LTTng's Trace Filtering and beyond
(with some eBPF goodness, of course!)

Suchakrapani Datt Sharma

Aug 20, 2015

École Polytechnique de Montréal
Laboratoire DORSAL
Suchakra

• PhD student, Computer Engineering  
  (Prof Michel Dagenais)  
  DORSAL Lab, École Polytechnique de Montréal - UdeM

• Works on debugging, tracing and trace analysis (LTTng),  
  bytecode interpreters, JIT compilation, dynamic  
  instrumentation

• Loves poutine
Agenda

LTTng's Trace Filter

- Filtering primer
- LTTng's trace filters

eBPF

- Mechanism, current status
  - BCC
- A small eBPF trial with LTTng
- Filtering performance with experimental userspace eBPF

Beyond

- KeBPF/UeBPF?
Filters
filter
\[(\text{type} = \text{ARP}) \text{ OR } (\text{type} = \text{IP})) \text{ AND } (\text{origin} = \text{SRC}) \text{ AND } (\text{size} < 1024)\]
\[
\begin{align*}
\underbrace{(type = ARP)}_{P1} \quad \text{OR} \quad \underbrace{(type = IP)}_{P2} \quad \text{AND} \quad \underbrace{(origin = SRC)}_{P3} \quad \text{AND} \quad \underbrace{(size < 1024)}_{P4}
\end{align*}
\]
Evaluating Filters
\[(\text{type} = \text{ARP}) \lor (\text{type} = \text{IP})\] \land (\text{origin} = \text{SRC}) \land (\text{size} < 1024)
\[(\text{type} = \text{ARP}) \text{ OR } (\text{type} = \text{IP}) \text{ AND } (\text{origin} = \text{SRC}) \text{ AND } (\text{size} < 1024)\]

Foo Evaluator

Take whole string expression and start parsing and evaluating by hand

TRUE / FALSE
(((\text{type} = \text{ARP}) \text{ OR} (\text{type} = \text{IP})) \text{ AND} (\text{origin} = \text{SRC}) \text{ AND} (\text{size} < 1024))

\[ P_1 \quad P_2 \quad P_3 \quad P_4 \]

Foo Evaluator
Take whole string expression and start parsing and evaluating by hand

TRUE / FALSE

42 billion runs
\[( (\text{type} = \text{ARP}) \text{ OR } (\text{type} = \text{IP})) \text{ AND } (\text{origin} = \text{SRC}) \text{ AND } (\text{size} < 1024) \]\n
**Bar Generator**

Parser → AST → IR → Bytecode

**Bar Interpreter**

 Bytecode

**TRUE / FALSE**
TRUE / FALSE

Bar Generator
Parser → AST → IR → Bytecode

Bar Interpreter
Bytecode

BAR GENERATOR
PARSER → AST → IR → BYTECODE

TRUE / FALSE
((\text{type} = \text{ARP}) \text{ OR } (\text{type} = \text{IP})) \text{ AND } (\text{origin} = \text{SRC}) \text{ AND } (\text{size} < 1024)
((type = ARP) OR (type = IP)) AND (origin = SRC) AND (size < 1024)

- **Bar Generator**
  - Parser → AST → IR → Bytecode

- **JIT Compiler**
  - Bytecode → Native Code

- **Native Code**
  - (x86/ARM)

**TRUE / FALSE**
Bar Generator
Parser → AST → IR → Bytecode

JIT Compiler
Bytecode → Native Code

Native Code
(x86/ARM)

TRUE / FALSE
Why do we need these blazingly FAST filters?
Network

- Sustain network throughput
- Effect is visible on embedded devices which work uninterrupted

Tracing

- Filtering huge event flood at runtime reliably
- High frequency events long-running trace events in production systems with limited resources to defer analysis
$\left( \left( \text{type} = \text{ARP} \right) \text{ OR } \left( \text{type} = \text{IP} \right) \right) \text{ AND } \left( \text{origin} = \text{SRC} \right) \text{ AND } \left( \text{size} < 1024 \right)$
LTTng's Trace Filtering
LTTng-UST

Instrumented Userspace Application

UST listener thread

LTTng Session Daemon

LTTng Consumer Daemon

SHM

CTF Trace
LTTng-UST

Instrumented Userspace Application

UST listener thread

LTThng Session Daemon

LTThng Consumer Daemon

SHM

Register Event

Setup Event Consumption

CTF Trace

Ring buffer
LTTng-UST

Instrumented Userspace Application

UST listener thread

LTTng Session
Daemon
LTTng-UST Filtering

Instrumented Userspace Application

New Event

LTTng Session Daemon
**LTTng-UST Filtering**

**Instrumented Userspace Application**

1. **New Event**
2. **Check for Filter**
3. **Parse → AST → IR**
4. **Generate Bytecode**
5. **LTTng Session Daemon**

