DeepErr: Automatic Root-Cause Analysis of System Call Failures

Nadav Amit
Technion - Israel Institute of Technology
Haifa, Israel
namit@technion.ac.il

Michael Wei
VMware Research Group
Palo Alto, CA, USA
michael@wei.email

Abstract

System call failures present significant challenges for operating system (OS) users, as the failures are often cryptic and difficult to diagnose due to limited error codes and missing documentation. As a result, software developers struggle to utilize system calls effectively, and power users encounter difficulties configuring the OS and resolving environment problems. Existing automatic root-cause analysis tools are inadequate, primarily due to dependence on comparative analysis, which requires similar successful executions that are often unavailable.

In this paper, we present a tracing and root-cause analysis solution to address these limitations. We enable comparative analysis by using symbolic execution to generate analogous successful executions in the absence of actual ones. Furthermore, to address the limited availability and shortcomings of hardware-based control-flow tracing, we propose probe-point based tracing of the entire control flow. Utilizing these techniques, we develop DeepErr, a system call analyzer that identifies the precise predicate responsible for failures. DeepErr's effectiveness is affirmed through application on 100 tests from the Linux Test Project, successfully pinpointing root causes in 91% of the scenarios and identifying the failing function in an additional 7% of cases.

CCS Concepts: • **Software and its engineering → Operating systems**; *Software testing and debugging*; Functionality.

Keywords: System call failures, root-cause analysis, error diagnosis, operating systems, debugging

1 Introduction

In software development, understanding how to use libraries or services can be a significant challenge, particularly when the interfaces and APIs are poorly defined [41]. This issue often leads to misunderstandings and bugs that are difficult to diagnose, as developers may lack familiarity with the underlying code. Error codes, which are commonly used to convey failure information, lack elaborate explanations, making it challenging to understand and resolve issues [20].

```
$ echo 2 > /sys/fs/cgroup/c/g/cgroup.procs
bash: echo: write error: Invalid argument
```

Listing 1. Example of a system call failure when configuring a cgroup in Linux. The system call, indirectly invoked through the 'echo' command, results in a non-informative error code that propagates back to the user.

This problem is particularly prevalent in operating systems, where software developers frequently struggle to understand why system call invocations fail. The complexity of the OS, combined with the privileged mode in which the code runs, complicates debugging. Furthermore, manual pages are often outdated or incomplete (e.g., [25]), and OSes tend to overload error codes, using a limited set of codes to represent a wide range of failures [34]. As a result, developers find it difficult to determine the root cause of system call failures.

The significance of this problem was recently highlighted at the Linux Storage, Filesystem, Memory Management, and BPF Summit (LSFMM) [11]. Both OS developers and users expressed frustration with the lack of detailed error messages and requested indications of where error codes are generated, with some suggesting that a call stack would be more informative. Field studies have established the importance of providing comprehensive documentation for low-level details such as error codes [22], highlighting the need for improved error reporting mechanisms in operating systems.

Understanding system call failures is essential not only for developers but also for IT professionals and power users when troubleshooting application issues or configuring the OS. These users often identify the root cause of such issues by monitoring failing system calls and their return error code [5]. However, the error code itself is often insufficient. For example, when moving a process into a Linux cgroup by writing to files in the `/sys` filesystem, the operation may fail without providing a meaningful reason, as shown in Listing 1, making diagnosis difficult.

The limitations of the system error code mechanism are well-known among software developers, but current solutions are still quite limited and often require substantial manual work. Engineers from Google, Red-Hat, and Meta have proposed various solutions, but these typically require significant changes to the OS [4], or, while they do simplify

SYSTOR '25, Virtual, Israel

ACM ISBN 979-8-4007-2119-9/25/09
https://doi.org/10.1145/3757347.3759134

```
/*
 * kthreads may acquire PF_NO_SETAFFINITY during
 * initialization. If userland migrates such a
 * kthread to a non-root cgroup, it can become
 * trapped in a cpuset, or RT kthread may be born
 * in a cgroup with no rt_runtime allocated.
 * Just say no.
 */
if (tsk->no_cgroup_migration ||
    (tsk->flags & PF_NO_SETAFFINITY)) {
    tsk = ERR_PTR(-EINVAL);
    goto out_unlock_threadgroup;
}
```

Listing 2. DeepErr analysis result for the failure in Listing 1, indicating the highlighted code as the root cause.

tracing to a certain extent, they still require manual analysis of the root cause [13, 21].

Automated root-cause analysis tools exist for application-level debugging [7, 15, 17], but they often come with limitations. Finding the root cause essentially requires comparing the failing execution against similar successful executions to identify the point where successful and failing executions with the longest common prefix diverge. However, in the case of a failing system call, a similar successful execution is often not available. In addition, some tools require memory snapshots for binary instrumentation, a prerequisite that poses significant challenges when dealing with OSes in production environments.

Similarly, reverse debugging methods in OSes, which can help trace errors to their origins, still require substantial manual effort and are often impractical to deploy in production settings. Techniques like QEMU record-replay involve recording interactions with virtual hardware and taking periodic snapshots, resulting in significant overhead [32]. Similarly, the hardware-based REPT approach collects data upon failure to recreate the system state, a process that can be costly and sometimes infeasible [9, 12]. Despite their usefulness in debugging by experts, they are unsuitable for system call failure analysis.

DeepErr is a system tailored to address these challenges, focusing specifically on the automatic identification of root causes for system call failures. It operates on the insight that a system call failure, signaled by the return of an error code, usually occurs shortly after the root cause of the failure is detected. This behavior allows DeepErr to employ symbolic execution [6] to create the most similar successful execution of the system call. By discerning the initial point of divergence between the observed failed execution and this potential successful execution, DeepErr can accurately pinpoint the root cause.

For example, the failure in Listing 1 can be easily recorded and analyzed using DeepErr by directing it to analyze failures of the `write` system call. DeepErr identifies the conditional branch or predicated instruction that was found to be the root cause of the failure, which is often not the last conditional branch in the function due to cleanup code (e.g., unlocking as done in the example) or code inlining. By leveraging debug information, DeepErr then pinpoints the exact predicate in the source code responsible for the error and provides the corresponding call stack. Such information can be used by oridinary software developers to resolve their issues [11].

In the case of the aforementioned failure, DeepErr's analysis indicates that the root cause lies in the code highlighted in Listing 2. Although the comment in the code clearly explains the cause of the failure, even without it, navigating the OS code or examining the commit log message that introduced the failing predicate can help identify the issue relatively easily.

DeepErr can utilize existing hardware branch tracing solutions like Intel PT to record execution paths for analysis. This means that even without any kernel code instrumentation, DeepErr can effectively root-cause over 90% of the system call failures.

Additionally, to overcome the availability and tracking limitations of hardware branch tracing, DeepErr introduces a novel software-based tracing alternative that instruments the kernel code upon failure. This approach leverages the idempotence of most system call failures to reinvoke the call after augmenting the execution path with probes that allows its instrumentation. By doing so, DeepErr collects data overlooked by hardware solutions, such as predicated instruction execution and repeated string instruction invocation counts.

The contributions of this paper are:

- Analysis of challenges in diagnosing root causes of system call errors.
- Introduction of a new approach to root cause analysis of system call failures, featuring:
  - A novel software-based OS execution flow tracing solution.
  - A strategy employing symbolic execution simulation and backtracking to pinpoint the failure root cause.
- Evaluation of DeepErr's effectiveness in identifying root causes of system call failures.

