Arbitrary Instruction Tracing with DTrace

Christos Margiolis
christos@FreeBSD.org

***Abstract*—This paper presents the high-level ideas behind `kinst`, a new DTrace provider available in FreeBSD, which enables tracing of arbitrary instructions and inline functions in the kernel.**

***Index Terms*—FreeBSD, DTrace, tracing, kernel, assembly, ELF, DWARF**

I. Introduction

DTrace [1] is a dynamic tracing framework that originated in Solaris in 2005 and was later ported to FreeBSD. It is used for profiling, performance measurement and debugging in real-time. DTrace is especially powerful thanks to the D language; a C and AWK-like scripting language used for writing DTrace scripts, offering a wide range of built-in functions.

In DTrace terminology, a "provider" is a module that performs a particular instrumentation in the kernel (and sometimes userland), for example, tracing a syscall or the entry point of a function. A "probe" is the specific point of instrumentation.

`kinst` is a new DTrace provider that was created with the aim of solving the limitations of the FBT (Function Boundary Tracing) provider, that is, absence of inline function tracing, and being unable to perform more fine-grained tracing.

The source code for `kinst` lives under `sys/cddl/dev/kinst` [2]. It is available from FreeBSD-14.0 onwards and is supported on amd64, arm64 and riscv.

The origin of the name `kinst` is inspired from an early paper written by A. Tamches and B. Miller [3] discussing a tracing tool they developed called "KernInst".

II. Usage

The following three different syntaxes can be used in kinst:

- `kinst::<function>:`
- `kinst::<function>:<instruction>`
- `kinst::<inline_function>:<entry|return>`

The first syntax will trace all instructions in `<function>`. The second will trace the specific instruction at the offset specified by `<instruction>`, which can be obtained from the function's disassembly. The third will trace either the entry or return point of an inline function.

Below is an example of inline tracing with `kinst`, which traces the return points of all inline copies of `critical_enter()`:

```
# dtrace -n 'kinst::critical_enter:return'
dtrace: description 'kinst::critical_enter:
   return' matched 130 probes
CPU     ID                    FUNCTION:NAME
  1  71024                 spinlock_enter:53
  0  71024                 spinlock_enter:53
  1  70992                   uma_zalloc_arg:49
  1  70925     malloc_type_zone_allocated:21
  1  70994                  uma_zfree_arg:365
  1  70924              malloc_type_freed:21
  1  71024                 spinlock_enter:53
  0  71024                 spinlock_enter:53
  0  70947          _epoch_enter_preempt:122
  0  70949            _epoch_exit_preempt:28
^C
```

Listing 1. Inline tracing with kinst

III. Instruction instrumentation

Probe information is passed from `dtrace(1)` to `kinst(4)` through libdtrace using a character device file in `/dev/dtrace/kinst`. `kinst(4)` is responsible for locating the function in the kernel, disassembling it, creating a private probe structure with metadata about the instruction and the probe, and creating DTrace probes for each of the target instructions. To achieve this, the original instruction is saved in the probe structure for later use, and overwritten with a breakpoint instruction. Once the CPU hits the DTrace-installed breakpoint, the kernel enters DTrace through its trap handler (see `sys/$ARCH/$TARGET_ARCH/trap.c`), and since the breakpoint was installed by `kinst`, the kernel eventually enters `kinst_invop()`, the function responsible for tracing the instruction, either by emulating it in software (similar to FBT), or executing it in a trampoline.

