SDC STORAGE DEVELOPER CONFERENCE

SNIA SANTA CLARA, 2017

RALPH BÖHME, SERNET, SAMBA TEAM

UNDERSTANDING AND IMPROVING SAMBA FILESERVER PERFORMANCE

HOW I FELL IN LOVE WITH SYSTEMTAP AND PERF

AGENDA

- Disclaimer: focus on userspace, not kernel, mostly Linux
- Linux tracing history tour de force
- perf
- Systemtap
- Samba fileserver performance improvements

AGENDA

KEY TAKEAWAY...

INTRODUCTION

...LINUX TRACING HAS EVOLVED . . .

INTRODUCTION

TRACING IN THE 90'S

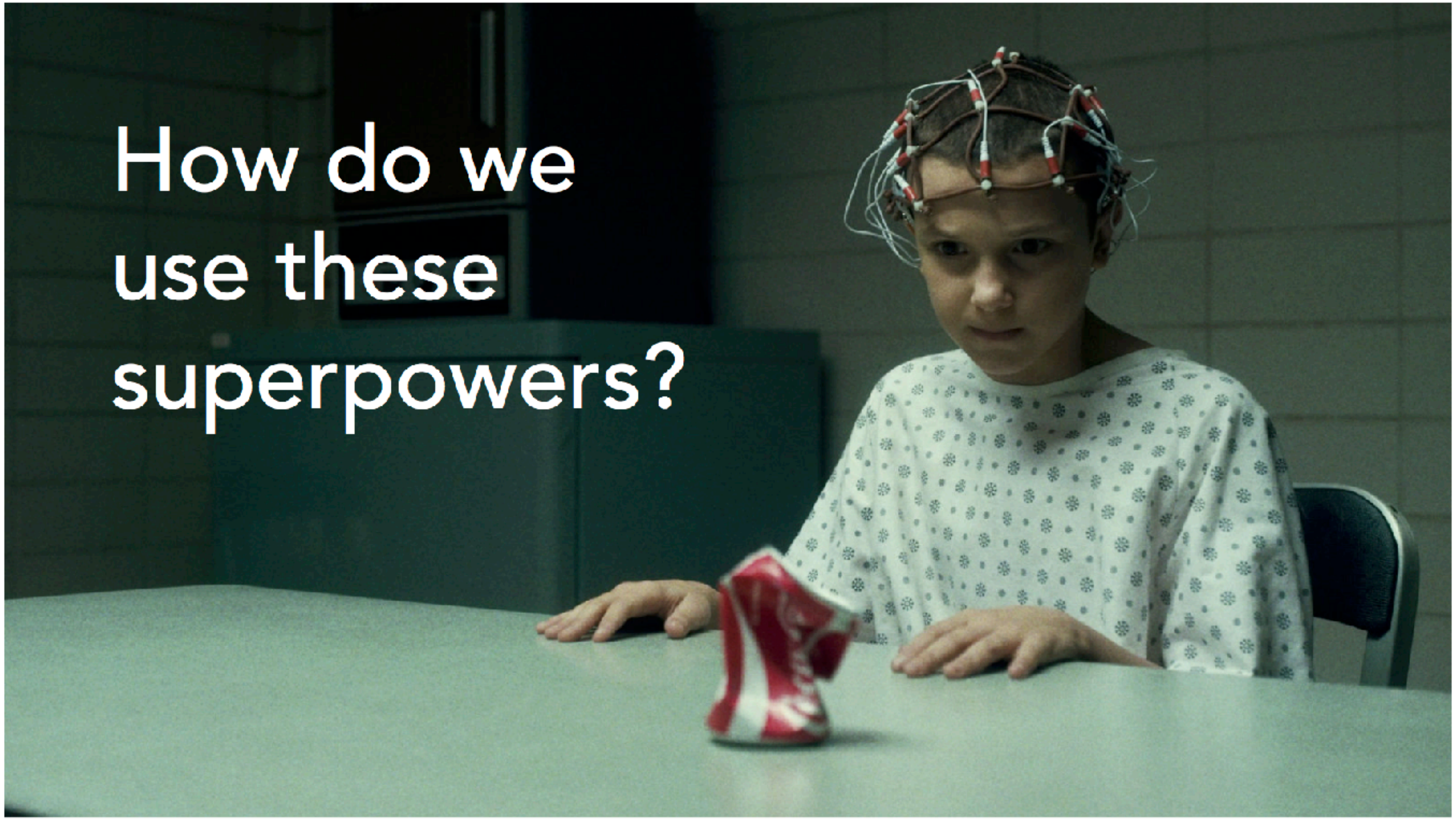
```
          ^__^
          (oo)
  /-------\/
 / |     ||
*  ||----||
   ^^    ^^
```