2 Background

Understanding the root cause of system call failures is a significant challenge that often leads users into time-consuming troubleshooting and debugging scenarios. System call failure scenarios manifest in various contexts: developers may struggle to diagnose failures when using less-common syscall features, system administrators encounter cryptic errors when configuring the system through syscalls like `mount` or `write` to pseudo-files, and power users face difficulties pinpointing why applications fail due to system misconfigurations or flawed assumptions about OS behavior. The difficulty of

syscall failure analysis frequently discourages thorough investigation, pushing users towards suboptimal workarounds rather than addressing the underlying issues.

The root of the problem lies in the POSIX-based error reporting mechanism. System calls typically return a single value indicating either success or an error code. These codes are frequently overloaded and reused across different contexts, significantly impeding precise error identification. OS kernels, being intricate systems operating in privileged mode, further complicate debugging and tracing processes. Moreover, documentation is often incomplete or misleading, and OS bugs can lead to unexpected and undocumented error codes. For example, while the `write` syscall may be well-documented for regular files, its usage with pseudo-files can fail for undocumented reasons, as illustrated in Listing 1.

The challenge of diagnosing system call failures has gained considerable attention in the developer community. At the 2024 Linux Storage, Filesystem, Memory Management, and BPF Summit (LSFMM), developers voiced their frustration with the lack of detailed error messages [11]. Miklos Szeredi, a prominent developer, articulated the problem: "There are lots of places in the kernel where an `EINVAL` can be returned to user space, but it is often unclear what the actual underlying problem is because the errno error codes are too generic." This sentiment resonates throughout the community, as evidenced by various attempts to address the issue by engineers from major tech companies [4, 13, 21], and numerous questions about system call errors on online forums [35].

Prior studies argued that to effectively diagnose similar issues, users need to identify the *failure-inducing predicate*, which is the condition that triggered the failure [15, 38, 40] and if negated could otherwise succeed. Users have also expressed the need for the entire call stack of the failing condition. By mapping the failing predicate to the source code, application software developers argue they can leverage kernel sources and in-code documentation to understand and resolve the issue [11].

Existing tools and approaches for addressing system call failures have significant limitations. `strace` can track syscall inputs and outputs but often provides insufficient information for complex error analysis. Proposals to modify the OS for more descriptive error codes have been rejected due to various challenges. These include technical difficulties in propagating multiple values from each function, structural issues in maintaining a stable ABI while introducing new error codes, and the additional maintenance burden placed on OS developers [4, 11]. Kernel debuggers like Drgn [24], Crash Utility [3], and kdb typically lack source-level debugging functionality and are designed primarily for post-mortem analysis. More advanced tools such as kgdb [10] or virtualization-based record-replay [36] require complex setups, limiting their practicality in production environments.

It might appear that existing approaches for userspace application bugs analysis could be applied to system call failures. However, a significant limitation of common root cause analysis tools is their reliance on comparing failing executions with similar successful ones [7, 15, 17]. This approach faces substantial challenges when applied to system calls, as maintaining an extensive collection of successful executions covering the wide range of possible inputs and system states is impractical, especially considering frequent updates and changes in modern OSes.

Even with adequate tracing and debugging tools, providing users with a single concrete root cause remains challenging. Automated failure analysis tools typically provide execution slices [15, 37, 39]—backward slices of the failure trace including only the necessary instructions to reproduce the error. However, these often present an overwhelming amount of information, making it difficult for users to isolate the root cause efficiently. We discuss these tools and their limitations in Section 6.

In practice, system call failure analysis is usually done using built-in OS code instrumentation tools like Linux's ftrace [23] or FreeBSD's dtrace. These tools can be used to collect detailed information about function calls and their return values in order to manually reconstruct and analyze the execution path leading to the failure. The collection of such traces can be somewhat simplified using `retsnoop`, a tool incorporated in some Linux distributions that can trace function return values during syscall failures. However, as such tracing might be expensive, it requires the user to specify which functions to monitor. Consequently, as tracing occurs only at the function level, debugging still requires extensive effort and expertise.

The absence of an effective, user-friendly solution for identifying root causes of system call failures in production environments motivates our work. DeepErr overcomes existing tools' limitations by employing symbolic execution to generate analogous successful executions in the absence of actual ones. For tracing the execution, DeepErr can utilize either hardware tracing support or probe-point based software tracing, offering flexibility across various environments.

3 Design

Our goal is to accurately identify the root cause of a syscall failure. We define the root cause as the last evaluated predicate that, if negated, would have caused the execution to succeed. This aligns with previous definitions in the literature, which describe it as either the condition that triggered the failure [15, 38] or the first instruction where the failing execution deviates from a successful one [40]. To provide comprehensive context, we aim to not only identify this failure-inducing predicate but also provide the call stack at the point of failure, enabling users to better understand and resolve the issue.

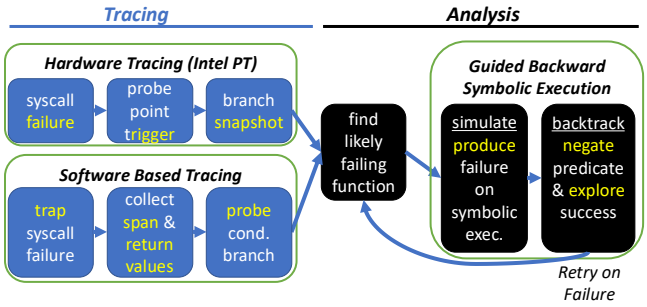


Figure 1. Overview of DeepErr design.

Identifying the root cause of syscall failures presents several challenges that make tracing of error codes insufficient. Compiler optimizations, inlining, and recoverable errors can obscure the true source of failure. Augmenting the trace with hardware branch trace can help, but still requires manual analysis to identify the failing predicate, as subsequent cleanup operations may execute additional conditional branches. As a result, tracing alone is insufficient, and we aim to automate this complex analysis process.

Our approach, depicted in Figure 1, consists of two parts: tracing and analysis. Tracing can be achieved through hardware mechanisms like Intel PT or our new software solution that performs full binary instrumentation of the failing system call execution. Analysis involves identifying the likely root cause function, then using symbolic execution to pinpoint the specific failure-inducing predicate without requiring a successful reference execution.

3.1 Software-Based Tracing Using Probe Points

Probe points[1] are used for dynamic instrumentation of instructions within operating systems for debugging and tracing purposes, and are supported by common systems like Linux and FreeBSD. Compared to hardware-based tracing tools such as Intel PT, which have their specific use cases, probe points offer unique advantages. They are versatile, applicable to any system regardless of hardware capabilities, and can provide more detailed execution traces.

Probe point tracing introduces tension: setting too many degrades performance— both during setup and runtime— while too few prevent root cause analysis. Our approach selectively enables probe points only during failing syscall invocations, targeting just certain instructions in the functions used. Since failure timing is unknown, we first trap failures, then re-invoke the syscall to identify used functions, and finally re-invoke with probe points for detailed tracing. This allows effective tracing without destabilizing the system or introducing high overheads.

