Arbitrary Instruction Tracing with DTrace

Christos Margiolis
christos@FreeBSD.org

September 17, 2023
EuroBSDCon 2023 — Coimbra, Portugal

Who?

- Christos Margiolis <christos@FreeBSD.org>
- FreeBSD contributor since 2020, `src` committer since 2023.
- Have worked mainly on DTrace.
- Terrible at introductions.

DTrace quick background

- Dynamic tracing framework.
- Originated in Solaris in 2005.
- Ability to observe kernel behavior (e.g function calls) in real-time.
- Provider: Module that performs a particular instrumentation in the kernel.
- Probe: Specific point of instrumentation.
- D language.
- https://illumos.org/books/dtrace

The FBT provider

- Trace the entry and return points of a kernel function.
- Cannot trace specific instructions and inline functions.

```
# dtrace -n 'fbt::malloc:entry {printf("%s", execname);}'
dtrace: description 'fbt::malloc:entry ' matched 1 probe
CPU     ID                    FUNCTION:NAME
  3  30654                     malloc:entry dtrace
  0  30654                     malloc:entry pkg
  1  30654                     malloc:entry Xorg
  3  30654                     malloc:entry firefox
  2  30654                     malloc:entry zfskern
  3  30654                     malloc:entry kernel
^C
```

The kinst provider

- Inspired by FBT.
- Trace arbitrary machine instructions in a kernel function.
- Can trace inline functions.
- More fine-grained tracing (specific if statements, loops, branches, ...). Requires good C-to-Assembly translation skills.
- Available on amd64, arm64 and riscv.
- In the future: build higher-level tooling, detect and put return probes on tail-call optimized functions.
- `sys/cddl/dev/kinst/`

kinst::<function>:

```
# dtrace -n 'kinst::amd64_syscall:'
dtrace: description 'kinst::amd64_syscall:' matched 458
    probes
CPU     ID                    FUNCTION:NAME
  2  80676                amd64_syscall:323
  2  80677                amd64_syscall:326
  2  80678                amd64_syscall:334
  2  80679                amd64_syscall:339
  2  80680                amd64_syscall:345
  2  80681                amd64_syscall:353
  ^C
```

kinst::<function>:<instruction>

```
# kgdb
(kgdb) disas /r vm_fault
Dump of assembler code for function vm_fault:
   0xffffffff80876df0 <+0>:     55       push   %rbp
   0xffffffff80876df1 <+1>:     48 89 e5         mov     %
    rsp,%rbp
   0xffffffff80876df4 <+4>:     41 57    push   %r15

# dtrace -n 'kinst::vm_fault:4 {printf("%#x", regs[R_RSI])
    ;}'
  2  81500                    vm_fault:4 0x827c56000
  2  81500                    vm_fault:4 0x827878000
  2  81500                    vm_fault:4 0x1fab9bef0000
  2  81500                    vm_fault:4 0xe16cf749000
  0  81500                    vm_fault:4 0x13587c366000
  ^C
```

kinst::<inline_func>:<entry|return>

```
# 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
  0  71024               spinlock_enter:53
  0  70947          _epoch_enter_preempt:122
  0  70949            _epoch_exit_preempt:28
  0  71024               spinlock_enter:53
  0  71024               spinlock_enter:53
  0  70947          _epoch_enter_preempt:122
  0  70949            _epoch_exit_preempt:28
  ^C
```

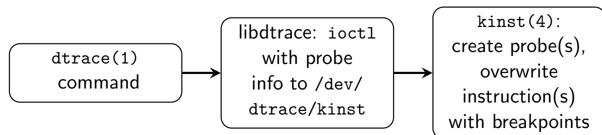
High-level ideas

- How are instructions instrumented?
- Architecture-dependent code.
- Inline function tracing.