ptrace

/proc

INTRODUCTION

TRACING TODAY



taken from Brendan Greggs presentation Performance Analysis with bcc/BPF

INTRODUCTION

A LINUX TRACING TIMELINE

1990’s: Static tracers

2000: LTT + DProbes

2004: kprobes

2005: DTrace

2005: SystemTap

2005: LTTng

2008: ftrace

2009: perf_events

2009: Kernel tracepoints

2012: uprobes

2013: ktap

2014: sysdig

2014: eBPF

INTRODUCTION

What can be done:

- Counting CPU events: cycles, branch misses, frontline stalls, ...
- Trace syscalls, but more efficiently
- Trace at the source code level by symbol (function name) or line number (both kernel and userspace)
- Provide stable tracepoint ABI (again kernel and userspace)
- Counting, statistics, latency, histograms...
- Some stuff (BPF, ftrace with hist-triggers, uprobes) requires newer kernels so might not be present on older systems

INTRODUCTION: KERNEL FRAMEWORKS

The whole zoo uses a smaller set of common in-kernel tracing frameworks:

1. **Static tracepoints**
2. **Dynamic tracepoints**: *kprobes* and *uprobes*
3. **perf_events**
4. **BPF** (previously also **E**nhanced **BPF**, aka eBPF)

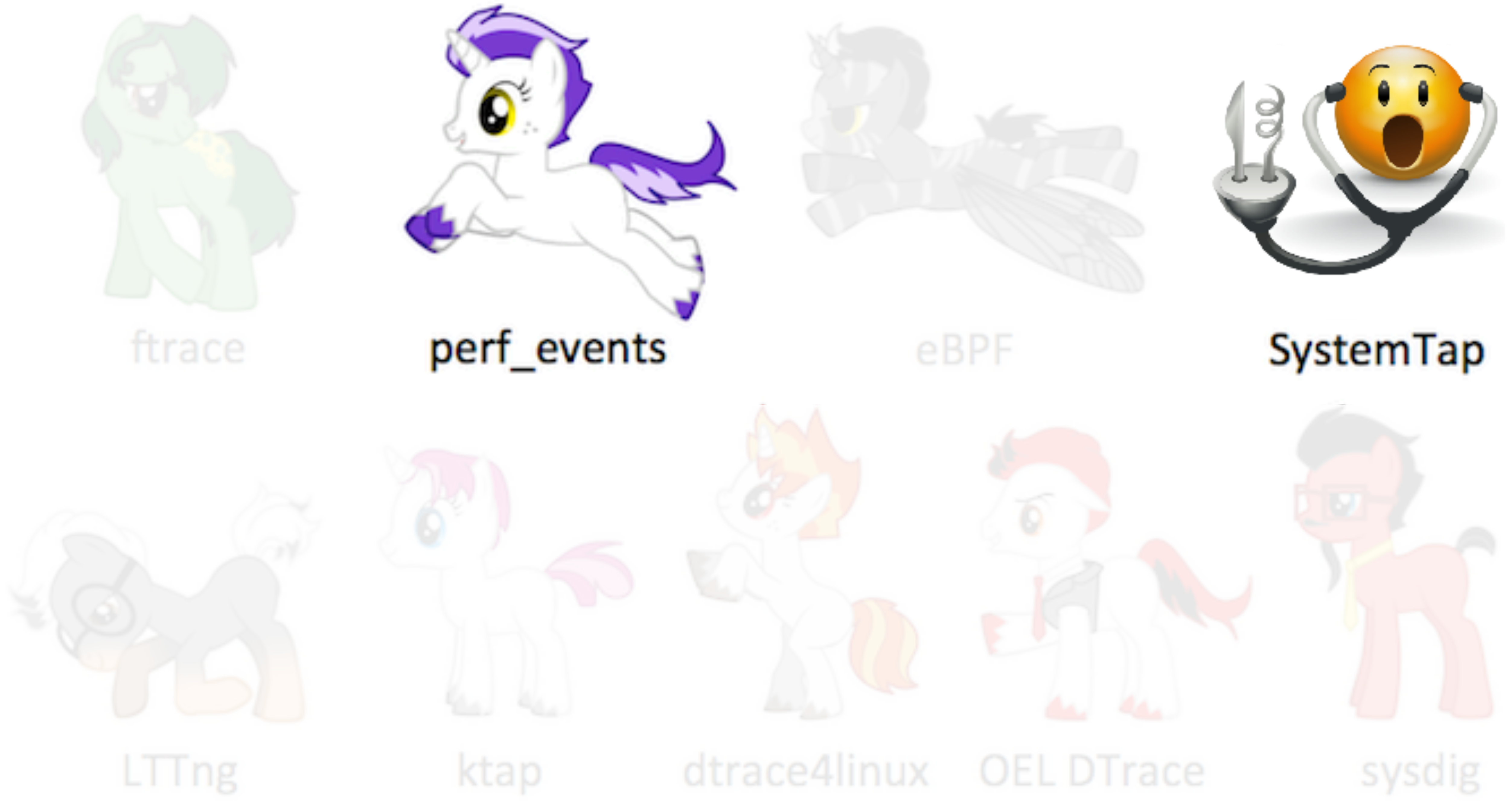
All frameworks incur low overhead when enabled per tracepoint and zero overhead when not enabled – except *uprobes* and *USDT* which take a context switch when firing.