In more detail, we use probe points as follows:

1. Failure trapping. DeepErr employs the `ptrace` OS services to monitor syscalls, identifying unsuccessful ones based on their return values. Users have the option to focus only on specific system calls, allowing DeepErr to disregard failures in others. Upon detecting a failure, DeepErr re-invokes the call twice using `ptrace`, resetting registers to their original values and confirming that the return value replicates the initial failure. Detailed steps of these re-invocations are explained in subsequent paragraphs.

2. Span tree generation. DeepErr generates a span tree during the first system call re-invocation to determine the functions to be traced and probed. While static analysis can create a call graph, it might lead to unnecessary probe points. To construct a reliable span tree, DeepErr tracks all executed functions using `ftrace`'s callstack functionality. It initially re-invokes the failing syscall without callstack tracing to compile a list of executed kernel functions. Based on this list, DeepErr refines the traced functions and activates callstack tracing. The resulting trace is then filtered to exclude untraceable or irrelevant functions, such as those that run inside interrupt service routines. The remaining functions in the trace are used to set probe points for detailed analysis.

3. Full trace generation. DeepErr then initiates execution tracing by setting probe points on functions identified during syscall execution. These probe points encompass conditional branch instructions and predicated instructions, such as `CMOVcc` on x86, which cannot be traced by existing hardware tracing mechanisms. It records the values of relevant registers (`RFLAGS` and `RCX` on x86) to ascertain whether the condition was true and the instruction executed. To trace indirect branch instructions, DeepErr sets probe points on instructions following the `ENDBR` instruction used by Intel's Control Enforcement Technology (CET). For instructions executed multiple times, such as those with a `REP` prefix, DeepErr probes the instruction following them and saves relevant registers to determine the number of iterations run. Along with these probe points, DeepErr sets probe points on the return of each traced function to capture its return value, thereby facilitating the identification of the function that precipitated the failure. In cases where OS restrictions prevent tracing a function, DeepErr sets a probe after its return to capture its return value.

3.2 Hardware Tracing

DeepErr also supports hardware tracing using Intel Processor Trace (Intel PT) [16], which, however, comes with several limitations. For instance, Intel PT cannot determine how many times REP-prefixed or predicated instructions have been executed.

[1]In this paper, we use "probe points" to refer to dynamic instrumentation points (e.g., Linux kprobes) that can be placed at runtime on arbitrary instructions in the compiled kernel code. This differs from static "tracepoints" that are built into the kernel at compile time at predetermined locations.

To limit the performance impact, DeepErr employs "snapshot" tracing with Linux `perf`. This approach is set to capture a snapshot of taken branches when a user-generated signal is delivered. Unfortunately, Linux's perf tool lacks an automatic mechanism to capture a branch trace when a specific event—in our case, a system call failure—triggers. With snapshot mode in play, it is critical to detect a system call failure promptly; otherwise, the snapshot risks being overrun. Determining where in the trace the failing syscall event occurred is a complex task, as data is not traced. Additionally, tracing system call failures does not offer much help since Linux does not employ mechanisms that would allow the integration of software events and hardware branches (`PTWRITE` instruction on Intel), and software and hardware tracing record different time source.

DeepErr navigates these limitations by deploying an eBPF program to monitor when a failing syscall exit, warranting analysis, occurs, and communicates it back to DeepErr via a "perf" event. The eBPF program also sends a `SIGSTOP` signal to itself, halting its execution until DeepErr captures the trace and restarts it. When DeepErr receives the event, it sends a special signal to perf to direct it to take a snapshot of the branch trace. The precise location of the failure within the trace is then determined based on custom branches that the eBPF program makes for this purpose.

Limitations and advantages. Probe point tracing and hardware-based tracing (e.g., Intel PT) each have distinct advantages and disadvantages. Probe point tracing faces certain limitations such as higher complexity and performance overhead. Its effectiveness is limited in scenarios where system call failures are not consistently reproducible, such as with `EAGAIN` or `EBUSY` errors, or failures influenced by race conditions in multithreaded applications. As noted by Linus Torvalds, while syscalls that return errors should ideally leave no side effects, in practice several errors violate this rule [33]. However, our experience shows that most syscall failures reoccur on re-invocation, and that the side-effects on invocation, if any, do not impact the failure.

Another limitation of probe point tracing is that OS constraints prevent probe point use on altered code (like alternative instructions or para-virtualization) and specific instructions (`UD2`), as well as indirect branches due to Spectre mitigations. These limitations require DeepErr to set probe points on adjacent instructions as a workaround.

Nevertheless, probe points offer more precise tracing than hardware-based methods, enhancing robustness in analysis. They can track conditions and predicated assignments untraceable by hardware methods and monitor return values from each function, aiding in accurately pinpointing the responsible function. Software-based tracing also allows for unlimited trace length, useful for complex systems, and does not require special hardware.

On the other hand, hardware-based tracing with Intel PT can trace the entire code, being unrestricted by instrumentation limits. It has lower overhead and full code coverage, making it more suitable for production systems. However, it operates in snapshot mode with perf, suiting only single failure tracing, and does not trace predicated assignments, which may lead to more analysis failures.

The choice between these methods depends on specific requirements and constraints. Hardware tracing may fail more in simulation and backtracking, while software tracing allows for shorter simulations. Both enable the essential backtracking for DeepErr's root-cause analysis.

3.3 Analysis

Analysis is performed in two stages: *simulation*, which attempts to recreate the failure using symbolic execution and collect all the possible states that did happen on each execution fork, followed by *backtracking* that checks which of these states that did not occur in practice would have eliminated the failure if they had occurred. Importantly, the entire analysis process is fully automated once the trace is recorded—no user intervention is required to guide the symbolic execution or identify successful paths. Finally, using debug information, DeepErr augments the *output* with the corresponding source code and line numbers. We now present the details of each of these stages, followed by an example.

Simulation. To initiate its analysis, DeepErr leverages a symbolic execution engine, simulating the failure based on the trace garnered during the actual error. This engine operates without predefined inputs for system calls or functions. The starting memory image of the OS text and read-only sections are derived from the OS memory (known as `kcore` in Linux), while all remaining inputs, registers, and residual memory are treated as holding unconstrained symbolic values.

Guided by the collected trace, DeepErr simulates the instruction flow precisely as it occurred in the actual execution. When simulation forks into several potential states, DeepErr continues the simulation using the state that matches the trace. States discovered during simulation that did not manifest in the actual execution are collated into a list of "non-taken-states". These states would later be explored during backtracking to identify the root cause of the failure. When the simulation fails due to unconstrained symbolic values, such as following indirect calls, DeepErr imposes necessary constraints on the simulation. Specifically, for indirect calls, it constrains the call target to the actual address observed in the trace, updates the registers, and continues simulation.

