LEADING
COLLABORATION
IN THE ARM
ECOSYSTEM

Debugging with eBPF on Arm Platforms

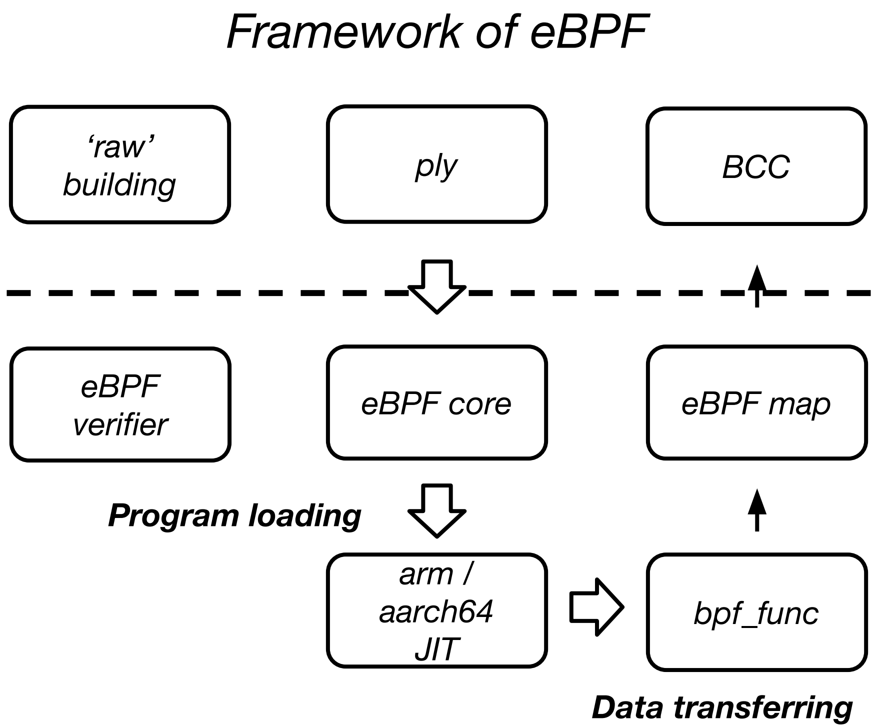
Leo Yan (and Daniel Thompson)

Introduction

eBPF stands for ‘Extended Berkeley Packet Filter’. Classic BPF (cBPF) was used for network packet filtering but now it can be used for much, much more.

We will review what’s the challenges for deployment eBPF on Arm platforms and talk about eBPF tooling. We will conclude the session by discussing two examples.

We will finish this material in 35 minutes.

