudan/KernelRCA/blob/master/tools/survey.md`.

We invite 27 experts from academia and industry who conduct research and development in the Linux kernel to attend our study. Finally, we received 19 responses. Among the participants, five have less than one year, ten have 1–3 years, and four have 3–6 years of experience working with the Linux kernel. Their experiences cover development, optimization, testing, bug analysis, exploitability assessment, etc. The characteristics of participants are in Figure 8.

Results. Statistics are shown in Figure 7. We find that after reviewing the CC-chain, the comprehension level has increased for all bugs from 2.28 to 3.70 on average. The result ($p = 2.24e-09$) of Mann-Whitney U-test [31] shows that participants' understanding improved significantly. For cases `b66d`, `5ba5`, and `8dba`, the comprehension level increases most significantly. They share common features: (1) The kernel only reports general protection faults with the crash site. (2) The accessing address is irregular (neither zero pointer nor in kernel address format). Such crash reports hardly reveal useful information for debugging. Two of them involve cross-syscall data dependency, making it hard to track back from the reported crash point. Oppositely, CC-chain directly shows the critical data dependency and the root cause, e.g., type confusion and stack UBI, which largely facilitates bug comprehension. For `d35e`, `6312`, and `afa3`, the increment of comprehension level is not significant. Although CC-chains show the bug-triggering procedure, analysts still need to be familiar with the specific subsystem logic to understand the bug. Overall, participants rated a 4.05 score of the KernelRCA's helpfulness (from 1 to 5). Components of CC-chain, including calling context, instruction chain, instruction description, and data dependency, provide valuable information. For bug fixing, 16 participants think KernelRCA's result provides important code positions to guide the debugging. One participant thinks s/he has learned how to fix the bug after reviewing the CC-chain. These results demonstrate that KernelRCA is helpful for bug understanding and fixing.

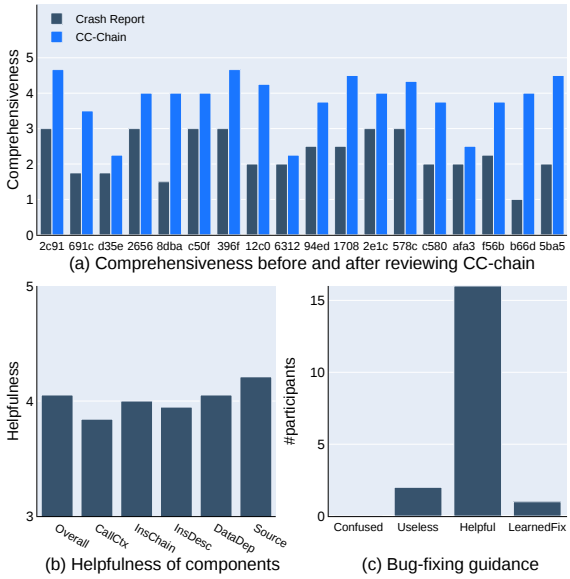


Figure 7: User Study Results.

4.5 Case Study

We additionally present a case study distinct from our motivating example. Bug `94ed` is a complex UAF involving interleaved tasks. KernelRCA produces a clear CC-chain that exposes the root cause and facilitates bug understanding and fixing.

5 Discussion

Bugs Involving Out-of-PoC Logic. Currently, KernelRCA traces only tasks triggered by the PoC. If bug-contributing instructions run outside this range, e.g., during kernel startup or in unrelated tasks, diagnosis may fail. However, tracing from startup is too costly, and identifying relevant tasks in advance is also difficult without prior bug knowledge.

Unreproducible Bugs. Reproducible PoC is a prerequisite for most RCA approaches including KernelRCA, yet achieving reproducibility can be challenging. Several potential causes exist. First, PoCs generated by fuzzers may be unstable due to the complex kernel states accumulated during persistent fuzzing [28], which are hard to reconstruct. Second, non-deterministic bugs add further difficulty. Tools like AITIA [20] may help in such scenarios. Lastly, some bugs lack PoCs entirely. Approaches like SyzDirect [37] can help generate one from textual reports.

Even with a seemingly correct PoC, understanding why it fails to reproduce a bug is non-trivial. Without a working reproducer, analysts must manually inspect source code and crash reports, iteratively adjusting the initial PoC to reproduce the bug and identify failure causes. As it is highly labor-intensive and challenging, we didn't thoroughly study the reproduction failures when constructing our dataset. We sampled and analyzed three failures and found their causes to be highly varied. One was due to an insufficient number of forked threads. One PoC employs a unstable race-based technique to trigger a bug that does not need to race actually. Another was disturbed by an unrelated bug. These diverse causes defy simple categorization. In short, both understanding the causes of reproduction failures and reliably reproducing bugs remain challenging problems that need further research.

Implicit Data Dependency. KernelRCA is not able to handle the implicit data dependency between the branch condition and the in-branch variable. Handling implicit data dependency

is a difficult problem [22]. Simply adding dependencies between all the branch-condition variables and in-branch variables may introduce much analyzing overhead and many false positives. We leave this problem in the future.

Scalability for More Bug Types. We focus on memory corruption due to its prevalence and high risk. However, KernelRCA can be extended to other kinds of bugs when provided with the knowledge to recognize the corresponding unexpected behaviors from contextual information. The remaining components like tracing strategy and chain-style analysis require almost no changes.