Symbolic execution engines typically optimize performance by avoiding state forking for predicated value assignments, such as `CMOVcc` instructions, opting instead for an

```
__fget_light+88                 include/linux/fdtable.h:87:5        files_lookup_fd_raw ()
__fget_light+88                 fs/file.c:1044:10                   __fget_light()
__fdget+14                      fs/file.c:1057:9                    __fdget()
sockfd_lookup_light+18          include/linux/file.h:65:9           fdget()
sockfd_lookup_light+18          net/socket.c:547:16                 sockfd_lookup_light()
__sys_sendmsg+61                net/socket.c:2491:9                 __sys_sendmsg ()      <--
__x64_sys_sendmsg+24            net/socket.c:2504:9                 __do_sys_sendmsg()
__x64_sys_sendmsg+24            net/socket.c:2502:1                 __se_sys_sendmsg()
__x64_sys_sendmsg+24            net/socket.c:2502:1                 __x64_sys_sendmsg()
do_syscall_64+87                arch/x86/entry/common.c:50:14       do_syscall_x64()
do_syscall_64+87                arch/x86/entry/common.c:80:7        do_syscall_64()
entry_SYSCALL_64+149            arch/x86/entry/entry_64.S:118:0     entry_SYSCALL_64_after_hwframe()

83 static inline struct file *files_lookup_fd_raw(struct files_struct *files, unsigned int fd)
84 {
85   struct fdtable *fdt = rcu_dereference_raw(files->fdt);
86
87   if (fd < fdt->max_fds) {        // <<<
88     fd = array_index_nospec(fd, fdt->max_fds);
89     return rcu_dereference_raw(fdt->fd[fd]);
90   }
91   return NULL;
92 }
```

Listing 3. Example of DeepErr output for LTP sendmsg01 test, showing the failing predicate (in `files_lookup_fd_raw`), the first function that returns the error code `__sys_sendmsg` and the rest of the callstack at the time of the failure. Comments from the code that simplify the understanding of the failure were removed for brevity.

"if-then-else" (ite) symbolic value. However, DeepErr, when utilizing software tracing, is able to determine the predicate value from the values recorded in registers. This capability is crucial, as predicated instructions might be the root cause of system call failures. To facilitate the identification of these root causes, DeepErr ensures that the symbolic execution engine forks its state before predicated assignments. It then applies appropriate constraints, based on the operation and the recorded trace, on the simulated state and the "non-taken-state". Importantly, DeepErr's constraints are designed solely to establish the truth value of the predicate, without specifying exact values for the registers.

When using software tracing, as noted earlier, some functions are untraceable. In such scenarios, we instruct the symbolic execution engine to bypass these functions, assigning their return value an uninitialized status. We adopt a similar approach for common functions unlikely to be the root cause of the system call failure, such as lock acquisition functions. Additionally, we use simulation procedures for common functions like memory copying and lock acquiring to accelerate and improve the simulation.

The scope of the simulation––whether it encompasses the whole system call or a part thereof—is vital for both analysis correctness and performance. Software tracing enables focusing on functions likely responsible for the failure, identified by matching their return value with the failure value. Our heuristic sequences function simulations based on their root cause probability, giving precedence to the "last deepest" function in the span tree that returns the error code. This order is established through a reverse post-order traversal, identifying candidate functions that might hold the root cause. DeepErr simulates each function symbolically and, upon completing the simulation, checks if the simulated return value is concrete and equals to the traced error code. A match signifies a successful simulation, prompting DeepErr to proceed with backtracking to pinpoint the error's origin.

Conversely, hardware tracing does not permit such focused simulation. In these cases, we simulate the system call from its initiation. For efficiency, albeit with a potential compromise in correctness, we terminate the simulation once the engine produces a concrete return value matching the encountered error code. This early termination is a strategic decision to handle the extensive scope of hardware tracing scenarios.

Backtracking. To pinpoint the root cause of a failure, DeepErr utilizes a backtracking process. This begins at the end of the trace, navigating through each fork in the symbolic execution. It explores states using the alternatives not taken during the simulation. The goal is to determine whether inverting a specific predicate *could* have led to a successful execution. In Linux, a successful execution returns either a non-negative value or a negative value less than or equal to

-1024, as the kernel reserves the range [-1, -1023] for error codes. DeepErr automatically tests this by symbolically executing the inverted branch path to completion and checking if it produces a non-error return value. This automated determination of successful paths requires no user input—the system leverages Linux's well-defined error code convention to distinguish failures from successful executions. If a state is found where inverting the predicate results in a successful return, that predicate is identified as the root cause of the failure. The process then moves to the previous execution fork.

Efficient backtracking involves crucial trade-offs to balance accuracy and performance. To optimize the process, we omit the simulation of callee functions during backtracking. This omission typically has minimal impact on identifying the root cause but significantly reduces symbolic execution time. We also assume that the root cause is not in functions without return values, using debug information to avoid initiating backtracking from these functions. These optimizations are essential because backtracking explores potentially long paths leading to success, which would be impractical to fully simulate. By making these trade-offs, we maintain the effectiveness of our root cause analysis while substantially improving its efficiency.

Output. DeepErr provides comprehensive output to aid in diagnosing the failure. It provides the failing predicate, the callstack, and both are augmented with debug information to pinpoint the location in the source code where the failure originated. An example of output is depicted in Listing 3.

The failing predicate is identified through the symbolic execution engine and is key to understanding the exact condition that led to the failure. The callstack, on the other hand, provides a hierarchical view of the function calls leading up to the failure, shedding light on the sequence of operations and their context.

Both the failing predicate and the callstack are enriched with debug information using DWARF. This standard provides a mapping between the binary code in the executable and the original source code, allowing DeepErr to translate addresses and binary conditions into human-readable code snippets and line numbers. As a result, users can directly relate the outputs provided by DeepErr to the corresponding parts of their source code.

One of the critical advantages of DeepErr lies in its ability to distinguish between the function containing the failing predicate and the function that eventually returns the error code. The function that houses the failing predicate is typically where the error originates. However, the error might not become apparent until a different function—possibly higher up the callstack—returns an error code. DeepErr clearly presents this difference in its output by highlighting the function that returns the error code in the callstack. This way, it helps users avoid the potential misconception

```
1  static bool
2  locks_conflict(struct file_lock *caller_fl,
3                 struct file_lock *sys_fl)
4  {
5    if (sys_fl->fl_type == F_WRLCK)
6      return true;
7    if (caller_fl->fl_type == F_WRLCK)
8      return true;
9    return false;
10 }
11
12 static int
13 posix_lock_inode(struct inode *inode,
14                  struct file_lock *request,
15                  struct file_lock *conflock)
16 {
17   list_for_each_entry(...) {
18     if (!locks_conflict(request, fl))
19       continue;
20     error = -EAGAIN;
21     if (!(request->fl_flags & FL_SLEEP))
22       goto out;
23     error = -EDEADLK;
24     goto out;
25   }
26   error = 0;
27 out:
28   return error;
29 }
```

Listing 4. Trace Simulation Process: The figure illustrates simplified versions of two Linux kernel functions. Dotted rectangles represent basic blocks. Numbers enclosed in black circles indicate states corresponding to the actual execution trace, whereas numbers enclosed in gray circles indicate states where the simulation diverged from the actual execution trace. After the simulation phase, the states not occurring in the actual execution trace are explored during the backtracking phase, in reverse order (5', 3', 2'), to identify the first state that could have led to a successful return of the function.

that the failure's root cause is necessarily at the top of the callstack, leading to more accurate and efficient debugging.

Root-causing example. To illustrate the process of simulation and backtracking, consider a scenario where the syscall `fcntl` fails with the `EAGAIN` error code, and our probepoints system is used to record the execution trace. Based on the execution trace, the "last deepest" function that returned the error code is determined to be `posix_lock_inode`. Therefore, the simulation process begins by simulating this function, and the symbolic execution of this function is invoked.

Listing 4 illustrates the simulation process. The listing depicts a simplified version of the Linux kernel function

posix_lock_inode, which calls the locks_conflict function as part of the inode locking process. Rectangles represent basic blocks, while the numbers enclosed in black circles represent the states corresponding to the actual execution trace. These numbers are placed adjacent to the basic blocks executed in the corresponding states. To maintain brevity, the simulation is depicted from the point where line 5 of the source code is simulated.