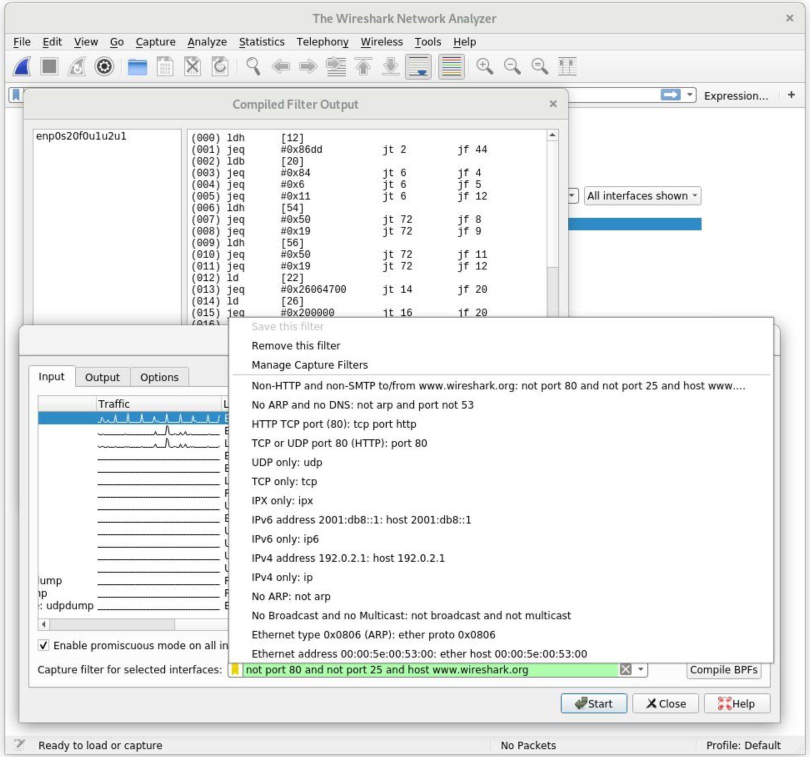


Course outline

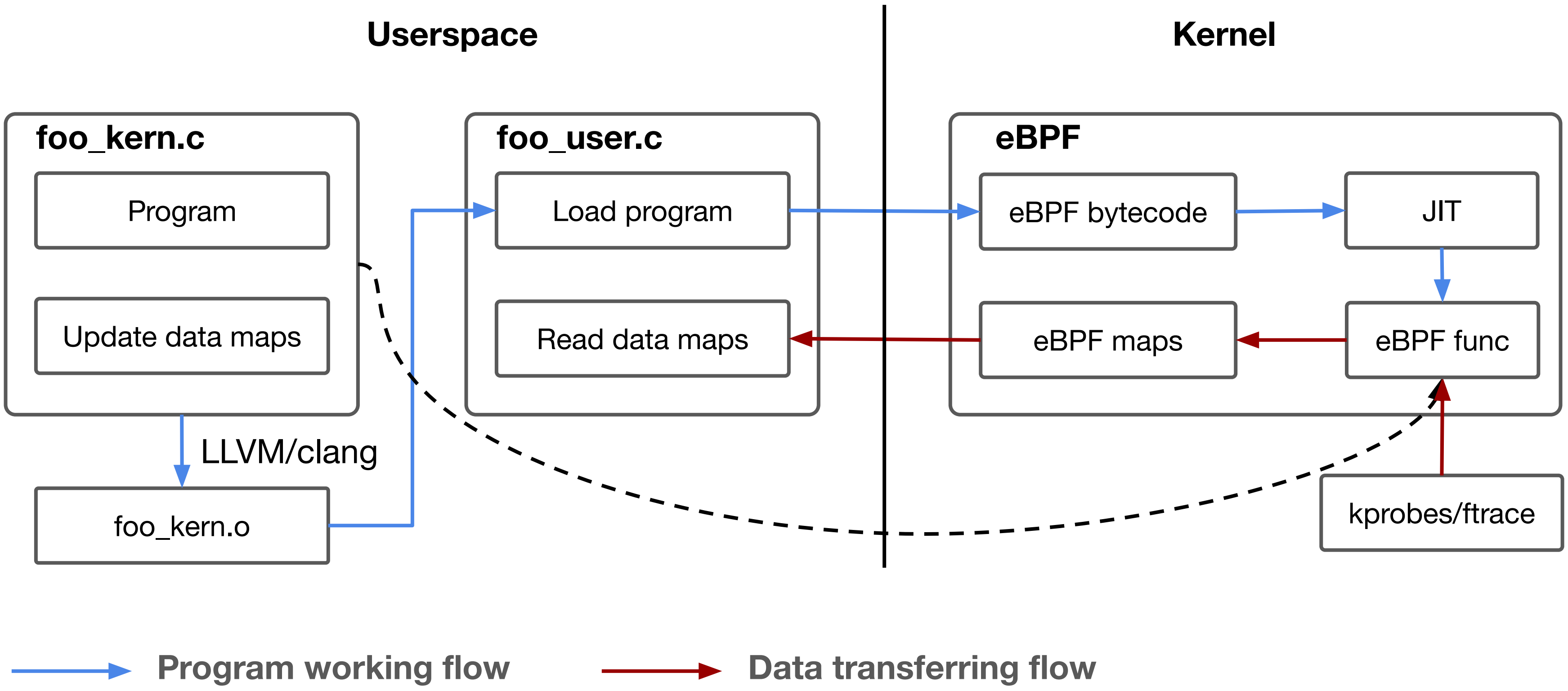
- Using eBPF for debugging
- Coding for eBPF in assembler
- eBPF tools
  - Kernel samples
  - Ply
  - BCC
  - SystemTap (stapbpf)
  - BPFtrace
  - Perf
- Debugging stories

Extending the Berkeley Packet Filter

- Historically Berkeley Packet Filter provided a means to filter network packets
  - If you ever used tcpdump you've probably already used it
  - `tcpdump host beech and \( ash or oak \)`
- eBPF has extended BPF hugely:
  - Re-encoded and more expressive opcodes
  - Multiple new hook points within the kernel to attach eBPF programs to
  - Rich data structures to pass information to/from kernel
  - C functional call interface (an eBPF program can call kernel function)



Using eBPF for debugging



Using eBPF for debugging - cont.

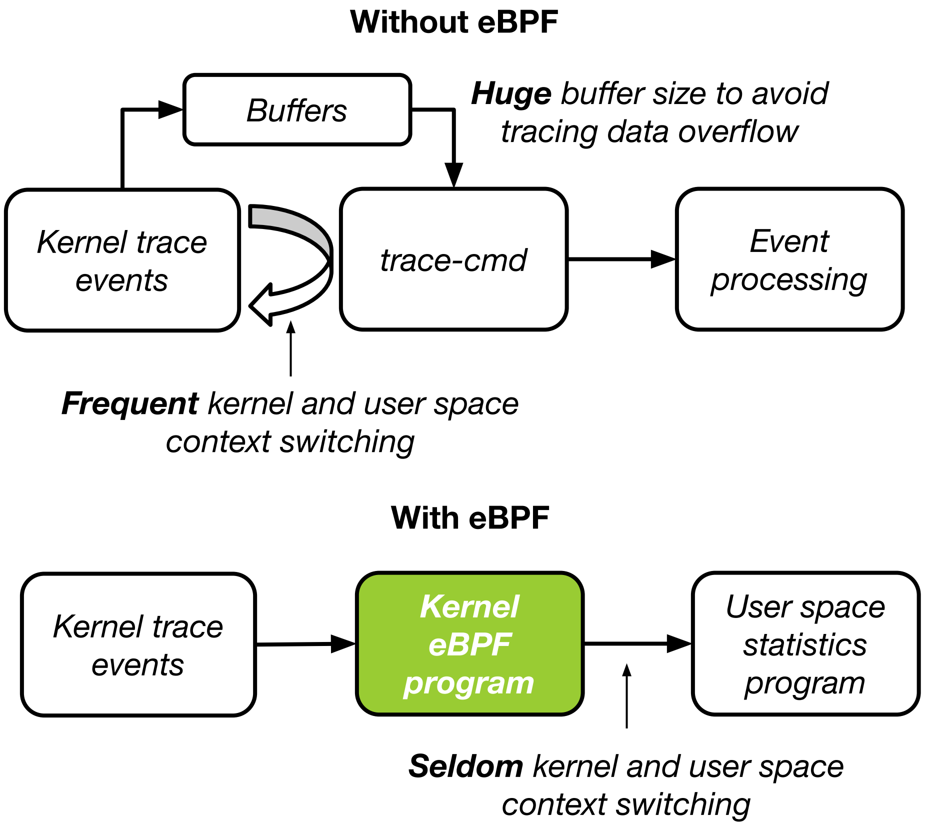
- eBPF program is written in C code and compiled to eBPF bytecode
  - LLVM/clang provides us a eBPF compiler (no support in gcc)
  - Direct code generation is also possible (or LLVM without clang)
- eBPF program is loaded inside eBPF virtual machine with sanity-checking
- eBPF program is "attached" to a designated code path in the kernel
  - eBPF in its traditional use case is attached to networking hooks allowing it to filter and classify network traffic using (almost) arbitrarily complex programs
  - Furthermore, we can attach eBPF programs to tracepoints, kprobes, and perf events for debugging the kernel and carrying out performance analysis.
- Kernel and user space typically use eBPF map; it is a generic data structure well suited to transfer data from kernel to userspace