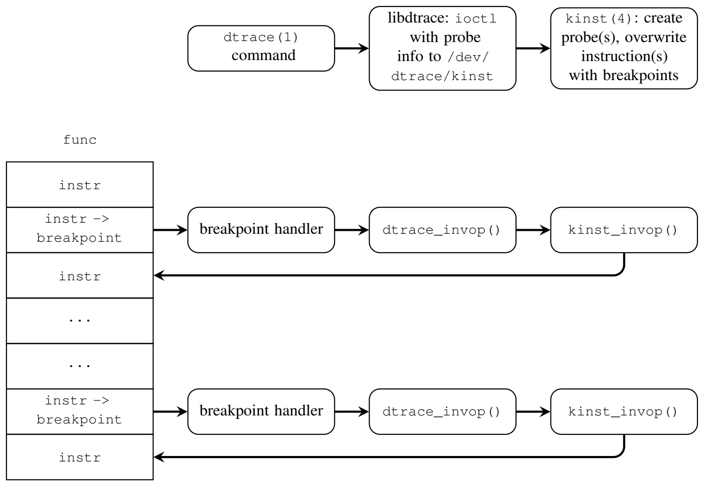


Fig. 1. General pipeline.

All architectures supported by `kinst` have instructions and functions that are unsafe to trace, which are listed in the man page.

IV. Trampoline

Trampolines are executable blocks of memory, which `kinst` uses to instrument instructions, in contrast to emulating them in software. The target instruction is copied to the trampoline and execution is transferred to it once the kernel enters `kinst`. After the trampoline has executed the instruction, the kernel resumes execution normally.

The need for executing instructions in a trampoline arose from the fact that, unlike FBT, which can trace only the "entry" and "return" instructions of a function, `kinst` can trace almost all instructions of an ISA. This, as is expected, makes writing emulation code for all instructions and architectures a rather tedious and error prone task.

A. Under the hood

`kinst` keeps an internal `TAILQ(9)` of memory "chunks" of size `PAGE_SIZE`, logically divided into individual trampolines using `BITSET(9)`. Memory is allocated using `vm_map_find (9)` with `VM_PROT_EXECUTE` and is located above `KERNBASE` on amd64, and `VM_MIN_KERNEL_ADDRESS` on arm64 and riscv. As shown in Figure 2, the trampoline contains the target instruction, followed by a breakpoint on arm64 and riscv, or a jump instruction on amd64, which is used to return back from the trampoline and resume execution (see Sections IV-B and IV-C).

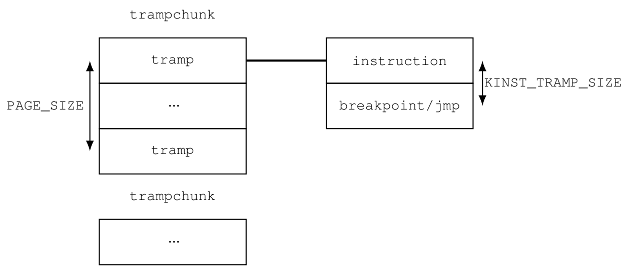


Fig. 2. Trampoline layout.

B. amd64 implementation notes

As mentioned in section IV-A, the trampoline in the amd64 port of `kinst` contains the target instruction, followed by a far-jump to the instruction following the one that was traced, in order to resume execution.

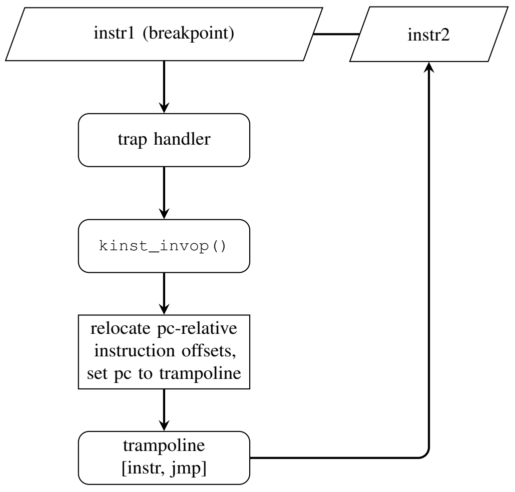


Fig. 3. amd64 control flow.

Each thread and CPU is assigned a trampoline, whose contents are rewritten upon every instrumentation. The reason for having both per-thread and per-CPU trampolines, is that per-thread trampolines cannot be used safely when the thread executing prior to the breakpoint had interrupts disabled. This mechanism, although memory-efficient, has proved to be quite bug-prone when run in VMs with more than one vCPUs, because of the constant fetching and rewriting whenever the kernel enters `kinst_invop()`. Work is being done to replace per-thread/per-CPU trampolines with per-probe trampolines, as is the case in arm64 and riscv.