Following each simulation stage (❶–❺), if the simulation forked, the states not occurring in the actual execution (❷′, ❸′, and ❺′), which are depicted with a gray background, are collected for later processing during the backtracking stage. The simulation continues with the states that match the actual execution trace until the function returns.

After the simulation, it is found that the function returned with an error code of EAGAIN, indicating that the simulation succeeded. Therefore, backtracking is initiated. The backtracking processes the "not-taken-states" in the inverse order of the simulation, starting with the last state of ❺′. From these states, all feasible states are explored. However, it is found that none of these states lead to a successful return of the function. Therefore, the backtracking continues to the next state, ❸′. From this state, it is found that there is an execution path that leads to a successful return of the function with a return value of 0. Note that during the exploration, all the information that was collected about the initial state is used, and unfeasible states are discarded. For example, in state ❸′, the state already indicates that sys_fl->fl_type != F_WRLCK.

Once ❸′ is found to be able to lead to a successful return of the function, the backtracking process is stopped. The branch that caused the fork between ❸ and ❸′ is identified as the root-cause of the failure. Using the debug information, DeepErr identifies the predicate in line 7 of the source code as the root-cause of the failure.

The user who receives the output can understand the root cause relatively easily. The comments in the code (not shown) and variable names indicate that the failure is due to a lock conflict caused by an overlapping lock on the file, where the requested lock (the one fcntl executed with) is a write lock.

Limitations. Beyond the constraints of software tracing, our analysis may encounter timeouts or inaccuracies in various scenarios. Multithreaded applications pose particular challenges: race conditions can make executions nondeterministic, meaning failures may not reoccur during our repeated execution phase (required for software tracing). Moreover, our symbolic execution engine models single-threaded execution and does not account for memory modifications by concurrent threads, which can cause simulation failures or incorrect root cause identification. While we attempt to realign the simulation when it deviates from the actual execution, this may lead to reduced precision and backtracking failures, as all states prior to the deviation are discarded. The optimizations we have implemented in the backtracking process, while improving efficiency, may potentially lead to incorrect results if their underlying assumptions are violated. For instance, if the root cause resides in a function without a return value, DeepErr may fail to identify it. Furthermore, if the failing predicate is compiled into a non-branching, non-predicated arithmetic operation, it may elude DeepErr's detection capabilities. These limitations underscore the challenges in achieving comprehensive root cause analysis in complex system environments.

4 Implementation

We implement DeepErr using a 4,300-line python code and 100-line eBPF code (C). DeepErr uses the angr open-source binary analysis platform [26] for analysis.

DeepErr adopts a dual-stage process akin to the usage model of tools like Linux's perf tool, requiring two separate invocations: one for recording and another for analysis. This approach is designed to minimize the analysis overhead on program execution. In the recording stage, DeepErr captures the execution of the failing system call, saving the data trace and, if anticipated for future analysis, storing the live kernel code (extracted from kcore) along with corresponding symbols at the time of the failure. Subsequently, DeepErr is invoked a second time to initiate the analysis stage, enabling offline analysis by loading the previously recorded trace. This stage entails simulating the trace and conducting backtracking to determine the root cause of the error.

The interface allows the user to specify the system call, error code, and number of invocations of the failure they wish to analyze. Linux ptrace is used to trap the system call failure in the software tracing mode. While we have not used seccomp to reduce the number of ptrace traps, it is possible to minimize ptrace overhead using seccomp.

During the development of DeepErr, we discovered and fixed two serious bugs in the Linux kernel that had been present for several years. The first bug caused probe points on certain conditional branch instructions to be emulated incorrectly, and the second bug resulted in a rogue write operation due to a race condition. Additionally, we found a bug in angr's condition evaluation. We fixed all of these bugs and upstreamed the fixes.

In addition, the Linux kernel does not allow for the use of probe points or the tracing of functions marked with the "inline" hint attribute. This restriction was put in place as a defensive measure due to past issues, but it has become limiting and prevents the tracing of functions of interest. Although we and others have proposed patches to address these issues, they have not been integrated yet.

Symbolically executing an OS kernel trace poses considerable challenges. Tools like PyVEX and LLVM, used by angr

Category	Function (%)	Predicate (%)
DeepErr software	98	91
DeepErr hardware	96	90
`retsnoop`	6	0

Table 1. Root cause identification success rates. **Function**: Percentage of cases where the function returning the error code was correctly identified. **Predicate**: Percentage of cases where the precise failure-inducing predicate was correctly identified.

and KLEE, have limited support for privileged code. For example, angr struggles with the accurate simulation of specific registers such as x86 `GDTR`, decoding instructions like `INVLPG` or tracking the interrupt flag. Workarounds like employing concrete execution to circumvent divergent paths [8], adjusting the symbolic state's instruction pointer to match trace values, or simply ignoring these issues, are not feasible for DeepErr. DeepErr requires a solution that ensures fidelity and alignment with the actual execution path, as any deviations could also lead to inaccuracies during backtracking. To address these issues, DeepErr implements hooks for problematic instructions and refines the simulation process.

One mechanism that can potentially be the root cause of failure is the OS "exception table" [18]. This mechanism allows the kernel to handle exceptions, like page faults, occurring within the kernel, and signal back to the faulting code that such an exception has occurred by returning to a different address. This could happen, for instance, if an invalid pointer is provided as a system call argument. DeepErr traces these exceptions (entry and exit to the exception handler) and simulates them during the symbolic execution.

5 Evaluation

Our evaluation of DeepErr aimed to assess its effectiveness in identifying root causes of system call failures, comparing the performance of software-based and hardware-based tracing methods. We also sought to benchmark DeepErr against an existing tool `retsnoop` and evaluate the impact of some of the techniques.

Testbed and tests. We conducted our tests on a server with Intel Xeon Gold 5420+ CPUs running Ubuntu 24.04 and Linux kernel 6.8. Although this kernel includes fixes for most of the kernel bugs we found, it still does not include a patch that allows the tracing of functions marked as "inline", which were not inlined.

To create a wide spectrum of system call failures, we employed the Linux Test Project (LTP) [19]. We executed the LTP tests using `strace` to identify tests that trigger failed system call executions. We ran LTP tests for 20 syscalls.[2] We chose these syscalls as their tests were documented well enough to understand the root cause of the failure. Using this information, we compared the results of DeepErr with the actual root cause. In total, we conducted 100 tests for both software and hardware tracing modes.

To further extend our evaluation, we analyzed five system call failures from StackExchange questions using DeepErr. These real-world cases were selected based on: (1) clear problem descriptions with reproducible code snippets, (2) confirmed solutions or accepted answers verifying the root cause, (3) diverse system calls representing different failure types, and (4) no requirement for specialized hardware or kernel modifications. This allowed us to assess DeepErr's effectiveness in diagnosing actual issues encountered by developers in practical scenarios.

We conducted evaluations using both software and hardware tracing to compare their effectiveness in different situations. To control the time for each test, we set a limit of 5 minutes for the analysis stage; if the analysis did not complete within this time frame, we terminated it.

Success analysis and limitations. To assess DeepErr's effectiveness and accuracy, we manually verified if the indicated failure cause was indeed the root cause. Overall, DeepErr successfully identified the precise root cause in 91% of cases using software tracing and 90% using hardware tracing, as summarized in Table 1. The function that returned the error code could be identified in over 96% of cases.