Debugging with eBPF versus tracing

Tracing is very powerful but it can also be cumbersome for whole system analysis due to the volume of trace information generated.

Most developers end up writing programs to summarize the trace.

eBPF allows us to write program to summarize trace information without tracing.



Course outline

- Using eBPF for debugging
- Coding for eBPF in assembler
- eBPF tools
  - Kernel samples
  - Ply
  - BCC
  - SystemTap (stapbpf)
  - BPFtrace
  - Perf
- Debugging stories

libbpf: helper functions for eBPF

`libbpf` library makes easier to write eBPF programs, which includes helper functions for loading eBPF programs from kernel space to user space and creating and manipulating eBPF maps:

- User program reads the eBPF bytecode into a buffer and pass it to `bpf_load_program()` for program loading and verification.
- The eBPF program includes the libbpf header for the function definition for building, when run by the kernel, will call `bpf_map_lookup_elem()` to find an element in a map and store a new value in it.
- The user application calls `bpf_map_lookup_elem()` to read out the value stored by the eBPF program in the kernel.

```
int bpf_map_lookup_elem(int fd, const void *key,
                        void *value)
{
        union bpf_attr attr;

        bzero(&attr, sizeof(attr));
        attr.map_fd = fd;
        attr.key = ptr_to_u64(key);
        attr.value = ptr_to_u64(value);

        return sys_bpf(BPF_MAP_LOOKUP_ELEM, &attr,
                       sizeof(attr));
}
```

Coding for eBPF in assembler

```
int main(void)
{
        int map_fd, i, key;
        long long value = 0, cnt;

        map_fd = bpf_create_map(BPF_MAP_TYPE_ARRAY, sizeof(key), sizeof(value), 5000, 0);
        struct bpf_insn prog[] = {
                BPF_MOV64_REG(BPF_REG_6, BPF_REG_1),
                BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
                BPF_STX_MEM(BPF_W, BPF_REG_10, BPF_REG_0, -4), /* *(u32 *)(fp - 4) = r0 */
                BPF_MOV64_REG(BPF_REG_2, BPF_REG_10),
                BPF_ALU64_IMM(BPF_ADD, BPF_REG_2, -4), /* r2 = fp - 4 */
                BPF_LD_MAP_FD(BPF_REG_1, map_fd),
                BPF_RAW_INSN(BPF_JMP | BPF_CALL, 0, 0, 0, BPF_FUNC_map_lookup_elem),
                BPF_JMP_IMM(BPF_JEQ, BPF_REG_0, 0, 2),
                BPF_MOV64_IMM(BPF_REG_1, 1), /* r1 = 1 */
                BPF_RAW_INSN(BPF_STX | BPF_XADD | BPF_DW, BPF_REG_0, BPF_REG_1, 0, 0), /* xadd r0 += r1 */
                BPF_MOV64_IMM(BPF_REG_0, 0), /* r0 = 0 */
                BPF_EXIT_INSN(),
        };
        size_t insns_cnt = sizeof(prog) / sizeof(struct bpf_insn);
        pfd = bpf_load_program(BPF_PROG_TYPE_KPROBE, prog, insns_cnt, "GPL",
                               LINUX_VERSION_CODE, bpf_log_buf, BPF_LOG_BUF_SIZE);

        attach_kprobe();

        sleep(1);
        key = 0;
        assert(bpf_map_lookup_elem(map_fd, &key, &cnt) == 0);
        printf("sys_read counts %lld\n", cnt);
        return 0;
}
```

The example is ~50 lines of code for eBPF in assembler; it demonstrates the eBPF code have components: eBPF bytecode, syscalls, maps.

