Fast (Trapless) Kernel Probes Everywhere

Abstract

The ability to efficiently probe and instrument a running operating system (OS) kernel is critical for debugging, system security, and performance monitoring. While efforts to optimize the widely used Kprobes in Linux over the past two decades have greatly improved its performance, many fundamental gaps remain that prevent it from being completely efficient. Specifically, we find that Kprobe is only optimized for ~80% of kernel instructions, leaving the remaining probeable kernel code to suffer the severe penalties of double traps needed by the Kprobe implementation. In this paper, we focus on the design and implementation of an efficient and general *trapless* kernel probing mechanism (no hardware exceptions) that can be applied to almost all code in Linux. We discover that the main limitation of current probe optimization efforts comes from not being able to assume or change certain properties/layouts of the target kernel code. Our main insight is that by introducing strategically placed *nops*, thus slightly changing the code layout, we can overcome this main limitation. We implement our mechanism on Linux Kprobe, which is transparent to the users. Our evaluation show an 10x improvement of probe performance over standard Kprobe while providing this level of performance for 96% kernel code.

1 Introduction

The ability to instrument (a.k.a. "probe") a running OS kernel is critical for not only debugging and event tracing [11] but also for system security [8, 9], performance monitoring [21], and dynamic patching [10]. An efficient and fast kernel probing mechanism is key to enabling the use of these applications directly in the field on production systems and can open up a rich set of new use cases, such as enforcing kernel control flow integrity (KCFI) with dynamic policies [12].

A kernel probe allows users to dynamically instrument arbitrary kernel instructions to execute user-provided handlers—pre-handler for before executing the probed instruction and post-handler for after. To intercede on the kernel's control flow and invoke these handlers, typical kernel probe implementations rely on *traps*. When a probe is registered, the kernel makes a copy of the probed instruction and replaces it with a breakpoint instruction. When execution hits the breakpoint instruction, a trap occurs and the control is transferred to the probe subsystem, which executes the pre-handler, probed instruction, and post-handler before resuming normal execution at the instruction following the probe point. The obvious drawback of trap-based probes is the significant overhead due to the expensive context-switches involved (more than 6,000 CPU cycles from our measurements).

To overcome the above drawback, a *trapless* approach is needed. The key idea is to replace the expensive traps with control redirecting instructions like jump instructions. While a trapless approach can eliminate the overhead associated with the traps, it introduces a major challenge. Specifically, for variable instruction set architectures like x86, a jump instruction can be larger than the instruction it is probing, causing an overwrite of multiple instructions. This can cause the jump instruction to span basic block boundaries and cause execution failure. This challenge can limit where trapless probes can be used, thus how many instructions can be trapless.

Whether using a trap-based or trapless approach, another challenge with implementing a kernel probing mechanism is how to execute the copied probed instruction efficiently. One typical choice is to execute the copied instruction directly on the processor. However, this direct execution does not work for some sets of instructions such as those related to the instruction pointer, e.g., calls and jumps. These instructions must be emulated, which is slow and adds complexity in terms of implementing and maintaining the emulation code.

In this paper, we present the design and implementation of a universally fast (*trapless* probing on all probe-able code) kernel probing mechanism that requires no runtime code emulation. Our design allows probe handlers to be executed synchronously in the same context that triggered the probe and thereby avoids expensive context switches. To achieve this design, we rely on a key insight—by strategically inserting nops into the kernel code, thus slightly changing the code layout, we can overcome the above two challenges. Specifically, for probe locations that straddle basic block boundaries, an inserted nop can ensure the jump instruction resides in one basic block; for instructions that require code emulation, placing a nop before such instructions allows a probe to be attached by overwriting the nop with a call to the kprobe handlers, thereby allowing the probed instruction to be executed in place with no copy or emulation. Most importantly, in our approach, the location to place the nops can be automatically identified and kept minimal, thus reducing the impact on normal kernel operations when no probe is installed.