INTRODUCTION: KERNEL FRAMEWORKS AND EVENT TYPES

The types of events are:

- **CPU Hardware Events**: CPU performance monitoring counters (PMU, Performance Monitoring Unit), eg CPU cycles
- **CPU Software Events**: these are low level events based on kernel counters. For example, CPU migrations, minor faults, major faults, branch misses etc.
- **Tracepoint Events**: This are static kernel-level (SDT) or user-level (USDT) instrumentation points that are hardcoded in interesting and logical places in the kernel or applications
- **Dynamic Tracing**: Software can be dynamically instrumented, creating events in any location. For kernel software, this uses the *kprobes* framework. For user-level software, *uprobes*.
- **Timed Events**: commonly used for profiling

INTRODUCTION



INTRODUCTION

INTRODUCTION

BPF/bcc, the new kid on the block:

- **BPF**: (enhanced) Berkeley Packet Filter with, the kernel framework
- **bcc**: BPF compiler collection
- BPF originated as a technology for optimizing packet filters. If you run tcpdump with an expression (matching on a host or port), it gets compiled into optimal BPF bytecode which is executed by an in-kernel sandboxed virtual machine
- Enhanced BPF (aka eBPF, but also just BPF) extended what this BPF virtual machine could do: allowing it to run on events other than packets, and do actions other than filtering

INTRODUCTION

ftrace

- It's been metioned as kernel hacker's best friend, built into the kernel and can consume all the mentioned kernel tracing frameworks
- event tracing, with optional filters and arguments
- until very recently not programmable and no builtin statistics support, changed with the addition of hist-triggers and BPF support

INTRODUCTION

How to choose? For userspace, like Samba:

- Recommendation: choose perf for CPU profiling
- Systemtap for all the rest
- Look at the others when something is missing (unlikely) or you feel like it
- Keep an eye on BPF

perf_events

PERF

PERF

- `perf_events`: a kernel subsystem(s) and a user-space tool
- Counting events & profiling with post-processing
- Not programmable and no builtin statistics and aggregations, though this changed recently
- It can instrument CPU performance counters (PMU), tracepoints, kprobes and uprobes

PERF: PROFILING

Linux profilers:

1. GNU gprof: requires special compilation
2. Valgrind Callgrind: sloooooooooooooooow
3. oprofile, just didn't work in my environment so I looked at:
4. perf

PERF

- Where do you get it?

  # yum install perf
  # apt-get install linux-tools

- When profiling you will want symbols so also install *-debuginfo/ *.dbg package of profiled application

- perf can do much more then profiling, but for me the key selling point is the text-based interactive interface to display the profile info:

  # perf report

PERF TUI DEMO

DEMO

SYSTEMTAP

SYSTEMTAP

- „SystemTap provides a simple command line interface and scripting language for writing instrumentation for a live running kernel plus user-space applications."
- „The essential idea behind … systemtap … is to name *events*, and to give them *handlers*. Whenever a specified event occurs, the Linux kernel runs the handler."
- ***You*** write the event handlers in the Systemtap script language which is C like with type inference, but safe with builtin runtime safety checks

SYSTEMTAP

- The script associates handlers with probes:
  probe EVENT { HANDLER }
- Several varieties of supported events:
  begin, end, timer, syscalls, tracepoints, DWARF, perf_events
- Handler can have filtering, conditionals, variables: primitive (numbers, strings), associative arrays, in kernel statistical aggregations
- Many helper functions: printf, gettimeofday, ...
- The script is translated to C

...continued on next slide...