6 Related Works

Reverse Debugging. Reverse debugging makes use of post-mortem artifacts like memory dumps to diagnose the root cause. CREDAL [42] paid attention to precisely recovering the control flow. RETracer [15] proposes a forward analysis to infer the intermediate values that cannot be recovered by pure reverse execution. POMP [43] uses hypothesis testing to validate the recovered states. Increasingly, POMP++ [32] uses value-set analysis (VSA) to discover possible memory aliases. DEEPVSA [18] combines the VSA with deep learning to make it more precise. KernelRCA has two differences from reverse debugging. First, PoC is available to diagnose the fuzzer-found bugs. Tracing PoC to recover the contextual information is much more precise than backward reasoning. Second, they report isolated instructions with data dependencies of the crash site and include many false positives.

Delta Debugging. Delta debugging compares the differences between buggy behaviors and normal behaviors to show the root cause. The program behaviors can be represented as predicates [13, 24, 29, 30] or spectrums [9, 21, 39]. Statistical methods are commonly used to compare the differences among predicates [29,30] and spectrums [16]. However, delta debugging faces two limitations in our case. First, it produces isolated representations (e.g., predicates or spectra), requiring extra effort to reconstruct the full bug-triggering process. Second, it relies on high-quality input pairs with similar functionality but differing normal/buggy behaviors, which are hard to obtain for the OS kernel. Although some work leverage input mutation [36, 48] and state mutation [33] to mitigate this problem, they introduce much overhead in the mutation stage. Instead, our method only requires a single buggy input (PoC), which is more practical for real-world kernel bugs.

Type-Specific Root Cause Analysis. KernelRCA is a general-purpose root-cause analyzer. Oppositely, type-specific RCA focus on accurately diagnosing a specific type of bug. Snorlax [23] models possible data racing patterns to check whether they occur in runtime execution. AITIA [20] enumerates the possible data racing order with forced execution to find what racing order would lead to the bug. FREEWILL [19] models the incorrect reference counting to diagnose the UAF bugs caused by ref-count miscounting. MemRay [49] diagnoses memory corruption caused by invalid data structure reference. These methods could discover specific semantical unexpected behaviors like ref-count miscounting or data racing. Although KernelRCA focuses on memory issues, it can collaborate with other approaches to uncover semantic causes and offer integrated results.

7 Conclusion

This paper proposes a novel root-cause representation, contextual causality chain (CC-chain), to better facilitate the analysts to understand the root cause of memory corruptions in Linux kernel. CC-chain contains a causality chain of instructions with contextual information, relieving analysts from the labor-intensive reasoning of the bug-triggering procedure from the isolated root-cause representations. We design an automatic root cause analysis system KernelRCA including selective tracing, contextual information recovery, and chain-style root cause analysis to automatically generate the CC-chain. KernelRCA effectively diagnoses 54 real-world memory corruptions in the Linux kernel. The user study demonstrates that CC-chain is helpful in bug understanding and fixing.

Acknowledgment

We would like to thank the anonymous reviewers for their insightful comments that helped improve the quality of this paper, and Xin Tan, Xin Xiong, and Yifei Hou for their valuable suggestions and assistance. Yuan Zhang and Min Yang are the corresponding authors. Min Yang is a faculty of Shanghai Institute of Intelligent Electronics & Systems, and Engineering Research Center of Cyber Security Auditing and Monitoring, Ministry of Education, China.

Ethical Considerations

We performed a systematic stakeholder-based ethics analysis.

Potential Stakeholders

Research Team; User Study Participants; Linux Developer/User Community

Impacts

– Research Team
  – Respect for Persons: No influence
  – Beneficence: No influence
  – Justice: No influence
  – Respect for Law and Public Interest: No influence
– User Study Participants
  – Respect for Persons: Mitigated
    – Participation is entirely voluntary. Participants can choose whether to respond to the questionnaire based on their own will and interest.

– We clearly inform participants about the study's purpose, which is to investigate bug root causes and evaluate our new tool compared with existing techniques.
– To protect participant privacy, we do not collect any identifiable information. Responses are gathered through an anonymous system rather than via non-anonymous channels like email.
– Only aggregate statistics (e.g., overall participant characteristics) are published. No individual participant characteristics are disclosed to prevent potential privacy risks.

– Beneficence: No harm

– The survey contains no harmful content and does not require participants to discuss sensitive or harmful topics.

– Justice: No harm

– All participants are treated equally. Each questionnaire containing the same set of questions to ensure fair workload.

– Respect for Law and Public Interest: No influence

– Linux Developer/User Community

– Respect for Persons: No influence

– Beneficence: Benefits outweigh Harms

– The disclosed bug analyses are based on vulnerabilities already publicly reported by Syzbot and fixed in the upstream Linux kernel and major distributions. Our analysis does not increase any security risk.

– Incorrect RCA results may mislead analysts, leading to additional effort or even incomplete fixes. However, KernelRCA's accuracy significantly outweighs its error rate, and overall, its contribution to vulnerability fixing far exceeds the potential negative impact.