To demonstrate the efficacy of our approach, we apply our design to Linux Kprobe [2] on x86, a widely used kernel probing mechanism and architecture. Kprobe has been transformed over the years from a purely trap-based probe mechanism to utilizing a trapless approach [10]. However, despite persistent optimization efforts, due to many fundamental limitations stemming from not being able to assume or change certain properties/layout of the kernel code at runtime, Kprobe is far from being universally trapless. Exacerbating the problem, existing Kprobe optimizations are often applied in an ad hoc manner, resulting in many instructions being

unoptimized, even if it is technically possible, oftentimes due to the sheer complexity of the implementation. We discovered that the existing Kprobe is only optimized for ~80% of kernel instructions, leaving the remaining probe-able kernel code to suffer the severe penalties of double traps when being probed.

Our universally trapless kprobe implementation consists of a new transformation pass in the LLVM x86 backend to identify the locations where a `nop` is needed and perform the insertion. To allow kprobes to be optimized using the inserted `nop`s, we implemented kernel support for efficient, scalable trampolines and runtime instruction rewriting.

We evaluate both performance and optimization coverage of our design. Our kprobe implementation achieves a speedup of up to 3x for kprobe-based KCFI enforcements on LEBench [20] over the original Kprobe with the single-probe performance increased by a factor of 10x. Our kprobe implementation optimizes all the kernel instructions that can be optimized at the compile time and brings the total instructions optimizable in the kernel to 96%. In fact, even more performance improvement can be potentially achieved (we prioritized compatibility and non-disruptive changes to accommodate the current optimizations of Linux Kprobe).

In summary, this paper makes the following contributions:

- We present a fast and universal trapless kernel probe design;
- We identify fundamental limitations of existing Linux Kprobe optimization techniques;
- We implement our design on top of Linux Kprobe and showing the efficacy of the approach.

2 Design

The design of our trapless kernel probe mechanism has two main goals: 1) the kernel probe should be fast when applied to any kernel instructions without using expensive traps (e.g., `int3` exceptions) and 2) the mechanism should be simple to not increase implementation and maintenance complexity of kernel code and lightweight to not incur significant overhead. Since a kernel probe is inserted at runtime when the code layout is fixed, there are two main challenges with implementing a universally trapless kernel probe mechanism: 1) how to ensure the inserted jump instruction does not span basic block boundaries (which would unsafely overwrite branch targets) and 2) how to allow instructions that would normally require emulation in a typical kernel probe implementation to be directly executed. In this section, we discuss a clean-slate design of a universal trapless kernel probe mechanism before describing how we apply our design to Linux Kprobe (§ 3).

2.1 A Baseline Solution

Our key insight for resolving the first challenge is to use the insertion of `nop`s to change the code layout of the kernel. The `nop`s inserted at compile time can provide space and be used as anchor points for inserting probes at runtime without tampering with regular kernel instructions.

The first basic incarnation of this approach is to insert a 5-byte `nop` instruction before every kernel instruction. In this way, probing a specific instruction works by rewriting the preceding `nop` into a same-sized relative `call` that redirects the control flow onto a global trampoline. The trampoline first saves register contexts on the stack. This prevents the user-defined handler from overwriting the current execution context. Next, the trampoline invokes the user-defined handler to perform actual probing. Finally, after the handler finishes, the trampoline pops the register contexts and returns. The returned control flow would land on the next instruction after the `call`, which is exactly the probing target. In this way, execution can continue with the correct context.

While this approach ensures that no trap is needed, and thus any kernel probe is fast, inserting a `nop` before every instruction would introduce significant overhead when the kernel probes are not used. Our evaluation shows a 75% slowdown for LEBench running on Linux v6.3.6 (§ 4).

2.2 Minimal `nop`s Design

Clearly, improving the baseline would require strategically placing `nop`s. Fortunately, we observe that the vast majority of the `nop` instructions in the baseline can be omitted. Specifically, if there is enough space before a jump target, there is a potential opportunity of rewriting a relative `jmp` to the trampoline in place [10]. Doing this requires us to copy the target instruction and any other instruction that could be overwritten by the 5-byte relative `jmp` to a temporary copy buffer. After the trampoline invokes the user-supplied handler and restores the register context, it executes the instructions saved in the copy buffer and performs another relative jump back to the normal execution path. This implementation requires the copied instructions to be executed out-of-line (in the buffer), not at the original code location.