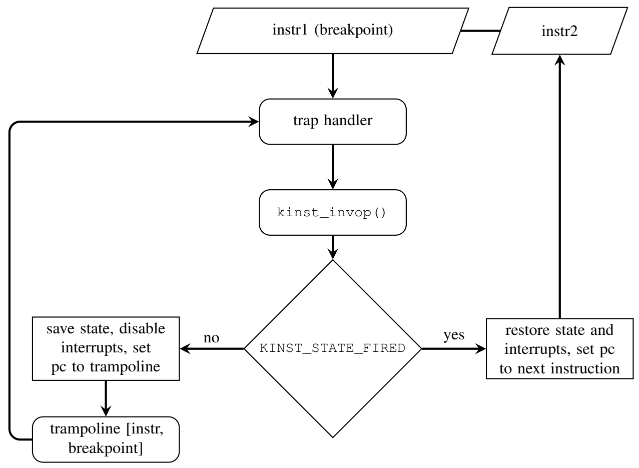
When copying an instruction to the trampoline, `kinst` has special handling for RIP-relative instructions, whose displacements have to be re-encoded to be relative to the trampoline in order to be executed in a trampoline.

Additionally, `call` instructions have to be emulated in assembly because re-encoding a trampoline-relative offset requires reserving space in the stack (see `bp_call` label in `sys/cddl/dev/dtrace/amd64/dtrace_asm.S`).

C. arm64 and riscv implementation notes

In contrast to amd64, it is not possible to encode a far-jump in a single instruction in arm64 and riscv, and is also not possible to clobber registers inside the trampoline to encode a two-instruction sequence to achieve a far-jump. An alternative approach to return from the trampoline in this case, is to use a breakpoint instruction, instead of a far-jump.

Once the trampoline instruction has executed and the CPU hits the breakpoint, the kernel will jump back to the trap handler, and re-enter `kinst_invop()` (see Section III). `kinst` keeps an internal per-CPU state structure to differentiate between breakpoints that were triggered from the trampoline and ones that did not (i.e., triggered by a probe that fired). In the former case, `kinst_invop()` saves the current CPU state register, disables interrupts so that the thread is not interrupted for the duration of the "double-breakpoint" execution, and sets the program counter to the trampoline. In the latter case, that is, when we have returned from the trampoline and it is time to resume execution, `kinst_invop()` restores the CPU state register and interrupts and manually sets the program counter to the instruction following the one that was traced. Figure 4 shows the control flow of this mechanism.



instr1 (breakpoint)
instr2
trap handler
kinst_invop()
KINST_STATE_FIRED
no
yes
save state, disable interrupts, set pc to trampoline
restore state and interrupts, set pc to next instruction
trampoline [instr, breakpoint]

Fig. 4. arm64 & riscv control flow.

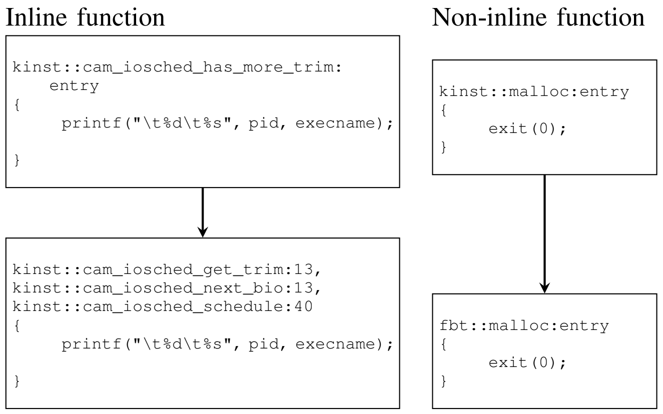
Each probe is assigned its own trampoline and written at instruction parse-time, which, as opposed to amd64, avoids the need to fetch and rewrite per-thread/per-CPU trampolines every time `kinst_invop()` executes.