– Justice: No influence

– Respect for Law and Public Interest: No influence

Decision Summary

– We disclose root-cause analyses for bugs that are already public and fixed, introducing no additional security risks.
– Participants are fully informed of the study's purpose and content before participation.
– No personally identifiable information is collected. Responses are anonymous.
– Participants freely decide whether to reply to the questionnaire.
– Published results include only aggregated findings on participant feedback, without any personal data.
– We publish only statistical summaries of participant characteristics to prevent leakage of personal information such as work experience.
– We release the full implementation of KernelRCA and acknowledge the potential for abuse by attackers in vulnerability analysis. However, while attackers may gain faster understanding of vulnerabilities, developers will also respond and patch them swiftly under the help of KernelRCA, thereby reducing the window of exposure. In our view, it does not increase the overall security risk.

Open Science

The full implementation of KernelRCA is open-sourced at `https://doi.org/10.5281/zenodo.17858513`. This artifact can also be available at `https://github.com/seclab-fudan/KernelRCA` or `https://github.com/xiaoguai0992/KernelRCA`.

The `s2e` and `rca` directories contain the full source code of KernelRCA. The `tools/causality_chain_ui` directory provides the visualization tool for the text reports generated by KernelRCA. The `dataset` directory has the dataset used in our evaluation.

References

[1] Capstone - The Ultimate Disassembler, 2024. `https://www.capstone-engine.org/`.

[2] DWARF Page, 2024. `https://www.prevanders.net/dwarf.html`.

[3] libvex in Valgrind, 2024. `https://sourceware.org/git/?p=valgrind.git;a=tree;f=VEX;hb=refs/heads/master`.

[4] Syzbot dashboard of Linux upstream, 2024. `https://syzkaller.appspot.com/upstream`.

[5] syzkaller - kernel fuzzer, 2024. `https://github.com/google/syzkaller`.

[6] Intermediate Representation. `https://docs.angr.io/en/latest/advanced-topics/ir.html`, 2025.

[7] Kernel Address Sanitizer (KASAN), 2025. `https://www.kernel.org/doc/html/latest/dev-tools/kasan.html`.

[8] Syzbot Cause Bisection, 2025. `https://github.com/google/syzkaller/blob/master/docs/syzbot.md#bisection`.

[9] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. An evaluation of similarity coefficients for software fault localization. In *2006 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, pages 39–46. IEEE, 2006.

[10] Hiralal Agrawal and Joseph R Horgan. Dynamic program slicing. *ACM SIGPlan Notices*, 25(6):246–256, 1990.

[11] Piramanayagam Arumuga Nainar, Ting Chen, Jake Rosin, and Ben Liblit. Statistical debugging using compound boolean predicates. In *Proceedings of the 2007 international symposium on Software testing and analysis*, pages 5–15, 2007.

[12] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *USENIX annual technical conference, FREENIX Track*, volume 41, pages 10–5555. California, USA, 2005.

[13] Tim Blazytko, Moritz Schlögel, Cornelius Aschermann, Ali Abbasi, Joel Frank, Simon Wörner, and Thorsten Holz. Aurora: Statistical crash analysis for automated root cause explanation. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 235–252, 2020.

[14] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):1–49, 2012.

[15] Weidong Cui, Marcus Peinado, Sang Kil Cha, Yanick Fratantonio, and Vasileios P Kemerlis. Retracer: Triaging crashes by reverse execution from partial memory dumps. In *Proceedings of the 38th International Conference on Software Engineering*, pages 820–831, 2016.

[16] Higor A de Souza, Marcos L Chaim, and Fabio Kon. Spectrum-based software fault localization: A survey of techniques, advances, and challenges. *arXiv preprint arXiv:1607.04347*, 2016.

[17] Yunlan Du, Zhenyu Ning, Jun Xu, Zhilong Wang, Yueh-Hsun Lin, Fengwei Zhang, Xinyu Xing, and Bing Mao. Hart: Hardware-assisted kernel module tracing on arm. In *Computer Security–ESORICS 2020: 25th European Symposium on Research in Computer Security, ESORICS 2020, Guildford, UK, September 14–18, 2020, Proceedings, Part I 25*, pages 316–337. Springer, 2020.

[18] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. Deepvsa: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 1787–1804, 2019.

[19] Liang He, Hong Hu, Purui Su, Yan Cai, and Zhenkai Liang. Freewill: Automatically diagnosing use-after-free bugs via reference miscounting detection on binaries. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2497–2512, 2022.

[20] Dae R Jeong, Minkyu Jung, Yoochan Lee, Byoungyoung Lee, Insik Shin, and Youngjin Kwon. Diagnosing kernel concurrency failures with aitia. In *Proceedings of the Eighteenth European Conference on Computer Systems*, pages 94–110, 2023.

[21] James A Jones, Mary Jean Harrold, and John Stasko. Visualization of test information to assist fault localization. In *Proceedings of the 24th international conference on Software engineering*, pages 467–477, 2002.

[22] Min Gyung Kang, Stephen McCamant, Pongsin Poosankam, and Dawn Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.

[23] Baris Kasikci, Weidong Cui, Xinyang Ge, and Ben Niu. Lazy diagnosis of in-production concurrency bugs. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 582–598, 2017.

[24] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *Proceedings of the 25th Symposium on Operating Systems Principles*, pages 344–360, 2015.

[25] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In *NDSS*, 2020.

[26] Thomas D LaToza, Gina Venolia, and Robert DeLine. Maintaining mental models: a study of developer work habits. In *Proceedings of the 28th international conference on Software engineering*, pages 492–501, 2006.