Only in the cases where in-place replacement of a `jmp` instruction is not possible, we apply the `nop`-based design. Such cases can be broadly classified into two categories:

- instructions that cannot be executed out-of-line.
- instructions that do not have enough space without overwriting a jump target (i.e., start of a basic block).

The set of instructions that cannot be executed out-of-line includes instructions for which their text address matters for execution. These groups of instructions include `call` instructions and also instructions that may trigger page faults. The `call` instruction belongs to this group because the callee may be expecting a specific caller (e.g., via `__builtin_return_address`) and, therefore, cannot be executed directly on the copy buffer. Otherwise, the return address obtained would point to the copy buffer rather than the original address. On the other hand, certain instructions, e.g.,

userspace access instructions, could trigger a page fault if an invalid address is being accessed. Linux defines short "fixup code" snippet that implement the fault-recovery logic to allow execution to continue for such instructions. The mapping between the faulting instruction and its fixup code is stored in a special exception table section of the kernel image. During a page fault, the page fault handler uses the address of the faulting instruction to search for the corresponding fixup code in the exception table. This effectively prevents the instruction from being executed out-of-line in the copy buffer, since there is not an entry in the exception table that corresponds to the address of the faulting copy. In these situations, the inserted nop will still make it possible to probe such an instruction without the need for a trap.

For instructions at the end of a basic block, the rewriting technique may not be viable, because blindly rewriting these instructions would overflow into the next basic block if there is not enough space. Overwriting the entry of the next basic block breaks instruction encoding when the control flow lands on that basic block through a jump. Therefore, nops are still needed under these situations since they allocate space to safely place a `call` for efficient probes and avoid the traps.

3 Implementation

We discuss the application of our approach to Kprobe in Linux. We start by describing the current Kprobe implementation with current optimization techniques. We then highlight the limitations of Kprobe in achieving universal trapless kernel probing. Finally, we discuss how our approach can help overcome those limitations and present the details of a working implementation. Our aim is to have an implementation that minimizes changes to the kernel code, be fully compatible with Kprobe, and ensure the changes are transparent to users.

3.1 Linux Kprobe

Without optimizations, a trap-based kprobe on x86-64 uses two traps (Figure 1). When a kprobe is registered, the kernel makes a copy of the probed instruction (to avoid race conditions) and replaces the first byte of the probed instruction with a breakpoint instruction, i.e., `int3` on x86-64. A second `int3` is appended at the end of the copied instruction.

When the execution hits the breakpoint instruction, a trap occurs, the CPU's registers are saved, and the control is transferred to the Kprobe subsystem, which executes a user-provided pre-handler that is associated with the kprobe. Next, Kprobe single-steps the copy of the probed instruction to ensure the kernel regains control to execute any user-installed post-handler associated with that kprobe. The single-step is implemented by the second `int3` trap after the copy. (known as "`int3`" single-step [15]) [1]. Execution then continues with

[1]The kernel uses `int3` instead of a Trap Flag (TF) to avoid implementation complexity of handling the side effect of using `iret` (interrupt return) [15]

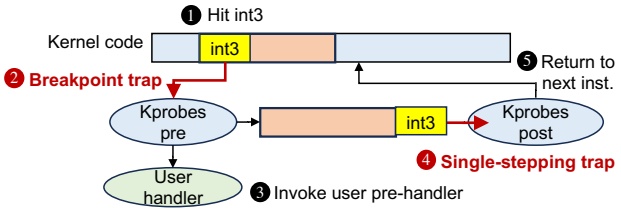

Figure 1: A kernel probe takes two traps (steps 2 and 4)

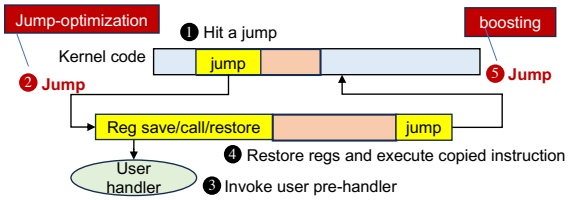

Figure 2: Existing Kprobe optimizations known as boosting and jump-optimization.

the instruction following the probe point.

Trap-based kprobes suffer greatly from the expensive context-switch overheads. Our measurement shows that such a kprobe consumes more than 6000 CPU cycles.