V. Inline function tracing

Inline tracing in `kinst` is nothing more than a syntactic layer added to libdtrace. When the user requests a probe of the form `kinst::<function>:<entry|return>`, libdtrace parses the DWARF information of all loaded kernel modules to find if `<function>` is an inline, in which case it creates regular `kinst` probes at the exact offset(s) the inline copy's entry or return [1] point(s) is found. However, if `<function>` is not an inline, the probe is automatically converted to an FBT one, to avoid code duplication. Figure 5 showcases an example of how libdtrace handles both inline and non-inline functions.

More information can be found on Christos Margiolis' website [5] [6].

[1]Unlike FBT, `kinst` cannot accept an empty probe field when in "inline tracing mode", so the user must specify either an `entry` or `return` probe. This is a deliberate design choice to differntiate between inline tracing and regular mode.



```
kinst::cam_iosched_has_more_trim:
    entry
{
     printf("\t%d\t%s", pid, execname);

}
```

```
kinst::cam_iosched_get_trim:13,
kinst::cam_iosched_next_bio:13,
kinst::cam_iosched_schedule:40
{
     printf("\t%d\t%s", pid, execname);

}
```

```
kinst::malloc:entry
{
     exit(0);
}
```

```
fbt::malloc:entry
{
     exit(0);
}
```

Fig. 5. Syntactic transformations implemented in `cddl/contrib/opensolaris/lib/libdtrace/common/dt_sugar .c`

A. DWARF overview

DWARF is a debugging standard used by compilers and debuggers, which contains all sorts of information (e.g. function names, variable locations, etc.) about a compilation unit. Debugging information is represented as a tree of entries called DIEs (Debugging Information Entry), one per compilation unit. Each entry consists of various attributes, such as its name, location in memory, declaration file, and more. [2]

Suppose the user runs the following DTrace command:

```
# dtrace -n 'kinst::vfs_freevnodes_dec:entry'
```

libdtrace has to first detect whether `vfs_freevnodes_dec()` is an inline function, meaning finding the following entry:

```
<1><1dfa144>: Abbrev Number: 94 (
  DW_TAG_subprogram)
   <1dfa145>   DW_AT_name        : (indirect
  string) vfs_freevnodes_dec
   <1dfa149>   DW_AT_decl_file   : 1
   <1dfa14a>   DW_AT_decl_line   : 1447
   <1dfa14c>   DW_AT_prototyped  : 1
   <1dfa14c>   DW_AT_inline      : 1
```

Listing 2. Inline function declaration DIE

The type of the entry is specified in `DW_TAG_*` and the attributes in `DW_AT_*`. In this case the tag `DW_TAG_subprogram` means that this entry represents a function and the `DW_AT_inline` attribute that the function is inlined. Now that libdtrace knows the function is inlined, it needs to find the DIEs of each inline copy, such as the following:

```
<3><1dfe45e>: Abbrev Number: 24 (
  DW_TAG_inlined_subroutine)
   <1dfe45f>   DW_AT_abstract_origin: <0
  x1dfa144>
   <1dfe463>   DW_AT_low_pc      : 0
  xffffffff80cf701d
   <1dfe46b>   DW_AT_high_pc     : 0x38
   <1dfe46f>   DW_AT_call_file   : 1
   <1dfe470>   DW_AT_call_line   : 3458
   <1dfe472>   DW_AT_call_column : 5
```

Listing 3. Inline function copy DIE

[2]DWARF entries can be inspected by running `readelf -wi file` or using dwarfdump(1).

Inline copies are defined with the `DW_TAG_inlined_subroutine` tag. The `DW_AT_abstract_origin` attribute specifies the DIE offset corresponding to the function declaration — in this case `0x1dfa144` points to the declaration of `vfs_freevnodes_dec()` (see above).