The technical limitations below sometimes caused complete failures (counted in Table 1), but more often resulted in minor discrepancies where DeepErr pointed to slightly different but related code. These minor discrepancies do not impact users' understanding of the root cause, and as such, we ignore them for the purpose of success evaluation:

(1) Conditional move instructions: In four cases, the root cause was a predicate compiled into a conditional move instruction for Spectre v2 protection. Since hardware tracing (Intel PT) does not capture predicated instructions, the analysis instead points to an adjacent predicate that checks the return value, still enabling root cause comprehension.

(2) Arithmetic operations for error codes: Four `EPERM` failures were misidentified because the compiler optimized the error code assignment using arithmetic operations (e.g., `sbb`) instead of conditional branches. We addressed this for software tracing by handling `sbb` instructions (similar to predicated moves), but hardware tracing cannot capture these non-branch operations. For instance, the Spectre v1 mitigation function `array_index_mask_nospec()` uses `sbb` instructions. With software tracing, DeepErr can handle these `sbb` instructions and point to them directly (though

[2] `mprotect`, `kill`, `epoll_create`, `mbind`, `getsockopt`, `open`, `getsid`, `mlock`, `mq_notify`, `pwrite`, `pread`, `rmdir`, `futex_wait`, `ioperm`, `splice`, `io_cancel`, `flock`, `recvfrom`, and `pipe`.

they may be harder for users to understand). With hardware tracing, which only records branches, DeepErr instead points to the subsequent condition check. While other arithmetic patterns exist, handling them would significantly increase analysis complexity with diminishing returns.

(3) Inline function limitations: The kernel redefines the `inline` keyword through a macro to also imply `notrace`, preventing instrumentation when inline-marked functions are not actually inlined by the compiler. This particularly affects syscall wrapper functions, which are marked inline but cannot be inlined as they are used as indirect branch targets from the syscall table. To address this limitation, we brought the issue to the kernel community's attention and provided patches to decouple the `inline` and `notrace` attributes [1, 2]. While the patches received positive feedback, they have not yet been merged into the mainline kernel.

(4) Function pointer calls: One case involved indirect calls where the callee unconditionally returned the error code, making it difficult to trace back to the actual root cause in the caller.

(5) Predicated instructions in specific functions: For example, in `_find_first_bit()`, conditional move instructions led to incorrect predicate identification with hardware tracing.

When these limitations caused complete failures rather than minor discrepancies, they particularly affected functions like `fdget` (often triggering `EBADF` errors). Overall, hardware tracing failed to identify the exact function in seven cases and the precise predicate in ten cases, while software tracing had seven cases with inexact predicate identification. However, many more cases had the minor discrepancies described above, where the identified location was slightly off but still meaningful to users.

***Comparison with* `retsnoop`.** We also attempted to run the `retsnoop` tool [21], which enables tracing of system call failures based on functions' return values. However, a fair comparison is complicated, as `retsnoop` requires the user to manually specify where the failure is likely to occur. We assumed the user might be able to identify the failing directory, for instance "security" if a permission error (`EPERM`) is returned, and accordingly ran `retsnoop` indicating it should record all functions in that directory.

With `retsnoop`, we assumed that the developer would know which directory pertains to each type of failure (e.g., fs for file-system related syscalls), but not the specific file. We attempted to set syscalls accordingly. However, `retsnoop`'s setting of kprobes for fs, mm, kernel, and other directories often fails due to the extensive number of kprobes required. Consequently, only mq_notify (in the ipc directory) succeeded, resulting in merely 6 tests being executed successfully. In practice, the experience with `retsnoop` can vary based on the type of failure one wishes to debug. Presumably due to the challenge of setting a limited number of probes, `retsnoop` provides profiles to set probes for common failures, specifically eBPF programs registration and perf events setup. However, this is not a general solution.

Callstack Distance	1	2	3	4	5	7
# Failures	81	5	6	3	3	2

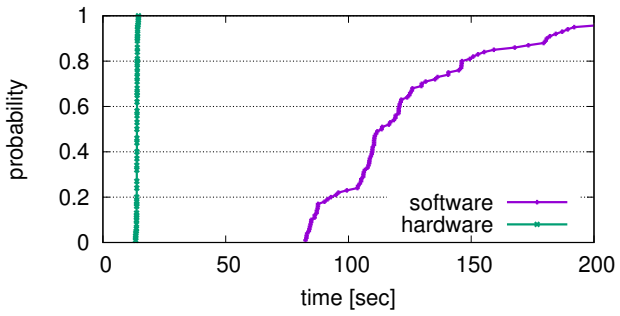
Table 2. The distance in functions on the call stack between the function that triggered the failure and the first function that returned the failure error code.

`retsnoop` can also leverage Last Branch Records (LBRs), a hardware feature in modern CPUs that records a limited history of branches taken by the processor. LBRs can provide additional context about the execution path leading to a failure, potentially helping to pinpoint the issue more accurately. This feature operates independently of `retsnoop`'s primary instrumentation, offering supplementary information rather than replacing the need to specify the failing syscall. However, in our tests, using `retsnoop`'s LBR capability did not yield any relevant information, with the message "No relevant LBR data were captured" being displayed. This result suggests that even with hardware-assisted tracing, identifying the root cause of system call failures remains challenging.

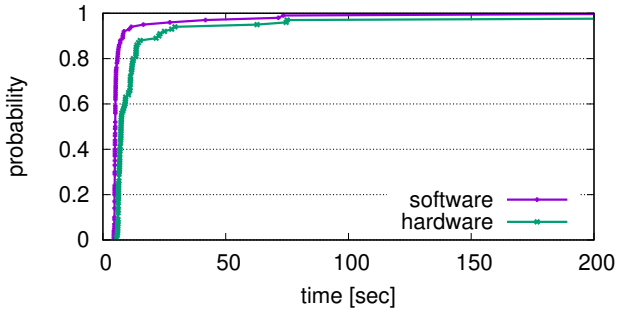
Overall, when comparing the success of pointing to the failure returning function, DeepErr outperforms `retsnoop`, with a 96-98% to 6% ratio in successful analysis.

Predicated instructions analysis. DeepErr's unique capability, when utilizing software tracing, is its precision in tracking predicated instructions beyond branches, a feature not shared with tools like `retsnoop`. We therefore examined the prevalence of common non-branch predicated instructions in root cause scenarios. Predominantly, these instructions fall within the `CMOVcc`, `SETcc`, and `SBB` families, which are handled by DeepErr. Analyzing the tests, we discovered that 5 instances had non-branch predicated instructions as their root cause. The impact might be greater since if the instruction is earlier it can affect the symbolic execution as well. This underscores the criticality of effectively monitoring such instructions for comprehensive root cause analysis.

Error code return vs. root cause location. One of the main limitations of `retsnoop` is that its methodology purely relies on return values to find the root cause, implicitly assuming that the function returning the error code holds the root cause of the failure. To challenge this assumption, we analyzed DeepErr's call stacks, measuring the function call distance between the error-returning function and the actual failure-triggering predicate. For this analysis, we ignored inlined functions within the error-returning function. Table 2 illustrates these findings, revealing that although the error-returning function or functions inlined within it are often the root cause, in 19 of the analyzed cases, the actual



(a) CDF of system call failure recording time.