3.1.1 Existing Kprobe Optimizations

Since trap-based kprobes suffer greatly from expensive context-switches, optimizations are developed to replace expensive traps with jump instructions [10] (Figure 2). In Linux, Kprobe currently employs two optimizations, named *boosting* and *jump-optimization* for the two traps, respectively.

Boosting. Boosting aims to replace the single-step trap with a jump instruction. The insight is that if a kprobe does not have a post-handler, then single-stepping may not be needed. So, for a kprobe without a post-handler, the boosting optimization adds a jump which jumps back to the next instruction, replacing the single-step trap. In this way, the kernel executes the copied instruction and the jump. Note that, unlike a pre-handler, a post-handler is not commonly used.

Boosting is limited in scope since it cannot handle instructions that change the instruction pointer register (`rip`), e.g., `call`, and the instructions which may require exception fixups. Instructions like `call` need to be emulated, instead of being executed out-of-line, because the return address pushed onto the stack needs to be corrected.

Jump-optimization. Jump-optimization builds upon boosting, aiming to replace the breakpoint trap with a five-byte relative jump that redirects the control flow to a pre-allocated trampoline. The trampoline calls into the Kprobe pre-handler and then jumps back to the next instruction. In this way, Kprobe invokes the pre-handler synchronously on the same execution context and avoids the expensive context switch.

Note that jump-optimization may need to copy several instructions, instead of only the probed instruction in the orig-

Instruction	Count (Percentage)
Non-boostable	949,396 (14.31%)
Non-jump-optmizable (boostable)	445,037 (6.71%)
Total	1,394,433 (21.02%)

Table 1: The number (percentage) of instructions in Linux (v6.3.6) that are not optimized by Kprobes. There are in total 6,632,661 kprobe-able instructions.

inal Kprobe (§3.1), because jump, as a five-byte instruction, may be longer than the probed instruction. The unoptimized Kprobe with `int3` does not encounter this problem because `int3` is a one-byte instruction. Kprobe implements an x86 instruction decoder to recognize the instruction boundary.

Like boosting, jump-optimization is also limited. It suffers from the same basic block spanning issues described Section 2. The current Kprobe implementation handles this limitation conservatively, because it cannot reason about basic block boundaries (i.e., branch targets) at runtime. Therefore, in addition to ensuring no near jump to the region of the jump instruction, Kprobe also refuses to optimize *any* instructions in a function that contains indirect jumps. Also, the implementation of jump-optimization depends on boosting; so, non-boostable instructions cannot be jump-optimized.

3.1.2 Kprobe Limitations

The Kprobe optimizations are not only limited in scope but also applied to kernel instructions in an *ad hoc* way. Table 1 shows the amount of instructions that cannot be optimized by either boosting or jump-optimization in Linux (v6.3.6).[2] In total, 21.3% of the instructions in Linux cannot be fully optimized, i.e., attaching a kprobe on around one-fifth of the kernel text needs to pay for the context switch overhead raised by breakpoint exceptions. Among them, the majority are non-boostable (and thus cannot be jump-optimized).

In addition to the limitations discussed in §3.1.1, the process of applying the two optimizations to the trap-based implementation (§3.1) is often ad hoc. Linux adopts a strategy of applying optimizations based on a few types of instructions that are deemed to be safe. Table 2 lists the types of instructions that are not non-boostable in Linux. For example, all Group 2, 3, and 4 instructions are deemed as non-optimizable in the kernel, while some of them can clearly be optimized (e.g., `inc`/`dec` for increments/decrements). In fact, the dependency of the two optimizations is also an implementation artifact; in principle, the two optimizations are independent.

3.2 Universally Fast Kprobe

Instead of continuing to manually apply existing optimizations to more types of instructions and special cases, we seek

[2]There are ~6K instructions that are not probe-able such as trap instructions like ud2 and int3 and functions labeled as non-traceable.