`attach_kprobe()` is used to enable kprobe event and attach the event with eBPF program.

```
void attach_kprobe(void)
{
    system("echo 'p:sys_read sys_read' >> \
               /sys/kernel/debug/tracing/kprobe_events")

    efd = open("/sys/kernel/debug/tracing/events/kprobes/sys_read/id",
               O_RDONLY, 0);
    read(efd, buf, sizeof(buf));
    close(efd);

    buf[err] = 0;
    id = atoi(buf);
    attr.config = id;

    efd = sys_perf_event_open(&attr, -1/*pid*/, 0/*cpu*/, -1, 0);
    ioctl(efd, PERF_EVENT_IOC_ENABLE, 0);
    ioctl(efd, PERF_EVENT_IOC_SET_BPF, pfd);
}
```

Course outline

- Using eBPF for debugging
- Coding for eBPF in assembler
- eBPF tools
  - Kernel samples
  - Ply
  - BCC
  - SystemTap (stapbpf)
  - BPFtrace
  - Perf
- Debugging stories

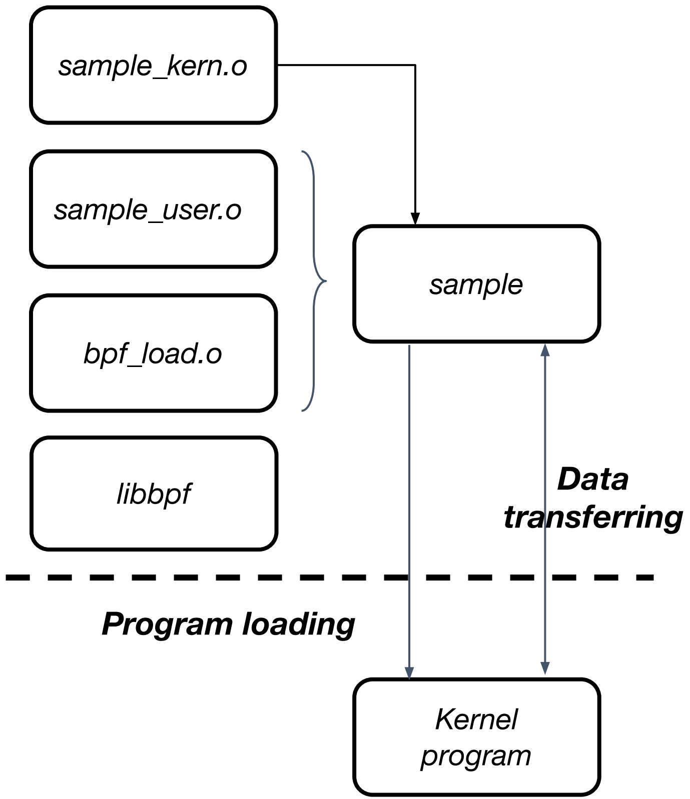
Kernel samples

It’s good to start from eBPF kernel samples; Linux kernel tree provides eBPF system call wrapper functions in lib `libbpf`; the samples use `bpf_load.c` to create map and load kernel program, **attach trace point**.

Kernel and user space programs use the naming convention `xxx_user.c` and `xxx_kern.c`, and the user space program to use file name `xxx_kern.o` to search kernel program.

The user space program is compiled by GCC for executable file and it reacts for ‘`CROSS_COMPILE=aarch64-linux-gnu-`’ for cross compiling. Kernel program is compiled by LLVM/Clang, by default it uses LLVM/Clang in distro and can specify path for new built LLVM/Clang. Build commands:

```
make headers_install   # creates "usr/include" directory in the build top directory
make samples/bpf/ LLC=xxx/llc CLANG=xxx/clang
```



Sample code: trace kmem_cache_alloc_node

tracex4_kern.c

```
struct bpf_map_def SEC("maps") my_map = {
        .type = BPF_MAP_TYPE_HASH,
        .key_size = sizeof(long),
        .value_size = sizeof(struct pair),
        .max_entries = 1000000,
};

SEC("kretprobe/kmem_cache_alloc_node")
int bpf_prog2(struct pt_regs *ctx)
{
        long ptr = PT_REGS_RC(ctx);
        long ip = 0;

        /* get ip address of kmem_cache_alloc_node() caller */
        BPF_KRETPROBE_READ_RET_IP(ip, ctx);

        struct pair v = {
                .val = bpf_ktime_get_ns(),
                .ip = ip,
        };

        bpf_map_update_elem(&my_map, &ptr, &v, BPF_ANY);
        return 0;
}
char _license[] SEC("license") = "GPL";
u32 _version SEC("version") = LINUX_VERSION_CODE;
```

Step 2: kernel program update map data

tracex4_user.c

```
static void print_old_objects(int fd)
{
        long long val = time_get_ns();
        __u64 key, next_key;
        struct pair v;

        /* Based on current ‘key’ value, we can get next key value
         * and iterate all bpf map elements. */
        key = -1;
        while (bpf_map_get_next_key(map_fd[0], &key, &next_key) == 0) {
                bpf_map_lookup_elem(map_fd[0], &next_key, &v);
                key = next_key;
                printf("obj 0x%llx is %2lldsec old was allocated at ip %llx\n",
                       next_key, (val - v.val) / 1000000000ll, v.ip);
        }
}

int main(int ac, char **argv)
{
        char filename[256];
        int i;

        snprintf(filename, sizeof(filename), "%s_kern.o", argv[0]);

        if (load_bpf_file(filename)) {
                printf("%s", bpf_log_buf);
                return 1;
        }

        for (i = 0; ; i++) {
                print_old_objects(map_fd[1]);
                sleep(1);
        }

        return 0;
}
```

Step 3: user space program reads map data

Step 1: load kernel program & enable kretprobe trace point

Ply: light dynamic tracer for eBPF

https://wkz.github.io/ply/

Ply uses an awk-like mini language describing how to attach eBPF programs to tracepoints and kprobes. It has a built-in compiler and can perform compilation and execution with a single command.