- User sets filter
- Basic IR Validation
LTTng-UST Filtering

Instrumented Userspace Application

- New Event
- Check for Filter
- Parse → AST → IR
- Generate Bytecode
- LTTng Session Daemon
- Validate → Link → Interpret
- Filtered Events
- Send Bytecode

User sets filter

Interpret for every event

Basic IR Validation
LTTng's Trace Filtering

A filtered session

$ lttng create mysession
$ lttng enable-event --filter '(foo == 42) && (bar == "baz")' -a -u

Filter '(foo == 42) && (bar == "baz")' successfully set

$ lttng start
<do some science>

$ lttng stop
$ lttng view
A filtered session

$ lttng create mysession
$ lttng enable-event --filter '(foo == 42) && (bar == "baz")' -a -u

Filter '(foo == 42) && (bar == "baz")' successfully set

$ lttng start
<do some science>
$ lttng stop
$ lttng view
Generating Bytecode
generate_filter()

- Flex-Bison generated lexer-parser
- Custom tokens and grammar

```c
ctx = filter_parser_ctx_alloc(fmem);
```

- Allocate/initilize parser, AST, create root node

```c
filter_parser_ctx_append_ast(ctx);
filter_visitor_set_parent(ctx);
```

- Run `yyparse()`, `yylex()`
- Generate syntax tree
Filter Bytecode Generation

Syntax Tree

\[
\begin{array}{c}
\text{id}(\text{foo}) \quad \text{c}(42) \\
\text{op}(==) \quad \text{op}(==)
\end{array}
\]

\[
\begin{array}{c}
\text{id}(\text{bar}) \quad \text{str}(\text{"bar"})
\end{array}
\]
Filter Bytecode Generation

```c
filter_visitor_ir_generate(ctx);
```

- Hand written IR generator
- Go through each node recursively, classify them
- **No binary arithmetic** supported for now. Only logic and comparisons

```c
filter_visitor_ir_check_binary_op_nesting(ctx);
filter_visitor_ir_validate_string(ctx);
```

- Basic IR Validation
  - Except logical operators, operator nesting not allowed
  - Validate string as literal part - No wildcard in between strings, no unsupported characters
Filter Bytecode Generation

```c
filter_visitor_bytecode_generate(ctx);
```

- Traverse tree post-order
- Based on node type, start emitting instructions
- Save the bytecode in ctx
- Add symbol table data to bytecode.
- We are done, lets send it to lttng-sessiond!
Interpreting Bytecode
Filter Bytecode Interpretation

`lttng_filter_event_link_bytecode()`

- Link bytecode to the event and create bytecode runtime
  - Copy original bytecode to runtime
  - Apply field and context relocations

`lttng_filter_validate_bytecode(runtime);`

- Check unsupported bytecodes (eg. arithmetic)
- Check range overflow for different insn classes
- Validate current context and merge points for all insn

`lttng_filter_specialize_bytecode(runtime);`

- We know event field types now
- Lets specialize operations based on that
Filter Bytecode Interpretation

lttng_filter_interpret_bytecode()

- Hybrid virtual machine
- 2 registers (ax & bx) aliased to top of stack
- Functions like register machine - flexible like stack
- Threaded instruction dispatch/normal dispatch (fallback)

```c
OP(FILTER_OP_NE_S64):
{
    int res;
    res = (estack_bx_v != estack_ax_v);
    estack_pop(stack, top, ax, bx);
    estack_ax_v = res;
    next_pc += sizeof(struct binary_op);
    PO;
}
```
eBPF
Filters & More
eBPF

Berkeley Packet Filter (BPF)

- Filter expressions → Bytecode → Interpret
- Fast, small, in-kernel packet & syscall filtering
- Register based, switch-dispatch interpreter

Current Status of BPF

- Extensions for trace filtering (Kprobes!! Kprobes!!)
- More than just filtering. JITed programs - FAST!
- Evolved to extended BPF (eBPF)
  - BPF maps, bpf syscall - aggregation and userspace access
  - More registers (64 bit), back jumps, tail-calls, safety
Example eBPF Session

Kernel

Userspace

foo_kern.c
Example eBPF Session

foo_kern.c

BPF LLVM backend

foo_kern.bpf

Kernel | Userspace
Example eBPF Session

Kernel | Userspace

foo_kern.c

foo_kern.bpf

foo_user.c

foo_kern.bpf

BPF LLVM backend

Load
Example eBPF Session

- **foo_kern.c**
- **foo_kern.bpf**
- **foo_user.c**
  - Foo_kern.bpf
  - Bytecode