1. `rip`-changing instructions, e.g. jumps and calls
2. Instructions that may trigger exceptions (e.g. pages faults)
3. Instructions that override address size or code segment
4. x86 instructions group 2/3/4/5 with reserved opcodes.
5. Instructions with 3-byte opcodes

Table 2: Instructions that cannot be boosted (thus not jump-optimized in the current Kprobe implementation).

a principled, universal implementation using our approach. Our implementation consists of a compiler transformation that selectively inserts `nop`s (§3.2.1) and kernel support for efficient trampolines and instruction rewriting (§3.2.2). The former takes 298 lines of C code and the latter takes 549 lines of C++ code. We build on top of Linux v6.3.6.

3.2.1 Compiler Transformation

We develop a compiler transformation to identify the minimal set of locations where `nop`s are needed to enable trapless kprobes. The transformation is implemented on LLVM as a MachineFunctionPass that works at the Machine IR (MIR) level [3] in the code-generation backend. We chose MIR instead of the generic LLVM IR because MIR closely models native code and thus makes it easy to check whether an instruction can be optimized by Kprobe at runtime.

The transformation takes two iterations and operates on each kernel function; it goes through each instruction in the function and identifies instructions that are not boosted or jump-optimized by Linux:

- If the instruction cannot be boosted, a `nop` instruction is inserted before that instruction.
- If the instruction can be boosted but not jump-optimized (which means that inserting a jump would overflow into another basic block), a `nop` instruction is inserted before the last instruction of the basic block.

Enabling universal boosting. Table 2 shows the categories of non-boostable instructions. For each non-boostable instruction category that can be identified by opcode, the compiler pass inserts a `nop` before the instruction.

Inserting a `nop` is straightforward in most cases. One special case is to handle *terminator instructions* of basic blocks, i.e., instructions at the end of the basic block that direct the control flow to the successor basic blocks. Since such instructions modify instruction pointers, they are almost always not boostable. The LLVM MIR implements the semantics similar to native assembly code and permits basic blocks to have multiple terminators (e.g., a basic block may have a conditional jump (`jcc`) followed by a direct jump (`jmp`)—the direct jump is executed when the conditional jump is not taken). The LLVM backend requires no non-terminator instructions between terminators. Therefore, it is invalid to insert `nop`s between terminators, which is required for non-boostable terminators. To handle this case, our compiler pass splits the

basic blocks with multiple terminators into multiple basic blocks, each of which has one terminator. The transformation maintains the original control flow, and allows the insertion of a nop in front of each terminator if needed.

The only non-boostable category that cannot be identified by opcode directly is the instructions that may trigger exceptions (e.g., page faults). Our design covers these instructions, but our current implementation does not handle them because the kernel exception table with the actual kernel address is not available until after linking.

Enabling universal jump-optimization. For instructions that can be boosted but cannot be jump-optimized, our compiler pass aims to enable jump-optimization to avoid the breakpoint trap. The reason that jump-optimization is not applicable is due to the lack of space to place the five-byte jump instruction.

The problem is straightforward to fix at compile time, because basic blocks are explicit and branches can only jump to the entries of basic blocks. Therefore, the problem is reduced to handling the case when rewriting a jump would overflow to the next basic block. We address this case by inserting a nop before the last instruction of each basic block. Note that basic block terminators are handled by the prior iteration.

3.2.2 Kernel Support

We keep the Kprobe interface for probe registration. If the kprobe being registered is not jump-optimizable, our implementation will optimize it onto a preceding nop (if exists). This leverages the kernel text-patching interface to replace the 5-byte nop instruction with a 5-byte relative call instruction that redirects the control flow onto our trampoline.

Efficient, scalable trampoline design. In Kprobe, the pre-handlers expect to receive the registers (pt_regs) of the context that triggered the probe. In trap-based kprobes (§3.1), registers are automatically saved by the processor. For jump-optimized kprobes that do not use a trap, their trampolines explicitly save the register context and invoke the user handlers with the saved context.

Currently, the Kprobe jump-optimizer creates one trampoline for each instruction a jump-optimized kprobe is attached. The jump-optimizer copies a pre-defined trampoline "template" and fills it with information specific to that probe. For example, the kernel copies the instructions overwritten by the jmp to the end of the trampoline. Such a design is needed because the instructions cannot be directly executed in place after being overwritten by the jmp – they must be copied to a trampoline customized for that kprobe. At the same time, since jump-optimization uses a relative jmp to redirect the control flow to the Kprobe pre-handlers, the jump-back address needs to be hardcoded into the trampoline in order to resume the normal execution after probing.