Ply can extract arbitrary data, i.e register values, function arguments, stack/heap data, stack traces.

Ply keeps dependencies to a minimum, leaving libc as the only runtime dependency. Thus, ply is well suited for **embedded targets**.

```
trace:raw_syscalls/sys_exit / (ret() < 0) /
{
        @[comm()].count()
}
```

```
^Cde-activating probes

@:
dbus-daemon                     2
ply                             3
irqbalance                      4
```

System call (sys_exit) failure statistics in ply

provider: selects which probe interface to use

probe definition: the point(s) of instrumentation

predicate: filter events to match criteria

```
trace:raw_syscalls/sys_exit / (ret() < 0) /
{
        @[comm()].count()
}
```

method: common way is to aggregate data using methods, have two functions: .count() and .quantize()

Key value

@: sign of map

```
^Cde-activating probes

@:
dbus-daemon                     2
ply                             3
irqbalance                      4
```

Tracing result: task name + counts

Build ply

https://github.com/iovisor/ply

If applicable, please check build: Fix kernel header installation on ARM64 is in your repository before building.

Method 1: Native compilation

```
./autogen.sh
./configure --with-kerneldir=/path/to/linux
make
make install
```

```
$ ldd src/ply
linux-vdso.so.1 (0x0000ffff9320d000)
libc.so.6 =>
/lib/aarch64-linux-gnu/libc.so.6
(0x0000ffff93028000)
/lib/ld-linux-aarch64.so.1
(0x0000ffff931e2000)
```

Method 2: Cross-Compilation

```
./autogen.sh
./configure --host=aarch64 --with-kerneldir=/path/to/linux
make CC=aarch64-linux-gnu-gcc
# copy src/ply to target board
```

BPF Compiler Collection (BCC)

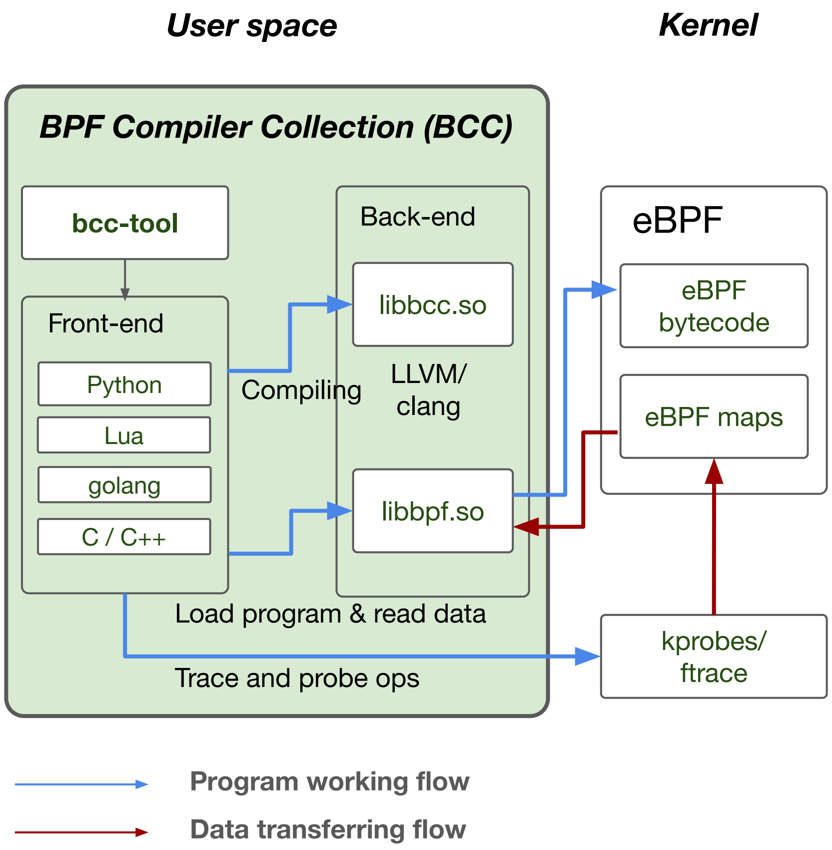
BPF compiler collection (BCC) project is a **toolchain** which reduces the difficulty for writing, compiling (invokes LLVM/Clang) and loading eBPF programs.

BCC **reports errors for mistake** for compiling, loading program, etc; this reduces difficulty for eBPF programming.

For writing short and expressive programs, **high-level languages** are available in BCC (python, Lua, go, etc).

BCC provides scripts that use **User Statically-Defined Tracing** (USDT) probes to place tracepoints in user-space code; these are probes that are inserted into user applications statically at compile-time.

BCC includes an impressive **collection** of examples and ready-to-use tracing tools.