(b) CDF of system call failure analysis time.

Figure 2. Cumulative Distribution Function (CDF) comparison of recording and analysis times for system call failures using DeepErr. Software tracing induces high overhead during recording but lower overhead during analysis.

root cause lies in a different function. This distinction poses a significant challenge for tools like `retsnoop`.

Analysis time. The time invested in recording and analyzing with DeepErr, as depicted in Figure 2, is non negligible. We evaluated recording overhead by comparing test durations with and without tracing. Figure 2a shows that software tracing, involving multiple stages, is always greater than 80 seconds as setting probe points takes a long time. In contrast, hardware tracing typically incurs around 10-15 seconds of overhead, as shown in Figure 2a. This overhead primarily stems from stopping the trace collection and saving the accumulated trace data to disk, rather than runtime performance impact. For long-running workloads, the actual runtime overhead of hardware tracing is negligible, making it well-suited for production debugging scenarios. However, trace analysis time, as detailed in Figure 2b, is longer with hardware tracing due to the necessity of simulating from the system call entry. In 3 tests when using hardware tracing and 1 case when using software tracing the analysis time exceeded 3 minutes. Despite the relatively long duration, the fact that analysis can be conducted offline mitigates its impact.

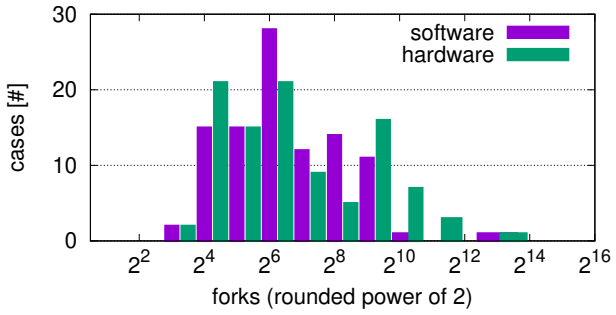


Figure 3. The number of state forks that the symbolic execution engine performs, represented on the x-axis as the nearest power of 2 and on the y-axis as the number of cases.

The analysis time is affected by the number of steps the symbolic execution engine has to take during simulation, particularly the number of times the state has to be forked and the trace needs to be followed. As Figure 3 demonstrates, the number of forks is significantly higher when hardware tracing is used, as the failure simulation has to start at the system call's entry point instead of the failing function, which is unknown with hardware tracing.

Real-world analysis. To further evaluate DeepErr's effectiveness in real-world scenarios, we analyzed five system call failures from StackExchange questions. DeepErr successfully identified the root cause in all cases, with one exception: the `prlimit64` case when using software tracing, due to the kernel's macro redefinition of `inline` to imply `notrace`. Table 3 details the time required to analyze these failures, encompassing both the record and analysis stages. Using hardware tracing, the analysis time was consistently under 30 seconds for all cases. Software tracing, while more portable, took approximately 3–5 times longer.

6 Related Work

Several tools and techniques have been developed that might be considered for system call failure analysis. However, upon closer examination, these approaches often fall short in providing comprehensive, easy-to-use solutions for identifying the root cause of failures in production environments. We discuss these tools and their limitations below.

Reverse debugging techniques, such as QEMU's record-replay [32] and hardware-based methods like REPT [9, 12], offer powerful capabilities for tracing errors backward in time. These tools can recreate the state of the system leading up to a failure, which can be invaluable for complex debugging scenarios. However, they often rely on virtual machine environments or specialized hardware support, making them challenging to deploy in production environments. Moreover, they typically introduce significant runtime overhead and lack automated analysis capabilities for pinpointing root

		tracing method [seconds]	
syscall	description	software	hardware
sched_setparam	Invalid priority for scheduling policy [28]	97 = 91 + 6	23 = 14 + 9
splice	Invalid argument due to file mode or non-pipe descriptor [27]	51 = 46 + 5	21 = 14 + 7
mmap	Incorrect alignment [30]	118 = 112 + 6	23 = 14 + 9
connect	Passing incorrect pointer to connect [29]	113 = 108 + 5	28 = 17 + 11
prlimit64	Memory soft limit higher than hard limit [31]	89 = 84 + 5	19 = 13 + 6

Table 3. System call failure analysis times using DeepErr are shown, with failures sourced from StackExchange. The times are divided into recording and analysis phases.

causes, requiring substantial manual effort to interpret results.

Comparative analysis tools such as HOLMES [7] and PerfCE [17] employ techniques that compare failing executions against known successful ones to identify divergence points. While this approach can be effective for application-level debugging, it faces significant challenges when applied to system calls. The primary issue is the need for an extensive collection of successful executions that cover the wide range of possible inputs and system states. Maintaining such a collection is impractical for system calls, especially considering the frequent updates and changes in modern OSes.

Tools focused on generating execution slices, such as Delta Debugging [39], dynamic backward slicing [37], and Gist [15], aim to provide developers with a minimal set of instructions or states necessary to reproduce a failure. These approaches can be powerful for understanding complex failure scenarios, but they often rely on assumptions that do not hold true for system call failures. Critically, Delta Debugging requires multiple iterative executions to minimize the failure-inducing input, assuming consistent reproducibility. In contrast, DeepErr's hardware tracing mode captures the complete execution trace in a single run, making it suitable for intermittent failures that may not reproduce reliably in production environments.

Symbolic execution techniques have been applied to failure analysis in tools like BugRedux [14], Replay Debugging [36], and Execution Reconstruction [42]. These approaches use symbolic execution to regenerate failures from partial execution traces or to recreate the inputs that led to a failure. While powerful, these techniques are generally aimed at reconstructing application-level failures or identifying inputs that trigger known bugs. In the context of system call failure analysis, where the inputs are typically known and the goal is to identify the specific predicate or condition causing the failure, these approaches are not directly applicable.

7 Conclusion

Addressing the challenge of analyzing system call failures, which traditionally required extensive manual effort, our work introduces DeepErr, a comprehensive solution pinpointing both the causative function and the specific triggering condition of a failure. We developed novel tracing techniques that leverage hardware branch tracing when available and complement it with software-based probe-point tracing for broader applicability. Our approach employs symbolic execution for differential analysis without requiring successful reference executions. We believe these techniques can be adapted and extended to address similar challenges in other contexts, potentially impacting areas beyond system call failure analysis.

Availability

We release DeepErr as open-source software under the BSD-2 license. The tool and source code are available at `https://github.com/anadav/deeperr`.

Acknowledgments

We thank the anonymous reviewers and our shepherd, Tobias Distler, for their valuable feedback and suggestions that helped improve this paper.

References

[1] Nadav Amit. 2022. [RFC PATCH] x86/syscalls: allow tracing of __do_sys_[syscall] functions. Linux Kernel Mailing List, https://lore.kernel.org/all/20220913135213.720368-1-namit@vmware.com/.

[2] Nadav Amit. 2023. [PATCH v2 2/3] compiler: inline does not imply notrace. Linux Kernel Mailing List, https://lore.kernel.org/all/20230525210040.3637-3-namit@vmware.com/.

[3] David Anderson. 2022. White Paper: Crash Utility. http://crash-utility.github.io/crash_whitepaper.html.

[4] Suren Baghdasaryan. 2022. Code tagging framework and applications. https://lore.kernel.org/lkml/20220830214919.53220-1-surenb@google.com/.