However, creating one trampoline for every probed instruction not only is complicated but also does not scale well. We address this issue by designing a single, global trampoline for all the kprobes that are optimized with the nop (§2.1), because the probed instructions can be executed in place.

Similar to the trampolines of jump-optimization, our trampoline pushes the registers onto the kernel stack with specific ordering so that the pushed values form a pt_regs struct which can be directly used by user handlers. A challenge for our one-trampoline design is to handle the instruction pointer register (rip). The pre-handler expects the rip to be the address of the probed instruction. However, the rip changes at each instruction and cannot be trivially pushed onto the stack. Unlike the per-instruction trampoline, the address of the probed instruction is not encoded on the global trampoline.

We solve the problem by rewriting our 5-byte nop into a 5-byte relative call instruction that calls onto the trampoline, instead of the relative jmp used by jump-optimization. The call instruction automatically pushes the return address onto the stack. This address is always the address of the next instruction after the nop, which is exactly the probed instruction and the rip value the handler expects. In this way, the trampoline can just read the value of the expected rip from the stack and store it into the pt_regs. The use of the call instruction also allows us to jump back to normal execution as a ret will be enough. We implement our trampoline in x86-64 assembly. The finished trampoline after compilation takes a constant 96 bytes in the kernel, a significant improvement over the linear memory complexity incurred by jump-optimization.

4 Evaluation

We evaluate our trapless kprobe mechanism, denoted as Uno-kprobe, on its optimization coverage, probe performance, and nop overhead. All measurements are performed on a bare metal server with an 8-core Intel Xeon E-2174G CPU with 32GB of memory on a 1Gb/s network.

4.1 Trapless Kprobe Coverage

We assess the amount of kernel code that can utilize our trapless Uno-kprobe. Under the original Kprobe implementation, the class of instructions in the kernel that are not optimizable include instructions that cannot be boosted to execute out-of-line (e.g., rip-modifying instructions) and instructions at the end of basic blocks where jump-optimization could overwrite branch targets. Both classes can now be optimized with Uno-kprobe. Table 3 shows the amount of kernel code that can be optimized under the original Kprobe and our Uno-kprobe. Our approach can optimize all currently unoptimized instructions that can be identified at compile time, bringing the total instructions optimizable in the kernel from 79% to 96%. The remaining non-optimizable instructions include instructions that may trigger pages faults and instructions from inline-assembly blocks as well as assembly files (these instructions cannot be trivially processed in the LLVM code generator).

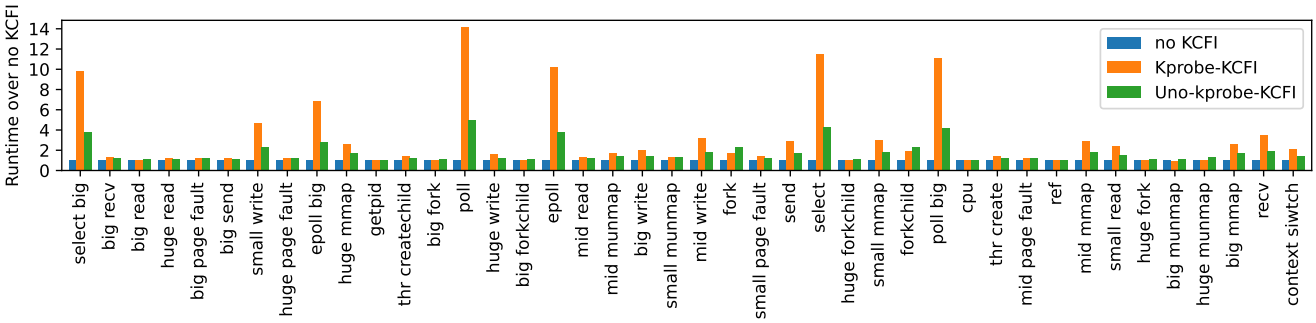

Figure 3: Runtime overhead of different KCFI policies when executing LEBench, with vanilla Kprobe in Linux and Uno-kprobe. Value reported is based on median runtime of each benchmark.