BCC example code

https://github.com/iovisor/bcc/blob/master/examples/tracing/task_switch.py

```
b = BPF(text="""

struct key_t {
  u32 prev_pid, curr_pid;
};

BPF_HASH(stats, struct key_t, u64, 1024);
int count_sched(struct pt_regs *ctx, struct task_struct *prev) {
  struct key_t key = {};
  u64 zero = 0, *val;

  key.curr_pid = bpf_get_current_pid_tgid();
  key.prev_pid = prev->pid;

  val = stats.lookup_or_init(&key, &zero);
  (*val)++;
  return 0;
}
""")

b.attach_kprobe(event="finish_task_switch", fn_name="count_sched")

# generate many schedule events

for i in range(0, 100): sleep(0.01)


for k, v in b["stats"].items():
    print("task_switch[%5d->%5d]=%u" % (k.prev_pid, k.curr_pid, v.value))
```

Kernel program

Enable kprobe event

Read map data “stats”

https://github.com/iovisor/bcc/blob/master/INSTALL.md

Build BCC

BCC runs on the target but cannot be easily cross-compiled. These instructions show how to perform a native build (and work on an AArch64 platform)

Install build dependencies

```
sudo apt-get install debhelper cmake libelf-dev bison
flex libedit-dev python python-netaddr python-pyroute2
arping iperf netperf ethtool devscripts zlib1g-dev
libfl-dev
```

Build luajit lib

```
git clone http://luajit.org/git/luajit-2.0.git
cd luajit-2.0
git checkout -b v2.1 origin/v2.1
make
sudo make install
```

Build LLVM/Clang

```
cd where-llvm-live
svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
cd where-llvm-live
cd llvm/tools
svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
cd where-llvm-live
mkdir build (in-tree build is not supported)
cd build
cmake -G "Unix Makefiles" \
      -DCMAKE_INSTALL_PREFIX=$PWD/install ../llvm
make; make install
```

Build BCC

```
# Use self built LLVM/clang binaries
export PATH=where-llvm-live/build/install/bin:$PATH

git clone https://github.com/iovisor/bcc.git
mkdir bcc/build; cd bcc/build
cmake .. -DCMAKE_INSTALL_PREFIX=/usr
make
sudo make install
```

BCC and embedded systems

- BCC native build has many dependencies
  - Dependency with libs and binaries, e.g. cmake, luajit lib, etc
  - Most dependencies can be resolved for Debian/Ubuntu by using ‘apt-get’ command
  - BCC depends on LLVM/Clang to compile for eBPF bytecode, but LLVM/Clang itself also introduces many dependencies
- BCC and LLVM building requires powerful hardware
  - Have big pressure for both memory and filesystem space
  - Building is impossible or, with swap, extremely slow on systems without sufficient memory
  - Consumes lots of disk space. For AArch64: BCC needs **12GB**, additionally LLVM needs **42GB**
  - Even with strong hardware, the compilation process takes a long time
  - Save LLVM and BCC binaries on PC and use them by mounting NFS node :)
- Difficult to deploy BCC on Android system
  - No package manager means almost all library dependencies must be compiled from scratch
  - Android uses bionic C library, which makes it difficult to build libraries that use GNU extensions
  - androdeb: https://github.com/joelagnel/adeb

SystemTap - eBPF backend

- SystemTap introduced, stapbpf, an eBPF backend in Oct, 2017
    - Joins existing backends: kernel module and Dyninst

- SystemTap is both the tool and the scripting language
    - Language is inspired by awk, and predecessor tracers such as DTrace…
    - Uses the familar awk-like structure: probe.point { action(s) }
    - Extracts symbolic information based on DWARF parsing

```
# stap --runtime=bpf -v - <<EOF
> probe kernel.function("ksys_read") {
>   printf("ksys_read(%d): %d, %d\n",
>           pid(), $fd, $count);
>   exit();
> }
> EOF
Pass 1: parsed user script and 61 library
scripts using
410728virt/101984res/8796shr/93148data kb, in
260usr/20sys/272real ms.
Pass 2: analyzed script: 1 probe, 2 functions,
0 embeds, 0 globals using
468796virt/161004res/9684shr/151216data kb, in
820usr/10sys/843real ms.
Pass 4: compiled BPF into "stap_10960.bo" in
10usr/0sys/33real ms.
Pass 5: starting run.
ksys_read(18719): 0, 8191
Pass 5: run completed in 0usr/0sys/30real ms.
```

SystemTap - Revenge of the verifier

- eBPF verifier is more aggressive than the SystemTap language
  - Language permits looping but verifier prohibits loops (3.2 did not implement loop unrolling to compensate)
  - The 4096 opcode limit restriction also looms
  - `$$vars` and `$$locals` cause verification failure if used (likely depends on traced function)
  - *This runtime is in an early stage of development and it currently lacks support for a number of features available in the default runtime.* -- STAPBPF(8)

- SystemTap has a rich library of useful tested examples and war stories
  - Almost all are tested and developed using the kernel module backend
  - Thus it common to find canned examples that only work with the kernel module backend
  - This quickly grows frustrating… so one tends to end up using the default backend

BPFtrace - high level tracing language for eBPF

BPFtrace allows users to write trace code with high level tracing language for eBPF; BPFtrace language is inspired by awk and C, and predecessor tracers such as DTrace and SystemTap.

BPFtrace provides one-liners commands so it's convenient to trace system and provides built in variables and functions for tracing data analysis, this is similar with ply but BPFtrace is more versatile.

BPFtrace one-liner for syscall count by process

```
# bpftrace -e
'tracepoint:raw_syscalls:sys_enter { @[pid,
comm] = count(); }'
Attaching 1 probe...
^C

@[3180, dbus-daemon]: 9
@[1, systemd]: 13
@[3526, sshd]: 18
@[3766, bpftrace]: 28
@[3186, systemd-logind]: 53
@[3530, systemd]: 1004
```

BPFtrace with data structure support

BPFtrace depends on BCC and LLVM/Clang in its internal mechanism. It uses uses lex/yacc parser to convert programs to AST, then llvm IR actions to build eBPF bytecode, finally it relies on BCC to interact with kernel for eBPF program loading and probes attaching.