[27] Xiangyu Li and Alessandro Orso. More accurate dynamic slicing for better supporting software debugging. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*, pages 28–38. IEEE, 2020.

[28] Xingwei Li, Yan Kang, Chenggang Wu, Danjun Liu, Jiming Wang, Yue Sun, Zehui Wu, Yunchao Wang, Rongkuan Ma, and Qiang Wei. Yesterday once more: Facilitating linux kernel bug reproduction via reverse fuzzing. *IEEE Transactions on Information Forensics and Security*, 2025.

[29] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. Scalable statistical bug isolation. *Acm Sigplan Notices*, 40(6):15–26, 2005.

[30] Chao Liu, Xifeng Yan, Long Fei, Jiawei Han, and Samuel P Midkiff. Sober: statistical model-based bug localization. *ACM SIGSOFT Software Engineering Notes*, 30(5):286–295, 2005.

[31] Henry B Mann and Donald R Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The annals of mathematical statistics*, pages 50–60, 1947.

[32] Dongliang Mu, Yunlan Du, Jianhao Xu, Jun Xu, Xinyu Xing, Bing Mao, and Peng Liu. Pomp++: Facilitating postmortem program diagnosis with value-set analysis. *IEEE Transactions on Software Engineering*, 47(9):1929–1942, 2019.

[33] Younggi Park, Hwiwon Lee, Jinho Jung, Hyungjoon Koo, and Huy Kang Kim. Benzene: A practical root cause analysis system with an under-constrained state mutation. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 74–74. IEEE Computer Society, 2023.

[34] Andrew Quinn, Jason Flinn, Michael Cafarella, and Baris Kasikci. Debugging the omnitable way. In *16th USENIX Symposium on Operating Systems Design and Implementation (OSDI 22)*, pages 357–373, 2022.

[35] Sergej Schumilo, Cornelius Aschermann, Robert Gawlik, Sebastian Schinzel, and Thorsten Holz. kafl: Hardware-assisted feedback fuzzing for os kernels. In *26th USENIX security symposium (USENIX Security 17)*, pages 167–182, 2017.

[36] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 537–549, 2021.

[37] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. Syzdirect: Directed greybox fuzzing for linux kernel. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pages 1630–1644, 2023.

[38] Daimeng Wang, Zheng Zhang, Hang Zhang, Zhiyun Qian, Srikanth V Krishnamurthy, and Nael Abu-Ghazaleh. Syzvegas: Beating kernel fuzzing odds with reinforcement learning. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2741–2758, 2021.

[39] W Eric Wong, Yu Qi, Lei Zhao, and Kai-Yuan Cai. Effective fault localization using code coverage. In *31st Annual International Computer Software and Applications Conference (COMPSAC 2007)*, volume 1, pages 449–456. IEEE, 2007.

[40] Xiaoyuan Xie, Tsong Yueh Chen, and Baowen Xu. Isolating suspiciousness from spectrum-based fault localization techniques. In *2010 10th International Conference on Quality Software*, pages 385–392. IEEE, 2010.

[41] Dandan Xu, Di Tang, Yi Chen, XiaoFeng Wang, Kai Chen, Haixu Tang, and Longxing Li. Racing on the negative force: Efficient vulnerability {Root-Cause} analysis through reinforcement learning on counterexamples. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4229–4246, 2024.

[42] Jun Xu, Dongliang Mu, Ping Chen, Xinyu Xing, Pei Wang, and Peng Liu. Credal: Towards locating a memory corruption vulnerability with your core dump. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 529–540, 2016.

[43] Jun Xu, Dongliang Mu, Xinyu Xing, Peng Liu, Ping Chen, and Bing Mao. Postmortem program analysis with hardware-enhanced post-crash artifacts. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 17–32, 2017.

[44] Carter Yagemann, Matthew Pruett, Simon P Chung, Kennon Bittick, Brendan Saltaformaggio, and Wenke Lee. Arcus: Symbolic root cause analysis of exploits in production systems. In *USENIX Security Symposium*, pages 1989–2006, 2021.

[45] Xiangyu Zhang, Rajiv Gupta, and Youtao Zhang. Precise dynamic slicing algorithms. In *25th International Conference on Software Engineering, 2003. Proceedings.*, pages 319–329. IEEE, 2003.

[46] Xing Zhang, Jiongyi Chen, Chao Feng, Ruilin Li, Wenrui Diao, Kehuan Zhang, Jing Lei, and Chaojing Tang. Default: mutual information-based crash triage for massive crashes. In *Proceedings of the 44th International Conference on Software Engineering*, pages 635–646, 2022.

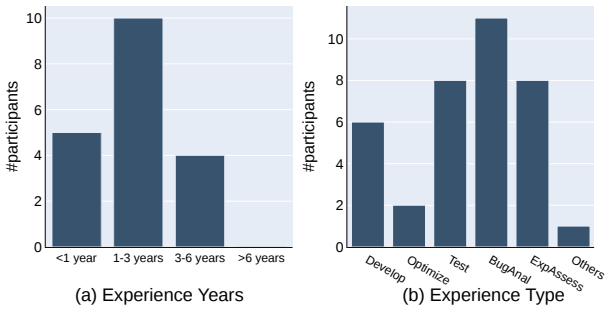
[47] Yiming Zhang, Yuxin Hu, Haonan Li, Wenxuan Shi, Zhenyu Ning, Xiapu Luo, and Fengwei Zhang. Alligator in vest: A practical failure-diagnosis framework via arm hardware features. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 917–928, 2023.