B. Calculating call boundaries

DWARF defines call bounadaries with either of the following two ways:

- By specifying the `DW_AT_low_pc` and `DW_AT_high_pc` attributes.
- By specifying the `DW_AT_ranges` attribute when the address range of the inline copy is not contiguous.

In the first case, the lower boundary of the inline copy's location is defined in `DW_AT_low_pc` and the upper boundary is calculated by adding `DW_AT_high_pc` to `DW_AT_low_pc`:

```
<3><1dfe45e>: Abbrev Number: 24 (
   DW_TAG_inlined_subroutine)
    <1dfe45f>   DW_AT_abstract_origin: <0
   x1dfa144>
    <1dfe463>   DW_AT_low_pc      : 0
   xffffffff80cf701d
    <1dfe46b>   DW_AT_high_pc     : 0x38
    <1dfe46f>   DW_AT_call_file   : 1
    <1dfe470>   DW_AT_call_line   : 3458
    <1dfe472>   DW_AT_call_column : 5
```

Listing 4. DIE with low and high PC boundaries

Using the formula mentioned above, we end up with the following boundaries:

$$\begin{aligned} L &= \texttt{0xffffffff80cf701d} \\ H &= \texttt{0xffffffff80cf701d} + \texttt{0x38} = \\ &\quad \texttt{0xffffffff80cf7055} \end{aligned} \tag{1}$$

The second case is a bit more involved. Consider the following DIE of an inline copy of `vfs_freevnodes_dec()` which has `DW_AT_ranges`:

```
<3><1dfd2e2>: Abbrev Number: 58 (
   DW_TAG_inlined_subroutine)
    <1dfd2e3>   DW_AT_abstract_origin: <0
   x1dfa144>
    <1dfd2e7>   DW_AT_ranges      : 0x1f1290
    <1dfd2eb>   DW_AT_call_file   : 1
    <1dfd2ec>   DW_AT_call_line   : 3405
    <1dfd2ee>   DW_AT_call_column : 3
```

Listing 5. DIE with `DW_AT_ranges` boundaries

The `DW_AT_ranges` attribute refers to the `.debug_ranges` section found in debug files, and can be inspected by running `dwarfdump -N file`. Searching for the `DW_AT_ranges` value `0x1f1290` specified in the DIE shown above, we find the following range group:

```
Ranges group 38809:
ranges: 3 at .debug_ranges offset 2036368 (0
   x001f1290) (48 bytes)
[ 0] range entry    0x000025c8 0x000025f9
[ 1] range entry    0x0000261a 0x00002621
[ 2] range end      0x00000000 0x00000000
```

Listing 6. `DW_AT_ranges` example

All "range entry" lines correspond to the different address ranges the inline copy is split into, usually as a result of having early returns. The call boundaries are calculated by adding each range entry's values to the lower boundary (`DW_AT_low_pc`) of the root DIE, that is, the file the inline function is implemented in:

```
<0><1dee9fb>: Abbrev Number: 1 (
   DW_TAG_compile_unit)
    <1dee9fc>   DW_AT_producer    : (indirect
   string) FreeBSD clang version 13.0.0 (
   git@github.com:llvm/llvm-project.git
   llvmorg-13.0.0-0-gd7b669b3a303)
    <1deea00>   DW_AT_language    : 12  (C99)
    <1deea02>   DW_AT_name        : (indirect
   string) /usr/src/sys/kern/vfs_subr.c
    <1deea06>   DW_AT_stmt_list   : 0x6cb448
    <1deea0a>   DW_AT_comp_dir    : (indirect
   string) /usr/obj/usr/src/amd64.amd64/sys/
   GENERIC
    <1deea0e>   DW_AT_low_pc      : 0
   xffffffff80cf4020
    <1deea16>   DW_AT_high_pc     : 0xde3d
```

Listing 7. Compilation unit DIE

Adding the range entries to the lower boundary of the root DIE (`0xffffffff80cf4020`), we get the following inline copy call boundaries:

First range entry:

$$\begin{aligned} L &= \texttt{0xffffffff80cf4020} + \texttt{0x000025c8} = \\ &\quad \texttt{0xffffffff80cf65e8} \\ H &= \texttt{0xffffffff80cf4020} + \texttt{0x000025f9} = \\ &\quad \texttt{0xffffffff80cf6619} \end{aligned} \tag{2}$$

Second range entry:

$$\begin{aligned} L &= \texttt{0xffffffff80cf4020} + \texttt{0x0000261a} = \\ &\quad \texttt{0xffffffff80cf663a} \\ H &= \texttt{0xffffffff80cf4020} + \texttt{0x00002621} = \\ &\quad \texttt{0xffffffff80cf6641} \end{aligned} \tag{3}$$

C. Finding the caller function

Having acquired the call boundaries of the inline copies, libdtrace can proceed to find the name of the caller function, which is needed to create a `kinst` probe. This is done by finding which ELF symbol the call boundaries are inside of, as expressed by the following condition:

$$Sym_{lo} \leq Inl_{lo} \leq Inl_{hi} \leq Sym_{hi}$$

`inlinecall(1)` [5] [7] is a small utility written as a testing program which implements the mechanism explained so far. It takes an inline function and a debug file as arguments, and prints the call boundaries of all inlines copies found, as well as the caller file and function.

D. *Calculating the `entry` and `return` offsets*

This is the last step libdtrace has to go through to convert the inline function probe to actual `kinst` probes. Here it fetches the boundaries of the caller function from ELF, and, using the boundaries of the inline copy, it calculates the exact offsets for either the entry or return probes.

In theory, the entry and return offsets would simply be calculated as:

$$Entry = Inl_{lo} - Caller_{lo}$$

$$Return = Inl_{hi} - Caller_{lo}$$

However, it turns out that there are multiple caveats we have to take into consideration, but the details are out of the scope of this paper (see [6] for an explanation).

These offsets are then used used to create regular `kinst` probes of the form `kinst::<function>:<instruction>`, which is what `kinst` actually expects:

```
# dtrace -dn 'kinst::cam_iosched_has_more_trim:
    entry'
kinst::cam_iosched_get_trim:13,
kinst::cam_iosched_next_bio:13,
kinst::cam_iosched_schedule:40
{
}

dtrace: description 'kinst::
    cam_iosched_has_more_trim:entry ' matched 4
     probes
CPU     ID                    FUNCTION:NAME
  0  81502          cam_iosched_schedule:40
  0  81501          cam_iosched_next_bio:13
  2  81502          cam_iosched_schedule:40
  1  81502          cam_iosched_next_bio:13
  1  81503          cam_iosched_schedule:40
^C
```

Listing 8. Inline to regular probe conversion

VI. Conclusion & Future work

`kinst`, although a low-level tool, can be used in a variety of workflows, especially when working with very large functions, where tracing the entry or return points of them does not always provide adequate answers.

Future work includes writing more high-level tooling around `kinst`, as well as detecting and putting return probes on tail-call optimized functions.

I would like to thank Mark Johnston `<markj@FreeBSD.org>` and Mitchell Horne `<mhorne@FreeBSD.org>` for their valuable help and feedback.

References

[1] Illumos Operating System "Dynamic Tracing Guide". https://illumos.org/books/dtrace
[2] FreeBSD src "kinst" https://cgit.freebsd.org/src/tree/sys/cddl/dev/kinst
[3] Tamches, Ariel & Miller, Barton P. "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels". https://www.usenix.org/legacy/publications/library/proceedings/osdi99/full_papers/tamche
[4] The DWARF Standard. https://dwarfstd.org/
[5] Christos Margiolis "Using DWARF to find call sites of inline functions". https://margiolis.net/w/dwarf_inline/
[6] Christos Margiolis "Inline function tracing with the kinst DTrace provider". https://margiolis.net/w/kinst_inline/
[7] GitHub, inlinecall(1). https://github.com/christosmarg/inlinecall