Furthermore, BPFtrace supports data structure with included header file, this functionality lets users to easily to read the data structure values even needs to traverse multiple pointers.

```
# cat > path.bt <<EOF
#include <linux/path.h>
#include <linux/dcache.h>

kprobe:vfs_open
{
    printf("open path: %s\n",
      str(((path *)arg0)->dentry->d_name.name));
}
EOF
# bpftrace path.bt
Attaching 1 probe...
open path: dev
open path: if_inet6
open path: retrans_time_ms
```

Build BPFtrace

Install build dependencies on olde Debian versions:

```
Manually build latest BCC so meet the requirement
```

Install build dependencies for Debian buster:

```
sudo apt-get install libbpfcc-dev
```

Build BPFtrace

```
git clone https://github.com/iovisor/bpftrace
mkdir bpftrace/build; cd bpftrace/build;
cmake -DCMAKE_BUILD_TYPE=Release ..
make -j8
make install
```

perf trace with eBPF event

perf is a suite of performance analysis tools which provided by Linux repository, it covers both hardware level features (e.g. PMU, timer, etc) together with software features (e.g. tracepoint, kprobe, etc) for performance profiling.

To decrease the bar for using eBPF in perf, perf has enhanced to integrate Clang for automatic eBPF program building and loading; the loaded eBPF program can invoke `perf_event_output()` to output eBPF events into user space. Different sub commands can be facilitated for tracing, or stores samples into perf.data for samples profiling.

```
perf trace
perf record -e bpf-kernel-prog.c
perf report
```

LLVM configurations in ~/.perfconfig

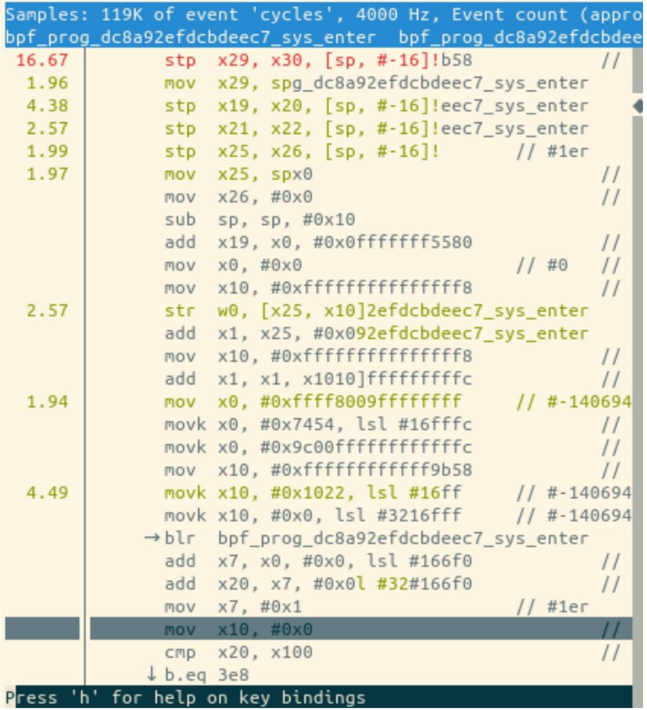
```
[llvm]
        clang-path = /usr/bin/clang-7
        kbuild-dir = /work/linux-cs-dev/
        clang-opt = "-DLINUX_VERSION_CODE=0x50200 -g"
        dump-obj = true

[trace]
        #add_events =
$Linux/perf/examples/bpf/augmented_raw_syscalls.c
        show_zeros = yes
        show_duration = no
        no_inherit = yes
        show_timestamp = no
        show_arg_names = no
        args_alignment = 40
        show_prefix = yes
```

perf with eBPF program profiling

Perf tool also can be used to profile eBPF program (same as other normal programs in the system) thus it can reflect the extra workload introduced by eBPF.

If connect with Perf's powerful functionality for samples profiling and program annotation, Perf tool also can be used to profile eBPF program performance.

So it's interesting that eBPF can be used for profiling and debugging, and on the other hand perf can profile eBPF programs.



Build perf

Install build dependencies on Debian

```
apt-get install flex bison libelf-dev
libaudit-dev libdw-dev libunwind*
python-dev binutils-dev libnuma-dev
libgtk2.0-dev libbfd-dev libelf1
libperl-dev libnuma-dev  libslang2
libslang2-dev libunwind8 libunwind8-dev
binutils-multiarch-dev elfutils
libiberty-dev libncurses5-dev
```

Install perf

```
cd $KERNEL_DIR
make VF=1 -C tools/perf/ install
```

Examples

- Using eBPF for debugging
- Coding for eBPF in assembler
- eBPF tools
  - Kernel samples
  - Ply
  - BCC
  - SystemTap (stapbpf)
  - BPFtrace
  - Perf
- Debugging stories

The story - Hunting leaks

I know I'm leaking memory (or some other precious resource) from a particular pool whenever I run a particular workload. Unfortunately my system is almost ready to ship and we've started disabling all the resource tracking. Is there anything I can do to get a clue about what is going on?

```
# cat track.ply
kprobe:kmem_cache_alloc_node {
	# Can't read stack from a retprobe :-(
	@[0] = stack();
}
kretprobe:kmem_cache_alloc_node {
	@[retval()] = @[0];
	@[0] = nil;
}
kprobe:kmem_cache_free {
	@[arg(1)] = nil;
}
# ply -t 1 track.ply
3 probes active
de-activating probes

@:
	<leaks show up here>
```

The story - Debug kernel functions at the runtime

Inspired by: [BPF: Tracing and More (Brendan Gregg)]

When I debug CPU frequency change flow in kernel, kernel have several different components to work together for frequency changing, including clock driver, mailbox driver, etc.