Diagram:
- Kernel
- Userspace
- BPF LLVM backend
- Load
Example eBPF Session

- **eBPF**
  - BPF Bytecode
  - BPF Maps

- **foo_kern.c**
  - BPF LLVM backend

- **foo_kern.bpf**
  - Load

- **foo_user.c**
  - foo_kern.bpf
  - Bytecode

- **bpf() Syscalls**
Example eBPF Session

```c
void blk_start_request (struct request *req) {
    blk_dequeue_request(req);
    ...
}
```

- **eBPF**
  - BPF Bytecode
  - BPF Maps

- **foo_kern.c**
  - BPF LLVM backend
- **foo_kern.bpf**
  - Load
- **foo_user.c**
  - Bytecode

- **block/blk-core.c**

- **Kprobe**

- **Kernel**
- **Userspace**

**bpf() Syscalls**
Example eBPF Session

```c
void blk_start_request (struct request *req) {
    blk_dequeue_request(req);
    
}
```

- **Kprobe**

- **BPF Bytecode**

- **BPF Maps**

- **foo_kern.c**
  - BPF LLVM backend
  - foo_kern.bpf
  - foo_kern.bpf

- **foo_user.c**
  - Bytecode
  - foo_kern.bpf
  - foo_kern.bpf
  - Read Maps

- **Kernel**

- **Userspace**

- **bpf() Syscalls**
Sample eBPF Filter

eBPF Filter on LTTng Kernel Event

```c
if ((dev->name[0] == "l") && (dev->name[1] == "o"))
{
    trace_netif_receive_skb_filter(skb);
}
```

eBPF Bytecode :

```c
static struct bpf_insn insn_prog[] = {
    BPF_LDX_MEM(BPF DW, BPF_REG 2, BPF_REG 1, 0),
    BPF_LDX_MEM(BPF DW, BPF_REG 3, BPF_REG 2, 0), /* ctx->arg1 */
    BPF_LDX_MEM(BPF DW, BPF_REG 4, BPF_REG 1, 8), /* ctx->arg2 */
    BPF_JMP_REG(BPF JEQ, BPF REG 3, BPF REG 4, 3), /* compare arg1 & arg2 */
    BPF_LD IMM64(BPF REG 0, 0), /* FALSE */
    BPF_EXIT_INSN(),
    BPF_LD IMM64(BPF REG 0, 1), /* TRUE */
    BPF_EXIT_INSN(),
};
```
Sample eBPF Filter

eBPF JITed:

0: push %rbp
1: mov %rsp,%rbp
4: sub $0x228,%rsp
b: mov %rbx,-0x228(%rbp)
12: mov %r13,-0x220(%rbp)
18: mov %r14,-0x218(%rbp)
20: mov %r15,-0x210(%rbp)
27: xor %eax,%eax
29: xor %r13,%r13
2c: mov 0x0(%rdi),%rsi
mov 0x0(%rsi),%rdx
34: mov 0x8(%rdi),%rcx
38: cmp %rcx,%rdx

Make some space on stack

3b: je 0x0000000000000049
3d: movabs $0x0,%rax ;FALSE
47: jmp 0x0000000000000053
49: movabs $0x1,%rax ;TRUE
53: mov -0x228(%rbp),%rbx
5a: mov -0x220(%rbp),%r13
61: mov -0x218(%rbp),%r14
68: mov -0x210(%rbp),%r15
6f: leaveq
70: retq

Load ctx args to R3 and R4

Jump to TRUE

Clear A and X

Compare R3, R4

Save callee saved regs

Restore regs

One-to-one direct method JIT. eBPF is close to modern architectures
Example eBPF Session

```c
void blk_start_request (struct request *req)
{
  blk_dequeue_request(req);
  :
}
```

**Kprobe**

void blk_start_request (struct request *req)
{
  blk_dequeue_request(req);
  :
}

**block/blk-core.c**

**Kernel** outflows to **Userspace**

**BPF LLVM** backend

**Load**

**BPF Bytecode**

**BPF Maps**

**foo_kern.c**

**foo_kern.bpf**

**foo_user.c**

**Read Maps**

**BPF Maps**

**foo_kern.bpf**

**BPF Bytecode**

**foo_kern.c**
Yes, 'bcc' exists!

https://github.com/iovisor/bcc
Example `bcc` Session

**eBPF**
- BPF Bytecode
- BPF Maps

**foo_kern.c**
- `load_func()`
- `get_table()`
- `attach_kprobe()`

**foo_user.py**
- `load_func()`
- `get_table()`
- `attach_kprobe()`

**Kernel**
- `blk_start_request`
  
  ```c
  void blk_start_request (struct request *req)
  {
    blk_dequeue_request(req);
  }
  ```