4.2 Performance

We first evaluate the performance of Uno-kprobe on a microbenchmark. Our experiment consists of a single kprobe with an empty handler attached to an instruction and measures the total latency of the kprobe and instruction. We measure the latency using both the original Kprobe implementation and Uno-kprobe and on both a boostable but not jump-optimizable `nop` (located at the end of a basic block) and a non-boostable `shr`. The results are shown in Table 4. Uno-kprobe is about 10x faster than the existing Kprobe on a non-boostable instruction (i.e., with two traps) and 5x faster than that on a boostable but not jump-optimizable instruction (i.e., with one trap) as both of these probe sites are fully trapless with Uno-kprobe.

We then evaluate Uno-kprobe performance on applications utilizing kprobes. Specifically, we measure the performance of the LEBench [20] benchmark with a kprobe on all indirect calls in the kernel that resembles a kernel CFI (KCFI) use case. Figure 3 shows the overhead of LEBench when KCFI is enabled using different techniques for invoking CFI handlers. As can be seen, using the original Kprobe results in the highest overhead, with some system calls having overhead up to 3x compared with our trapless kprobes. On average, Uno-kprobe achieves a speedup of 1.4x across all LEBench. Kprobe has been recognized by prior work to perform poorly in use cases that require a high rate of invocation [8, 12]. Uno-kprobe makess it feasible to use kprobe for such an application.

Lastly, we evaluate the overhead introduced by our inserted `nop`s when kprobes are not used. We measure the performance of the LEBench under different `nop` insertion strategies: vanilla kernel (no `nop`, `nop` before non-optimizable instructions, and a `nop` before every instruction. We found that inserting `nop`s before every instruction yields a rather large overhead, 30% on average. Uno-kprobe, in contrast, incurs an overhead of 10% on average on LEBench when kprobes are not registered and achieves a performance advantage of up to 1.5x (`mid mmap`) over inserting `nop`s before every instruction. This may be acceptable for applications that heavily rely on the probing functionality and already must incur the probing overhead, given the performance boost (up to 10x) at all probe-able locations that can be achieved with Uno-kprobe.

Table 3: Kernel code trapless-probe coverage of Uno-kprobe

Total instructions	Vanilla Kprobe	Uno-kprobe
6.63M	79% (5.24M)	96% (6.38M)

Table 4: Latency (in cycles) of invoking an empty handler using vanilla Kprobe and Uno-kprobe.

	Vanilla Kprobe	Uno-kprobe
No optimization	6235 ± 817	612 ± 407
Boost-only	2625 ± 2459	562 ± 369

5 Related Work

SystemTap [17], DTrace [6], and Ftrace [1] all use trap-based probing mechanisms. Similarly, Ptrace uses traps to provide userspace applications with a mechanism to hook onto processes. Another trap-based probing mechanism is the Xenprobes [18] for probing guest kernels. Mechanisms for secure active monitoring [16], dynamic operating systems monitoring [7], memory introspection [13], and intrusion detection in the kernel [19] use forms of trap-based probing. Our work offers a principled solution to optimize probes with handlers sharing the same address space, but would require different design for probes requiring a world-switch.

Unlike trap-based techniques, many instrumentation probes utilize trampolines instead of breakpoints. DynamoRIO [5], DynInst [4], and Intel Pin [14] are such tools where some of them use emulation, which might benefit from our work and would be an interesting future work.

6 Concluding Remarks

This paper presented a fast and universal trapless kernel probe design and its implementation on top of the Linux Kprobe subsystem. We show that the performance of kernel probes can be effectively improved through a general design. Our future work includes further improving our implementation to completely realize pervasive trapless kprobes.

References

[1] ftrace - Function Tracer. https://docs.kernel.org/trace/ftrace.html, 2023. Accessed October 19, 2023.

[2] Kernel Probes (Kprobes). https://docs.kernel.org/trace/kprobes.html, 2023. Accessed October 19, 2023.

[3] Machine IR (MIR) Format Reference Manual. https://llvm.org/docs/MIRLangRef.html, 2023. Accessed December 27, 2023.