[48] Yongle Zhang, Kirk Rodrigues, Yu Luo, Michael Stumm, and Ding Yuan. The inflection point hypothesis: a principled debugging approach for locating the root cause of a failure. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 131–146, 2019.

[49] Lei Zhao, Keyang Jiang, Yuncong Zhu, Lina Wang, and Jiang Ming. Capturing invalid input manipulations for memory corruption diagnosis. *IEEE Transactions on Dependable and Secure Computing*, 20(2):917–930, 2022.

A Participant Characteristics in User Study

See Figure 8.

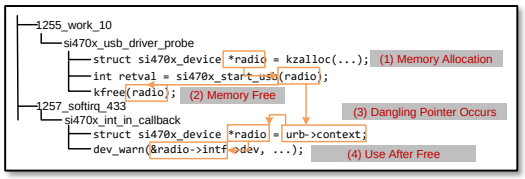


(a) Experience Years (b) Experience Type

Figure 8: Characteristics of Participants

```
1  /* Work Handler */
2  static int si470x_usb_driver_probe(...)
3  {
4      struct si470x_device *radio = kzalloc(...);
5      int retval = si470x_start_usb(radio);
6  -   if (retval < 0)
7  +   if (retval < 0 && !radio->int_in_running)
8          goto err_buf;
9  err_buf:
10     kfree(radio);
11 }
12
13 /* Softirq Handler */
14 static void si470x_int_in_callback(...)
15 {
16     struct si470x_device *radio = urb->context;
17     dev_warn(&radio->intf->dev, ...);
```

(a) Source Code



(b) Contextual Causality Chain (Simplified)

Figure 9: Case Study of Syzbot#94ed.

B Case Study

B.1 94ed - A UAF Involving Multiple Tasks

Besides the motivating example, we showcase another UAF bug to demonstrate how the CC-chain would facilitate bug understanding and fixing. The bug `94ed` is a use-after-free bug that involves multiple indirectly emitted tasks. The source code is shown in Figure 9(a). The PoC first launches a queued work to create a USB device at line 4, then saves the pointer `radio` in a complex data structure at `si470x_start_usb`. Next, the function `si470x_start_usb` fails and `radio` is freed. However, it forgets to remove the pointer from the data structure. Later, a softirq handler obtains the dangling pointer `radio` and accesses a freed object.

To understand and fix the bug, the analyst needs to trace how the dangling pointer propagates. However, the call stacks of the free and UAF sites in the KASAN report have no overlap, forcing analysts to manually examine a large codebase to reconstruct the data dependency. It is quite challenging since the dangling pointer is read in a softirq handler but was written in a seemingly unrelated work handler. Instead, CC-chain(Figure 9(b)) intuitively depicts the whole procedure, including this cross-task data dependency. With the chain, the bug-fixing approach becomes intuitive. The use-after-free can be prevented by clearing the dangling pointer after the `si470x_start_usb` returns, but `radio` has been registered. The patch in Figure 9(a) fixes the issue exactly in this manner.

C Keywords of Dataset Selection

See Table 5.

Table 5: Keywords and Shorthands of Crash Types

Keyword	Crash Type
general protection fault	GPF
Bug: NULL pointer dereference	NPD
unable to handle kernel paging request	UPR
KASAN: use-after-free	UAF
KASAN: slab-out-of-bounds	OOB
KASAN: null-ptr-deref	KNP

D Effectivenss of Selective Tracing

See Table 6.

Table 6: Effectiveness of Selective Tracing. "#Inst."=number of instructions. "ratio"=percentage of the instructions compared to the raw.