**Userspace**

-Kprobe

**block/blk-core.c**

-`bpf()` Syscalls

POLYTECHNIQUE MONTREAL – Suchakrapani Datt Sharma
#include <uapi/linux/ptrace.h>
#include <linux/sched.h>

struct key_t {
    u32 prev_pid;
    u32 curr_pid;
};

BPF_TABLE("hash", struct key_t, u64, stats, 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;
}

from bpf import BPF
from time import sleep

b = BPF(src_file="task_switch.c")
fn = b.load_func("count_sched", BPF.KPROBE)
stats = b.get_table("stats")
BPF.attach_kprobe(fn, "finish_task_switch")

# generate many schedule events
for i in range(0, 100): sleep(0.01)

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

- Primarily for filters & script driven tracing - FAST, very FAST!
- Add sophisticated features to tracing, at low cost
  - Fast stateful kernel event filtering/data aggregation
    - Record system wide sched_wakeup only when target process is blocked to reduce overhead
  - Utilize *side-effects* for assisted-tracing
- A more uniform way of filtering events across userspace and kernel
Experiments

Userspace eBPF (UeBPF)

- Experimental *libebpf* to provide filtering in userspace tracing
- Includes side-effects through communication with modified KeBPF
- Easy switch between JIT/interpret for performance analysis
- Includes LLVM BPF backend.
- Load bytecode from eBPF binaries

Performance Analysis

- Apply LTTng, eBPF, eBPF+JIT, hardcoded filters
- Measure $t_{\text{execution}} + t_{\text{tracepoint}}$
Experiments

Performance Analysis

- Pure filter evaluation.
  - TRUE/FALSE biased AND chain with varying predicates
- Measure $t_e + t_t$ with varying DoE (Biased TRUE)
Experiments

Performance Analysis

• Steady gain in 3x range for JIT vs Interpreted with increasing events (3.1x to 3.3x)
Experiments

Performance Analysis

- eBPF JITed filter is 3.1x faster than LTTng's interpreted bytecode and eBPF's interpreted filter is 1.8x faster than LTTng's interpreted version.
Inferences from Experiments

- JIT is so fast it makes everything slow
  - Next thing after “throw some cores” and “add some cache”
- Small specialized interpreters can be quite fast too (LTTng)
- For the tracing use-case, LTTng's filter works remarkably well
- Integrate with LTTng and real life benchmarks on specialized hardware
Beyond

KeBPF ↔ UeBPF Extensions

• Syscall latency tracking use-case.
• Latency threshold is defined statically and manually
  • In real life, it may need to be set dynamically - different machines can have different normal levels for syscalls
  • We may need to adaptively set thresholds per syscall based on user's criteria as well as tracking the normal behaviour.
• We can use eBPF side-effects to provide dynamic and adaptive thresholds
Beyond

KeBPF ↔ UeBPF Extensions

● Side-effects?
  ● eBPF can do more complex things like perform internal actions in addition to decisions
  ● Use it to make decisions in kernel BPF based on userspace BPF inputs
  ● Access shared data from KeBPF/UeBPF
Beyond

KeBPF ↔ UeBPF Syscall Latency Tracking

KeBPF FILTER

threshold
{predicate}

Register 42
latency()
tracepoint()

Latency Tracker Module

Kernel

UeBPF FILTER

reg_ioctl()

PID 42

bpf_set_threshold()

Userspace

Kernel

Userspace
Beyond

KeBPF ↔ UeBPF Syscall Latency Tracking

Kernel

Latency Tracker Module
- KeBPF FILTER
  - threshold
  - proc_state
  - {predicate}
- latency()
- tracepoint()

Userspace

PID 42
- UeBPF FILTER
  - reg_pid()
  - bpf_set_threshold()

Shared Mem
- proc_state
- threshold

POLYTECHNIQUE MONTREAL – Suchakrapani Datt Sharma
References

- Graphics and text on slide 24-26 have been adapted from David Goulet's talk at FOSDEM '14.
- Example for 'bcc' on slide 54: https://github.com/iovisor/bcc
- Experimental libebpf: https://github.com/tuxology/libebpf
- BPF Internals
  - Part - I: http://ur1.ca/nheth
  - Part - II: http://ur1.ca/nheto

All the images in this presentation drawn by the author are released under Creative Commons. All other graphics have been taken from OpenClipArt and are under public domain.
Acknowledgments

Thanks to EfficiOS, Ericsson Montréal and DORSAL Lab, Polytechnique Montreal for the awesome work on LTTng/UST, TraceCompass and LTTngTop. Thanks to DiaMon Workgroup for the opportunity to present.
Questions?

suchakrapani.sharma@polymtl.ca

suchakra on #lttng (irc.oftc.net)
@tuxology
http://suchakra.in