Instruction instrumentation

- Probe information is passed from `dtrace(1)` to libdtrace to `kinst(4)` using a character device file in `/dev/dtrace/kinst`.
- kinst disassembles the function and creates probes for each of the target instructions.
- The original instruction is overwritten with a breakpoint instruction.
- When the CPU hits the breakpoint, we jump into `kinst_invop()` through the trap handler.
- kinst decides if the instruction is to be emulated or executed in a trampoline.
- Trace the instruction and continue execution.

Instruction instrumentation



func

instr
instr -> breakpoint
instr
...
...
instr -> breakpoint
instr

breakpoint handler → dtrace_invop() → kinst_invop()

breakpoint handler → dtrace_invop() → kinst_invop()

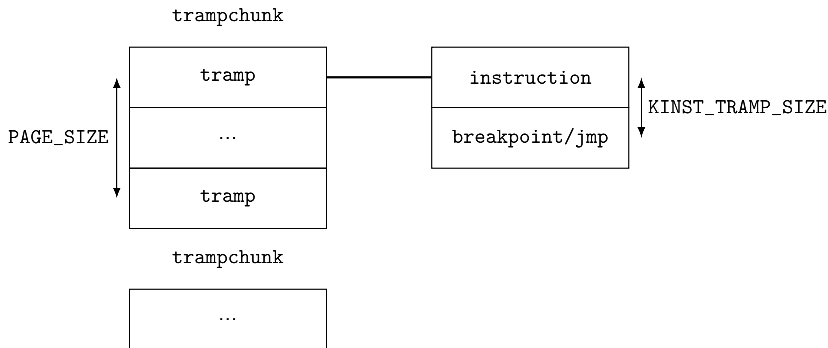
Trampoline: Overview

- Emulating every single instruction for each architecture is tedious and error prone.
- Target instruction is copied there and execution is transferred to the trampoline manually.
- How do we return back?

Trampoline: Under the hood

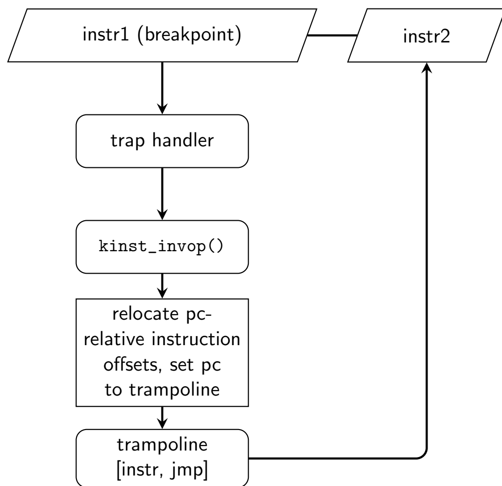
- Executable block of memory.
- Memory "chunks" of size `PAGE_SIZE` stored in a `TAILQ`
- `vm_map_find(9)` with `VM_PROT_EXECUTE`, `vm_map_remove(9)`, `kmem_back(9)`, `kmem_unback(9)`, `malloc(9)`.
- Allocated above `KERNBASE` (amd64), or `VM_MIN_KERNEL_ADDRESS` (rest).
- Logically divided into individual trampolines using `BITSET(9)`.
- `kinst_trampoline_alloc()` finds and returns the next free trampoline.