I want to confirm if the functions have been properly called and furthermore to check function arguments have expected values.

How can I dynamically debug kernel functions at the runtime with high efficiency and safe method?

- SystemTap and Kprobes can be used to debug kernel function, but eBPF is safer to deploy because the verifier will ensure kernel integrity.
- For kernel functions tracing, eBPF can avoid to change kernel code and save time for compilation.
- If it's safe enough, we even can use it in production for customer support.
- In this example, we use tools from the bcc distribution

Debug kernel functions

```
static int hi3660_stub_clk_set_rate(struct clk_hw *hw, unsigned long rate,
                                     unsigned long parent_rate)
{
        struct hi3660_stub_clk *stub_clk = to_stub_clk(hw);

        stub_clk->msg[0] = stub_clk->cmd;
        stub_clk->msg[1] = rate / MHZ;

        mbox_send_message(stub_clk_chan.mbox, stub_clk->msg);
        mbox_client_txdone(stub_clk_chan.mbox, 0);

        stub_clk->rate = rate;
        return 0;
}
```

```
$ ./tools/trace.py 'hi3660_stub_clk_set_rate "rate: %d" arg2'
PID     TID     COMM            FUNC             -
2002    2002    kworker/3:2     hi3660_stub_clk_set_rate rate: 1421000000
2469    2469    kworker/3:1     hi3660_stub_clk_set_rate rate: 1421000000
2469    2469    kworker/3:1     hi3660_stub_clk_set_rate rate: 1421000000
84      84      kworker/0:1     hi3660_stub_clk_set_rate rate: 903000000
2469    2469    kworker/3:1     hi3660_stub_clk_set_rate rate: 903000000
84      84      kworker/0:1     hi3660_stub_clk_set_rate rate: 903000000
84      84      kworker/0:1     hi3660_stub_clk_set_rate rate: 903000000
2469    2469    kworker/3:1     hi3660_stub_clk_set_rate rate: 903000000
```

BCC `tools/trace.py` can be used to debug kernel function; this tool can trace function with infos: kernel or user space stack, timestamp, CPU ID, PID/TID.

We can use tool `trace.py` to confirm function `hi3660_stub_clk_set_rate()` has been invoked and print out the target frequency.

Debug kernel functions - cont.

```
static int hi3660_mbox_send_data(struct mbox_chan *chan, void *msg)
{
        [...]

        /* Fill message data */
        for (i = 0; i < MBOX_MSG_LEN; i++)
                writel_relaxed(buf[i], base + MBOX_DATA_REG + i * 4);

        /* Trigger data transferring */
        writel(BIT(mchan->ack_irq), base + MBOX_SEND_REG);
        return 0;
}
```

```
$ ./tools/trace.py 'hi3660_mbox_send_data(struct mbox_chan *chan, void *msg)
"msg_id: 0x%x rate: %d", *((unsigned int *)msg), *((unsigned int *)msg + 1)'

PID     TID     COMM            FUNC             -
84      84      kworker/0:1     hi3660_mbox_send_data msg_id: 0x2030a rate: 903
2413    2413    kworker/1:0     hi3660_mbox_send_data msg_id: 0x2030a rate: 903
2413    2413    kworker/1:0     hi3660_mbox_send_data msg_id: 0x2030a rate: 903
```

We can continue to check program flow from high level function to low level function for arguments, and BCC supports C style sentence to print out more complex data structure.

These data “watch points” can easily help us to locate the issue happens in which component.

For left example, we can observe the `msg_id` value to check if pass correct message ID to MCU firmware.

Statistics based on function arguments

```
static int hi3660_stub_clk_set_rate(struct clk_hw *hw, unsigned long rate,
                                    unsigned long parent_rate)
{
        struct hi3660_stub_clk *stub_clk = to_stub_clk(hw);

        stub_clk->msg[0] = stub_clk->cmd;
        stub_clk->msg[1] = rate / MHZ;

        mbox_send_message(stub_clk_chan.mbox, stub_clk->msg);
        mbox_client_txdone(stub_clk_chan.mbox, 0);

        stub_clk->rate = rate;
        return 0;
}
```

After the kernel functionality has been validated, we can continue to do simple profiling based on Kernel function argument statistics.

Using the `argdist.py` invocation below, we can observe the the CPI frequency mostly changes to 533MHz and 1844MHz.

```
$ tools/argdist.py -I 'linux-mainline/include/linux/clk-provider.h'
  -c -C 'p::hi3660_stub_clk_set_rate(struct clk_hw *hw, unsigned long rate,
unsigned long parent_rate):u64:rate'

      COUNT      EVENT
      1          rate = 903000000
      1          rate = 2362000000
      1          rate = 999000000
      27         rate = 1844000000
      31         rate = 533000000
```

Summary (and thank you)

Everything is awesome…

… and many, many thanks to all the people who have worked to make it so!

Hand-rolled

Asm	Hack value?
Pure C	No “magic”, great examples in kernel

Awk-like

Ply	Easy to deploy esp. on embedded system
SystemTap	DWARF parsing (and wait a bit?)
BPFtrace	`#include <linux/dentry.h>`
Perf	C code and facilitate perf profiling functionality

BCC	Great tool for tool makers (and running tools from tool makers)

support@linaro.org