Bug ID	Raw	Task		Norm		Bug ID	Raw	Task		Norm	
	#Inst.	#Inst.	Ratio	#Inst.	#Ratio		#Inst.	#Inst.	Ratio	#Inst.	#Ratio
02ddd83	137,265K	33,203K	0.24	5,266K	0.04	12c0ee3	1,237K	693K	0.56	26K	0.02
15bf7d9	49,558K	18,398K	0.37	3,889K	0.08	16a7f16	31,942K	14,255K	0.45	2,326K	0.07
1708e7a	53,065K	22,662K	0.43	3,420K	0.06	1bc76a5	1,193K	689K	0.58	36K	0.03
1f0cb39	52,149K	17,398K	0.33	2,492K	0.05	1fbcb60	3,244K	1,254K	0.39	194K	0.06
2364bdc	1,378K	1,033K	0.75	52K	0.04	24a275e	59,005K	13,306K	0.23	1,227K	0.02
26567b1	3,002K	2,020K	0.67	717K	0.24	2c91a9e	1,474K	1,031K	0.70	66K	0.05
2e1c269	5,805K	2,489K	0.43	241K	0.04	3674e1e	1,331K	1,122K	0.84	74K	0.06
396fb46	4,238K	3,046K	0.72	298K	0.07	3aacade	5,674K	2,642K	0.47	221K	0.04
3d67d69	20,351K	6,994K	0.34	1,259K	0.06	3f6daa8	52,747K	11,283K	0.21	1,390K	0.03
4187226	826K	655K	0.79	53K	0.06	449f561	1,024K	1,005K	0.98	49K	0.05
4b1e841	1,269K	792K	0.62	129K	0.10	578ca23	5,974K	2,854K	0.48	521K	0.09
57c3bc1	1,961K	1,503K	0.77	92K	0.05	59997d8	770K	564K	0.73	43K	0.06
5ba5b3f	3,000K	1,723K	0.57	140K	0.05	6312526	153,257K	51,772K	0.34	6,231K	0.04
691c051	2,906K	2,219K	0.76	274K	0.09	6922c8c	1,068K	795K	0.74	42K	0.04
694d503	5,939K	2,625K	0.44	333K	0.06	6c6ad1a	1,338K	1,053K	0.79	53K	0.04
8c7784a	14,448K	14,149K	0.98	2,415K	0.17	8dba39e	106,473K	4,983K	0.05	576K	0.01
8e599aa	2,168K	1,666K	0.77	223K	0.10	91f769e	6,202K	2,307K	0.37	316K	0.05
94ed6dd	129,012K	35,265K	0.27	4,521K	0.04	990d1ea	914K	625K	0.68	48K	0.05
a0f577f	10,462K	6,162K	0.59	562K	0.05	a13951b	138,096K	45,634K	0.33	10,435K	0.08
a8e52ae	1,881K	1,129K	0.60	168K	0.09	ae313e1	1,118K	750K	0.67	43K	0.04
afa337a	4,368K	2,517K	0.58	420K	0.10	b44aa2c	1,044K	988K	0.95	56K	0.05
b66d8de	4,676K	3,238K	0.69	497K	0.11	c50f1f1	5,287K	1,955K	0.37	417K	0.08
c5c6967	1,483K	1,350K	0.91	88K	0.06	d35e6e8	22,033K	9,748K	0.44	1,855K	0.08
d5cd7bc	681K	474K	0.70	21K	0.03	f1d7f87	3,468K	1,511K	0.44	127K	0.04
f451140	4,476K	2,385K	0.53	425K	0.10	f56bbe6	830K	602K	0.73	63K	0.08
f9cfa5c	2,307K	1,657K	0.72	193K	0.08	defb47b	2,646K	1,962K	0.74	741K	0.28
7b96055	1,666K	1,375K	0.83	142K	0.09	c580fc5	1,029K	676K	0.66	59K	0.06
Average	20,940K	6,744K	0.32	1,029K	0.05						

E Effectiveness of Contextual Information for RCA

See Table 7.

Table 7: Effectiveness of Contextual Information for RCA. #Dep.Inst=number of instructions having data dependency with the crash site. #BC.Inst=number of bug-contributing instructions in the final report

Bug ID	#Dep.Inst	#BC.Inst.	Ratio	Bug ID	#Dep.Inst.	#BC.Inst	Ratio
02ddd83	63,202	35	0.0006	12c0ee3	333	6	0.0180
15bf7d9	15,997	14	0.0009	16a7f16	8,796	1	0.0001
1708e7a	33,591	13	0.0004	1bc76a5	442	8	0.0181
1f0cb39	13,069	6	0.0005	1fbcb60	449	6	0.0134
2364bdc	404	3	0.0074	24a275e	8,360	78	0.0093
26567b1	814	8	0.0098	2c91a9e	83	9	0.1084
2e1c269	753	9	0.0120	3674e1e	1,426	6	0.0042
396fb46	6,566	3	0.0005	3aacade	1,603	1	0.0006
3d67d69	1,990	11	0.0055	3f6daa8	4,431	6	0.0014
4187226	229	8	0.0349	449f561	32	2	0.0625
4b1e841	13,480	2	0.0001	578ca23	6,109	6	0.0010
57c3bc1	1,097	4	0.0036	59997d8	47	2	0.0426
5ba5b3f	1,778	10	0.0056	6312526	28,583	12	0.0004
691c051	825	5	0.0061	6922c8c	122	8	0.0656
694d503	1,151	1	0.0009	6c6ad1a	17	6	0.3529
8c7784a	6,686	11	0.0016	8dba39e	502	7	0.0139
8e599aa	604	7	0.0116	91f769e	1,408	6	0.0043
94ed6dd	36,318	8	0.0002	990d1ea	48	4	0.0833
a0f577f	3,183	8	0.0025	a13951b	28,136	10	0.0004
a8e52ae	1,921	5	0.0026	ae313e1	16	6	0.3750
afa337a	2,874	5	0.0017	b44aa2c	545	5	0.0092
b66d8de	3,337	5	0.0015	c50f1f1	2,320	4	0.0017
c5c6967	1,451	9	0.0062	d35e6e8	13,968	74	0.0053
d5cd7bc	2	2	1.0000	f1d7f87	113	5	0.0442
f451140	1,218	17	0.0140	f56bbe6	68	5	0.0735
f9cfa5c	4,674	3	0.0006	defb47b	751	8	0.0107
7b96055	2,789	7	0.0025	c580fc5	10	3	0.3000
Average	6,087	9.5	0.0016				

F Overhead of KernelRCA

See Table 8.

G Dataset List

See Table 9.

Table 8: Overhead of Root Cause Analysis.