[5] Joe 'Zonker' Brockmeier. [n. d.]. Using Strace to Trace Problems. https://www.serverwatch.com/guides/using-strace-to-trace-problems/. Accessed: 2025-07-28.

[6] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. 2008. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs.. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*. 209–224.

[7] Trishul M Chilimbi, Ben Liblit, Krishna Mehra, Aditya V Nori, and Kapil Vaswani. 2009. Holmes: Effective statistical debugging via efficient

path profiling. In *IEEE International Conference on Software Engineering (ICSE)*.

[8] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. 2011. S2E: A platform for in-vivo multi-path analysis of software systems. *ACM SIGPLAN Notices* 46, 3 (2011), 265–278.

[9] Weidong Cui, Xinyang Ge, Baris Kasikci, Ben Niu, Upamanyu Sharma, Ruoyu Wang, and Insu Yun. 2018. REPT: Reverse debugging of failures in deployed software. In *USENIX Symposium on Operating Systems Design & Implementation (OSDI)*.

[10] The Linux Kernel Documentation. 2018. Using kgdb, kdb and the kernel debugger internals. https://www.kernel.org/doc/html/v4.18/dev-tools/kgdb.html. Accessed: 2025-07-28.

[11] Jake Edge. 2024. Tracing the source of filesystem errors. https://lwn.net/Articles/984556/. Accessed: 2025-07-28.

[12] Xinyang Ge, Ben Niu, and Weidong Cui. 2020. Reverse debugging of kernel failures in deployed systems. In *USENIX Annual Technical Conference (ATC)*.

[13] Stefan Hajnoczi. 2019. Determining why a Linux syscall failed. http://blog.vmsplice.net/2019/08/determining-why-linux-syscall-failed.html.

[14] Wei Jin and Alessandro Orso. 2012. BugRedux: Reproducing Field Failures for in-House Debugging. In *IEEE International Conference on Software Engineering (ICSE)*. 474–484.

[15] Baris Kasikci, Benjamin Schubert, Cristiano Pereira, Gilles Pokam, and George Candea. 2015. Failure sketching: A technique for automated root cause diagnosis of in-production failures. In *ACM Symposium on Operating Systems Principles (SOSP)*. 344–360. https://doi.org/10.1145/2815400.2815412

[16] Andi Kleen and Beeman Strong. 2015. Intel processor trace on Linux. https://tracingsummit.org/ts/2015/IntelProcessorTraceOnLinux/. In *Tracing Summit*.

[17] Ben Liblit, Mayur Naik, Alice X Zheng, Alex Aiken, and Michael I Jordan. 2005. Scalable statistical bug isolation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*.

[18] Linux Kernel Documentation [n. d.]. x86 exception-tables. https://mjmwired.net/kernel/Documentation/x86/exception-tables.txt. Accessed: 2025-07-28.

[19] Linux Test Project. [n. d.]. Linux Test Project. http://linux-test-project.github.io/. Accessed: 2025-07-28.

[20] Michael Meng, Stephanie M Steinhardt, and Andreas Schubert. 2020. Optimizing API documentation: Some guidelines and effects. In *ACM International Conference on Design of Communication (SIGDOC)*. 1–11.

[21] Andrii Nakryiko. 2023. retsnoop: Investigate kernel error call stacks. https://github.com/anakryiko/retsnoop. Accessed: 2025-07-28.

[22] Martin P Robillard and Robert DeLine. 2011. A field study of API learning obstacles. *Empirical Software Engineering* 16 (2011), 703–732.

[23] Steven Rostedt. 2010. Ftrace Linux kernel tracing. In *Linux Conference Japan*.

[24] Omar Sandoval. 2021. drgn: How the Linux Kernel Team at Meta Debugs the Kernel at Scale. https://developers.facebook.com/blog/post/2021/12/09/drgn-how-linux-kernel-team-meta-debugs-kernel-scale/.

[25] Heinrich Schuchardt. 2014. ioctl_list.2: complete overhaul needed. https://lore.kernel.org/all/545F8D2E.5030308@gmx.de/.

[26] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy (SP)*.

[27] Stack Overflow User. 2010. Invalid argument when calling linux splice(). https://stackoverflow.com/questions/2580390/invalid-argument-when-calling-linux-splice. Accessed: 2025-07-28.

[28] Stack Overflow User. 2010. pthread_setschedprio() fails with "EINVAL". https://stackoverflow.com/questions/3140505/pthread-setschedprio-fails-with-einval. Accessed: 2025-07-28.

[29] Stack Overflow User. 2014. socket connect() returns errno EINVAL. https://stackoverflow.com/questions/24141307/socket-connect-returns-errno-einval. Accessed: 2025-07-28.

[30] Stack Overflow User. 2020. How do I troubleshoot the EINVAL and EPERM errors I'm getting in this x86-64 assembly code using the mmap syscall on Linux. https://stackoverflow.com/questions/65203077/how-do-i-troubleshoot-the-einval-and-eperm-errors-im-getting-in-this-x86-64-ass. Accessed: 2025-07-28.

[31] Stack Overflow User. 2022. setrlimit fails and rlim_max is 0. https://stackoverflow.com/questions/74066587/setrlimit-fails-and-rlim-max-is-0. Accessed: 2025-07-28.

[32] The QEMU Project Developers [n. d.]. Record/replay. https://www.qemu.org/docs/master/system/replay.html. Accessed: 2025-07-28.

[33] Linus Torvalds. 2024. Re: [PATCH v8 0/4] Introduce mseal. https://lore.kernel.org/lkml/CAHk-=wjNXcqDVxDBJW8hEVpHHAE0odJEf63+oigabtpU6GoCBg@mail.gmail.com/.

[34] UNIX developers. 1973. sys/user.h. https://github.com/dspinellis/unix-history-repo/blob/Research-V4-Snapshot-Development/sys/user.h#L39.

[35] StackExchange Users. [n. d.]. Search results for 'einval' on Stack Exchange. https://stackexchange.com/search?q=einval. Accessed: 2025-07-28.

[36] Peipei Wang, Hiep Nguyen, Xiaohui Gu, and Shan Lu. 2016. RDE: Replay DEbugging for diagnosing production site failures. In *IEEE International Symposium on Reliable Distributed Systems (SRDS)*. 327–336.

[37] Yan Wang, Harish Patil, Cristiano Pereira, Gregory Lueck, Rajiv Gupta, and Iulian Neamtiu. 2014. DrDebug: Deterministic replay based cyclic debugging with dynamic slicing. In *IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 98–108.

[38] Cristian Zamfir, Gautam Altekar, and George Candea. 2011. Debug Determinism: The Sweet Spot for Replay-Based Debugging. In *USENIX Workshop on Hot Topics in Operating Systems (HOTOS)*.

[39] Andreas Zeller and Ralf Hildebrandt. 2002. Simplifying and isolating failure-inducing input. *IEEE Transactions on Software Engineering* 28, 2 (2002), 183–200.

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

[41] Hao Zhong and Hong Mei. 2017. An empirical study on API usages. *IEEE Transactions on Software Engineering* 45, 4 (2017), 319–334.

[42] Gefei Zuo, Jiacheng Ma, Andrew Quinn, Pramod Bhatotia, Pedro Fonseca, and Baris Kasikci. 2021. Execution reconstruction: Harnessing failure reoccurrences for failure reproduction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 1155–1170.