SYSTEMTAP

...continued from previous slide...

- The C code is compiled to a kernel module
- The kernel loads the module and enables the probes, inserting jumps (kernel) or breakpoints (userspace)
- with DWARF debug symbols you can place probes on file.c:linenumber (kernel or user-space)
- Associative arrays, Statistics (aggregates)
- Probe handlers have access to execution context (variables, parameters)

SYSTEMTAP: LIST AVAILABLE STATIC PROBES

Terminal — 100×30

```
$ # DWARF debug symbols
$ stap -l 'kernel.function("*")' | wc -l
54049

$ # krobes, doesn't require debug symbols
$ stap -l 'kprobe.function("*")' | wc -l
43792

$ # SDT, no debug symbols needed
$ stap -l 'kernel.trace("*")' | wc -l
2203

$ # CPU PMU Hardware/Software
$ stap -l 'perf.*.*' | wc -l
19

# man stapprobes
```

SYSTEMTAP EXAMPLE: HELLO WORLD

Terminal — 100×30

```
# cat hello_world.stp
probe begin {
        printf("Hello world!\n")
}

probe end {
        printf("\nGoodbye!\n")
}

# stap hello_world.stp
Hello world!
^C
Goodbye!
```

SYSTEMTAP EXAMPLE: SYSCALL WITH STATISTICS

Terminal — 100×30

```
# cat pwrite.stp
global bytes_written

probe begin {
        printf("Tracing, press ctrl-c to stop... ")
}

probe syscall.pwrite.return{
   if(pid() == target())
           bytes_written += $return
}

probe end {
        printf("\nTotal bytes written: %d\n", bytes_written)
}

# stap -x 18113 pwrite.stp
Tracing, press ctrl-c to stop... ^C
Total bytes written: 2825879
```

SYSTEMTAP EXAMPLE: USERSPACE FUNCTION, NEEDS DEBUG SYMBOLS

Terminal — 100×30

```
# cat smb2-reqs.stp
global smb2_reqs

probe begin {
        printf("Tracing, press ctrl-c to stop... ")
}

probe process("/.../libsmbd-base-samba4.so").function("smbd_smb2_request_dispatch")
{
   if(pid() == target())
          smb2_reqs++
}

probe end {
        printf("\nGot %d SMB2 requests\n", smb2_reqs)
}
# stap smb2-reqs.stp
Tracing, press ctrl-c to stop... ^C
Got 7163 SMB2 requests
```

SYSTEMTAP EXAMPLE: ADDING USING USDT PROBE TO SAMBA

Terminal — 100×30

```
commit 3caa363dcf41aed3c2e4486d9f77880c3bb140f1
Author:     Ralph Boehme <slow@samba.org>
...
    s3/smbd: add instrumentation for SMB2 request
...
--- a/source3/smbd/smb2_create.c
+++ b/source3/smbd/smb2_create.c
...
@@ -703,6 +705,8 @@ static void reprocess_blocked_smb2_lock(...)
        if (!smb2req->subreq) {
                return;
        }
+
+       SAMBA_PROBE(smb2, request_start, 2, smb2req->smb1req->mid, SMB2_OP_LOCK);
        SMBPROFILE_IOBYTES_ASYNC_SET_BUSY(smb2req->profile);

        state = tevent_req_data(smb2req->subreq, struct smbd_smb2_lock_state);
```

SYSTEMTAP: ANATOMY OF A USDT PROBE

Terminal — 100×30

```
commit cad76c44f6f88caa08ff92d2dea73a120d4e9b59
Author:     Ralph Boehme <slow@samba.org>
...
    libreplace: add Systemtap include wrapper
...
--- /dev/null
+++ b/lib/replace/system/systemtap.h
@@ -0,0 +1,63 @@
+#ifdef HAVE_SYS_SDT_H
+#include <sys/sdt.h>
...
+#define SAMBA_PROBE(provider, probe, n, ...) \
+       SAMBA_PROBE_INTERNAL(provider, probe, n, ## __VA_ARGS__)
+
+#define SAMBA_PROBE_INTERNAL(provider, probe, n, ...) \
+       DTRACE_PROBE##n(provider, probe, ## __VA_ARGS__)
+
+#define DTRACE_PROBE0(provider, probe) \
+       DTRACE_PROBE(provider, probe)
+
+#endif
...
```