Trampoline: Under the hood



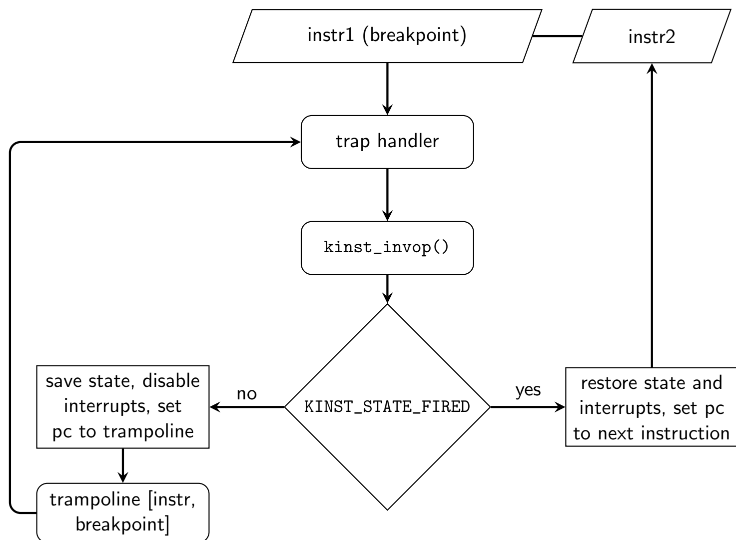
- amd64: Per-thread and per-CPU trampolines that are rewritten upon every instrumentation. To be deprecated.
- arm64 and riscv: Per-probe trampolines.

Trampoline: amd64 control flow (to be deprecated)



The per-thread/per-CPU trampolines are rewritten upon every instrumentation. Will be replaced by the arm64/riscv mechanism and use per-probe trampolines to avoid race bugs.

Trampoline: arm64/riscv control flow



Synchronization is done through a `DPCPU(9)` `kinst_cpu_state` structure.

Caveats

- amd64
  - The ISA is... complicated. Instruction parsing is tedious.
  - RIP-relative instructions have to have their displacements re-encoded to be relative to the trampoline in order to be executed in a trampoline.
  - `call` instructions have to be emulated in assembly (see `bp_call` label in `sys/cddl/dev/dtrace/amd64/dtrace_asm.S`).
- arm64, riscv
  - Unlike amd64, encoding trampoline-relative offsets for PC-relative instructions is not possible in a single instruction, so all PC-relative instructions have to be emulated in `kinst_emulate()`.
- Some functions and instructions are unsafe to trace (listed in man page).

Inline function tracing

- Syntax: `kinst::<inline_func>:<entry>`
- All the hard work is done in libdtrace, instead of `kinst(4)`.
- Uses the DWARF and ELF standards.
  - If the function is an inline, libdtrace calculates the function boundaries and offsets and creates regular kinst probes for each one of the inline copies found.
  - If the function is *not* an inline, the probe is converted to an FBT one, to avoid code duplication.
- Done for each loaded kernel module. Painfully slow...
- Can handle nested inline functions.
- `cddl/contrib/opensolaris/lib/libdtrace/common/dt_sugar.c`
- https://margiolis.net/w/dwarf_inline/
- https://margiolis.net/w/kinst_inline/

Inline function tracing

Inline function

```
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);
}
```

Non-inline function

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

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

Inline function tracing: DWARF overview

- Debugging information is represented as a tree of entries, one per compilation unit. The entries correspond to functions, variables, arguments, etc and each entry has various attributes (name, location, ...).
- Functions that get inlined have an "inlined" attribute set.
- The inline copy entries point to the declaration entry's offset and include information about the copy's lower and upper boundaries.

Inline function tracing: DWARF overview

Inline function declaration 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
```

Inline copy entry:

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

Inline function tracing: Finding the caller function

- Can fetch the upper and lower boundaries of the inline copy from DWARF.
- Need to know which function the inline copy is inlined in and at what offset, so we can create kinst probes. Done by scanning ELF symbol tables.

The name of the caller function corresponds to the name of the symbol satisfying the following condition:

$$Sym_{lower} \leq Inl_{lower} \leq Inl_{upper} \leq Sym_{upper}$$

Then, the entry and return* offsets are calculated as:

$$Entry = Inl_{lower} - Sym_{lower}$$

$$Return = Inl_{upper} - Sym_{lower}$$

* Not exactly that simple...

Acknowledgments

Mark Johnston <markj@FreeBSD.org>
Mitchell Horne <mhorne@FreeBSD.org>

Questions/Suggestions

- What feature(s) would you like kinst to have?
- Is there a particular use case you want to achieve with kinst?