Bug ID	Trace Size (GB)	Time (s)				
		Total	Tracing	Norm	Context	RCA
02ddd83	6.14	299	171	36	67	25
12c0ee3	0.13	57	42	0	1	14
15bf7d9	3.29	217	104	22	76	15
16a7f16	3.11	147	79	23	30	15
1708e7a	4.92	187	97	27	42	21
1bc76a5	0.13	63	41	0	1	21
1f0cb39	3.75	170	103	22	30	15
1fbcb60	0.23	63	44	1	3	15
2364bdc	0.19	58	42	1	1	14
24a275e	2.92	154	96	19	17	22
26567b1	0.37	73	42	3	8	20
2c91a9e	0.19	60	42	1	2	15
2e1c269	0.53	84	63	2	4	15
3674e1e	0.21	60	42	1	2	15
396fb46	0.57	73	46	3	5	19
3aacade	0.57	78	58	3	4	13
3d67d69	1.50	103	63	9	17	14
3f6daa8	2.46	131	83	14	19	15
4187226	0.12	59	43	1	2	13
449f561	0.19	59	42	1	1	15
4b1e841	0.14	59	40	1	3	15
578ca23	0.52	67	42	3	7	15
57c3bc1	0.29	65	43	1	2	19
59997d8	0.10	59	44	1	1	13
5ba5b3f	0.32	64	41	2	3	18
6312526	11.50	319	150	65	80	24
691c051	0.40	62	41	2	4	15
6922c8c	0.14	57	40	1	2	14
694d503	0.56	83	61	3	4	15
6c6ad1a	0.19	59	42	1	2	14
8c7784a	2.61	128	62	14	30	22
8dba39e	1.07	121	91	7	8	15
8e599aa	0.30	62	41	2	3	16
91f769e	0.42	69	41	3	5	20
94ed6dd	7.76	239	114	43	59	23
990d1ea	0.11	59	43	1	1	14
a0f577f	1.14	89	55	7	8	19
a13951b	9.92	380	150	79	135	16
a8e52ae	0.20	63	43	2	2	16
ae313e1	0.14	55	39	1	1	14
afa337a	0.46	69	41	3	6	19
b44aa2c	0.18	62	45	1	1	15
b66d8de	0.61	72	43	3	7	19
c50f1f1	0.35	61	41	2	5	13
c5c6967	0.25	59	42	1	1	15
d35e6e8	1.75	106	56	12	23	15
d5cd7bc	0.09	54	40	0	1	13
f1d7f87	0.32	78	59	2	2	15
f451140	0.43	69	46	2	6	15
f56bbe6	0.11	58	42	0	2	14
f9cfa5c	0.30	60	42	1	4	13
defb47b	0.36	72	42	3	8	19
7b96055	0.25	61	43	1	3	14
c580fc5	0.12	56	41	1	1	13
Average	1.39	97	59	8	14	16

Table 9: Bug List in Dataset.