[4] BERNAT, A. R., AND MILLER, B. P. Anywhere, Any-Time Binary Instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE'11)* (2011).

[5] BRUENING, D. *Efficient, Transparent, and Comprehensive Runtime Code Manipulation*. PhD thesis, Massachusetts Institute of Technology, Department of Electrical Engineering and Computer Science, 2004.

[6] CANTRILL, B. M., SHAPIRO, M. W., AND LEVENTHAL, A. H. Dynamic Instrumentation of Production Systems. In *Proceedings of the 2004 USENIX Annual Technical Conference (ATC'04)* (2004).

[7] ESTRADA, Z. J., PHAM, C., DENG, F., YAN, L., KALBARCZYK, Z., AND IYER, R. K. Dynamic VM Dependability Monitoring Using Hypervisor Probes. In *Proceedings of the 11th European Dependable Computing Conference (EDCC'15)* (2015).

[8] FOURNIER, G. Return to sender: Detecting kernel exploits with eBPF. In *Blackhat USA* (2022). https://i.blackhat.com/USA-22/Wednesday/US-22-Fournier-Return-To-Sender.pdf.

[9] HARDENEDVAULT. Ved-ebpf: Kernel exploit and rootkit detection using ebpf. https://github.com/hardenedvault/ved-ebpf, 2023. Accessed 2023.

[10] HIRAMATSU, M. The Enhancement of Kernel Probing — Kprobes Jump Optimization. In *Tracing Summit* (2010). https://tracingsummit.org/ts/2010/files/HiramatsuLinuxCon2010.pdf.

[11] IOVISOR. bpftrace. https://github.com/iovisor/bpftrace, 2023. Accessed 2023.

[12] JIA, J., LE, M. V., AHMED, S., WILLIAMS, D., AND JAMJOOM, H. Practical and Flexible Kernel CFI Enforcement using eBPF. In *Proceedings of the 1st Workshop on eBPF and Kernel Extensions (eBPF'23)* (2023).

[13] KLEMPERER, P. F., JEON, H. Y., PAYNE, B. D., AND HOE, J. C. High-Performance Memory Snapshotting for Real-Time, Consistent, Hypervisor-Based Monitors. *IEEE Transactions on Dependable and Secure Computing 17*, 3 (2018), 518–535.

[14] LUK, C.-K., COHN, R., MUTH, R., PATIL, H., KLAUSER, A., LOWNEY, G., WALLACE, S., REDDI, V. J., AND HAZELWOOD, K. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)* (2005).

[15] LUTOMIRSKI, A. Why do kprobes and uprobes singlestep? https://lore.kernel.org/all/161469874601.49483.1198532588716692107 6.stgit@devnote2/T/#mbb8fd3431b354681310a12741adfd57fad0e7d95, 2021. Accessed December 26, 2023.

[16] PAYNE, B. D., CARBONE, M., SHARIF, M., AND LEE, W. Lares: An Architecture for Secure Active Monitoring Using Virtualization. In *Proceedings of the 2008 IEEE Symposium on Security and Privacy (SP'08)* (2008).

[17] PRASAD, V., COHEN, W., EIGLER, F., HUNT, M., KENISTON, J., AND CHEN, B. Locating System Problems Using Dynamic Instrumentation. In *Proceedings of the 2005 Ottawa Linux Symposium* (2005).

[18] QUYNH, N. A., AND SUZAKI, K. Xenprobes, A Lightweight User-space Probing Framework for Xen Virtual Machine. In *Proceedings of the 2007 USENIX Annual Technical Conference (ATC'07)* (2007).

[19] REEVES, J., RAMASWAMY, A., LOCASTO, M., BRATUS, S., AND SMITH, S. Intrusion detection for resource-constrained embedded control systems in the power grid. *International Journal of Critical Infrastructure Protection 5*, 2 (2012), 74–83.

[20] REN, X. J., RODRIGUES, K., CHEN, L., VEGA, C., STUMM, M., AND YUAN, D. An Analysis of Performance Evolution of Linux's Core Operations. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles (SOSP'19)* (2019).

[21] SUSE. System analysis and tuning guide. https://documentation.suse.com/sles/15-SP3/html/SLES-all/book-tuning.html, 2024. Accessed 2024.