SYSTEMTAP EXAMPLE: USING USDT PROBE

Terminal — 100×30

```
# cat smb2-reqs2.stp
global smb2_reqs

probe begin {
        printf("Tracing, press ctrl-c to stop... ")
}

probe process("/.../libsmbd-base-samba4.so").provider("smb2").mark("request_start")
{
   if(pid() == target())
           smb2_reqs++
}

probe end {
        printf("\nGot %d SMB2 requests\n", smb2_reqs)
}

# stap smb2-reqs2.stp
Tracing, press ctrl-c to stop... ^C
Got 8130 SMB2 requests
```

SYSTEMTAP

While at it, let's also add instrumentation to these:

1. tevent events
2. sending / receiving data from the network
3. disk IO
4. syscalls
5. smbd $\Leftrightarrow$ ctdb communication latency

Let me introduce you to **`tsmbd`**:

SYSTEMTAP: TSMBD

Terminal — 100×30

```
# examples/systemtap/tsmbd -h
Trace an smbd process with Systemtap

USAGE: tsmbd [-s|—d|-h] pid

    pid            # trace this process ID
    -d             # show distribution histograms
    -h             # print this help text
#
```

SYSTEMTAP: TSMBD

Terminal — 100×30

```
# examples/systemtap/tsmbd 11327
Compiling Systemtap script, this may take a while...
Collecting data, press ctrl-C to stop... ^C
```

SYSTEMTAP: TSMBD

Terminal — 100×30

```
# examples/systemtap/tsmbd 11327
Compiling Systemtap script, this may take a while...
Collecting data, press ctrl-C to stop... ^C

Ran for:                          38728 ms

Time waiting for events:          32029 ms
Time receiving SMB2 packets:        157 ms
Time running SMB2 requests:        3820 ms
Time sending SMB2 packets:          832 ms
Time waiting for ctdb:                0 ms

Time in syscalls:                  2165 ms
Time in disk IO (read):               9 ms
Time in disk IO (write):             45 ms

Number of tevent events:          29407
Number of SMB2 requests:          26937
Number of ctdb requests:              0
```

SYSTEMTAP: TSMBD

Terminal — 100×30

```
...continued...

SMB2 Requests          Count    Total us      Avg us      Min us      Max us
SMB2_OP_CREATE         8295     2516071       303         65          13378
SMB2_OP_CLOSE          8152      573601        70         19           8218
SMB2_OP_SETINFO        5297      258329        48         19           8154
SMB2_OP_WRITE          2464      333957       135         62           8246
SMB2_OP_GETINFO        2729      125258        45         34           8222

ctdb Requests          Count    Total us      Avg us      Min us      Max us

...continued on next page...
```

SYSTEMTAP: TSMBD SERVICE TIME HISTOGRAM

Terminal — 100×30

```
...SMB2 request service time histogram (optional with tsmbd -d)...

SMB2_OP_CREATE distribution (microseconds)
value |-------------------------------------------------- count
   16 |                                                       0
   32 |                                                       0
   64 |@                                                    289
  128 |@@@@                                                 846
  256 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@  9315
  512 |                                                      48
 1024 |                                                       7
 2048 |                                                       0
 4096 |                                                       1
 8192 |                                                       2
16384 |                                                       0
32768 |                                                       0
...
```

SYSTEMTAP: TSMBD

tsmbd summary:

- Nice and detailed high level overview, it's non-intrusive
- tsmbd is only 356 lines of code, much of that is just boilerplate stuff for the probes
- Tracing too much details (like all syscalls) does have performance impact due to context switching, we may have to make that an option
- Another useful thing is to extend it to trace all SMB2 sessions, currently only one selected by process pid

Canine Craze
Performance Center

SMBD PERFORMANCE IMPROVEMENTS

SAMBA PERFORMANCE IMPROVEMENTS

1. Clustered Samba: directory enumeration
2. Name mangling: new option „mangled names = illegal"
3. Make use of struct smb_filename plumbing in the 4.5 VFS: avoid redundant stats
4. GPFS VFS module improvements: avoid GPFS API calls to fetch creation date
5. Internal messaging improvements: connection caching
6. Exclusive lease optimisations (Samba had this for oplocks but they didn't make it into the lease area): check file handle before looking into the leases database

SAMBA PERFORMANCE IMPROVEMENTS: RESULTS

Small file copy throughput:

- before: 136 files / s
- after: 151 files / s
- ~10% improvement by drilling into existing code with perf TUI

LINKS

WIP Samba instrumentation git branch

Systemtap Beginners Guide

Systemtap Language Reference

Linux perf site by Brendan Gregg

BPF Compiler Collection

Flamegraphs

THE END

THANK YOU!

QUESTIONS?

Ralph Böhme <slow@samba.org>