Crash ID	Version	Commit	Crash Title	LoC of PoC	LoC of Patch
02ddd838	5.10.0-rc3	e28c0d7c	general protection fault in wext_handle_ioctl	509	5
12c0ee39	5.6.0-rc2	ca7e1fd1	BUG: unable to handle kernel NULL pointer dereference in cipso_v4_sock_setattr	29	6
15bf7d92	5.9.0-rc8	549738f1	general protection fault in gfs2_withdraw	7176	15
16a7f16e	5.9.0-rc6	ba4f184e	KASAN: slab-out-of-bounds Read in squashfs_get_id	202	8
1708e7a5	5.4.0-	63de3747	KASAN: slab-out-of-bounds Write in decode_data	345	6
1bc76a57	5.14.0	835d31d3	KASAN: null-ptr-deref Write in __pm_runtime_resume	33	8
1f0cb396	5.7.0-rc1	50cc09c1	KASAN: slab-out-of-bounds Read in skb_gso_transport_seglen	255	26
1fbcb607	5.6.0-rc3	f8788d86	general protection fault in nldev_stat_set_doit	80	2
2364bdc6	5.5.0-rc5	ae608821	general protection fault in hash_ipportnet4_uadt	177	3
24a275e9	5.11.0-rc1	5c8fe583	KASAN: slab-out-of-bounds Read in squashfs_export_iget	211	41
26567b12	5.11.0-rc3	0da0a8a0	BUG: unable to handle kernel NULL pointer dereference in fbcon_cursor	40	3
2c91a9eb	5.5.0-rc5	e69ec487	general protection fault in tcf_ife_cleanup	41	7
2e1c2693	5.7.0-rc5	152036d1	KASAN: slab-out-of-bounds Read in fl6_update_dst	63	30
3674e1e1	5.5.0-rc6	51d69817	general protection fault in nft_tunnel_get_init	158	2
36b975e3	5.0.0	ebc551f2	general protection fault in ebitmap_destroy (2)	34	13
37522d15	5.9.0-rc6	805c6d3c	general protection fault in madvise_cold_or_pageout_pte_range	180	2
396fb46b	5.13.0	dbe69e43	general protection fault in blk_mq_run_hw_queues	220	2
3aacade3	5.2.0	c6dd78fc	KASAN: slab-out-of-bounds Read in do_jit	84	9
3d67d693	5.5.0-rc5	e69ec487	KASAN: slab-out-of-bounds Write in mpol_parse_str	159	6
3f6daa8d	5.9.0-rc6	171d4ff7	KASAN: slab-out-of-bounds Read in f2fs_build_segment_manager	241	6
41872265	5.0.0-rc3	787a3b43	general protection fault in rxrpc_connect_call	43	4
449f5619	5.5.0	b3a60822	general protection fault in nf_flow_table_offload_setup	141	6
48de800a	5.5.0-rc6	f5ae2ea6	general protection fault in nft_chain_parse_hook	109	29
4b1e8410	5.8.0	fb893de3	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post (2)	33	2
517fa734	5.14.0	78e70952	KASAN: use-after-free Write in null_skcipher_crypt	166	12
53c05996	5.14.0-rc7	77dd1143	general protection fault in legacy_parse_param	38	22
578ca237	5.10.0-rc6	e87297fa	BUG: unable to handle kernel paging request in dqput	243	19
57c3bc1b	5.14.0	b91db6a0	general protection fault in __io_file_supports_nowait	175	13
59997d82	5.2.0-rc5	bed3c0d8	general protection fault in btf_struct_resolve	55	12
5ba5b3fc	5.10.0-rc2	4ef8451b	general protection fault in io_uring_show_cred	74	3
5c9918d2	5.8.0	2cc3c4b3	general protection fault in io_poll_double_wake	219	34
6312526a	5.14.0-rc1	7fef2edf	KASAN: null-ptr-deref Read in filp_close (2)	186	23
691c051d	5.10.0-rc6	34816d20	BUG: unable to handle kernel paging request in diFree	191	3
6922c8c3	5.9.0-rc8	3dd0130f	general protection fault in qp_release_pages	44	10
694d503a	5.2.0	c6dd78fc	KASAN: slab-out-of-bounds Read in bpf_int_jit_compile	84	9
6c6ad1a5	5.4.0-rc1	b145b0eb	general protection fault in dsmark_init	75	2
6d28e246	5.10.0-rc7	a68a0262	BUG: unable to handle kernel NULL pointer dereference in __lookup_slow	220	6
7b960555	5.6.0-rc5	2363d73a	general protection fault in ethnl_parse_header	131	4
8c7784ae	5.13.0	3dbdb38e	general protection fault in try_grab_compound_head	131	5
8dba39ec	5.5.0-rc5	e69ec487	general protection fault in xt_rateest_put	652	19
8e599aa9	5.9.0-rc5	325d0eab	general protection fault in jffs2_parse_param	173	10
91f769e6	5.11.0	a99163e9	general protection fault in kvm_hv_irq_routing_update	45	2
94ed6ddd	5.11.0	f40ddce8	KASAN: use-after-free Read in si470x_int_in_callback (2)	765	4
990d1ea1	5.5.0-rc3	46cf053e	general protection fault in xt_rateest_tg_checkentry	852	27
9f4513b4	5.6.0-rc3	f8788d86	general protection fault in smc_ib_remove_dev	100	2
a0f577f4	5.11.0-rc3	7c53f6b6	general protection fault in io_disable_sqo_submit	271	3
a13951ba	5.8.0	fb893de3	KASAN: use-after-free Read in path_init (2)	50	2
a8e52aea	5.7.0-rc2	c578ddb3	general protection fault in fq_codel_enqueue	132	26
ae313e19	5.2.0	3bfe1fc4	general protection fault in tcf_ife_init	40	5
afa337a1	5.11.0-rc2	36bbbd0e	BUG: unable to handle kernel paging request in percpu_ref_exit	114	12
b44aa2c3	5.5.0-rc5	ec7b3f53	general protection fault in dccp_timeout_nlattr_to_obj	320	6
b53aed2f	5.5.0-rc7	d96d875e	KASAN: slab-out-of-bounds Write in setup_udp_tunnel_sock (2)	54	10
b66d8de2	5.10.0	d635a69d	general protection fault in j1939_netdev_notify (2)	38	150
c4bd8ccf	5.13.0	3dbdb38e	BUG: unable to handle kernel paging request in vga16fb_fillrect	66	6
c50f1f10	5.0.0-rc2	7fbfee7c	general protection fault in fuse_dev_do_write	62	2
c5c6967d	5.9.0-rc1	c3d8f220	general protection fault in fib_dump_info (2)	109	5
c580fc53	5.5.0-rc5	59c820b2	general protection fault in ip6_datagram_dst_update	69	2
d35e6e87	5.7.0-rc1	50cc09c1	KASAN: null-ptr-deref Write in choke_reset	242	3
d5cd7bc7	5.2.0-rc5	bed3c0d8	general protection fault in btf_array_resolve	48	12
defb47bf	5.11.0	5695e516	BUG: unable to handle kernel NULL pointer dereference in hide_cursor	39	3
f1d7f87f	5.8.0-rc2	1590a2e1	KASAN: slab-out-of-bounds Read in qrtr_endpoint_post	32	6
f451140b	5.7.0	7ae77150	general protection fault in __apic_accept_irq (2)	47	3
f56bbe66	5.7.0	7ae77150	general protection fault in qrtr_endpoint_post	29	6
f82ab894	5.5.0-rc7	4703d911	general protection fault in do_con_write	355	29
f9cfa5c5	5.0.0	63bdf428	general protection fault in sctp_sched_rr_dequeue	51	2