LTTng's Trace Filtering
and beyond

(with some eBPF goodness, of course!)

Suchakrapani Datt Sharma

Aug 20, 2015

École Polytechnique de Montréal

Laboratoire DORSAL

whoami

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

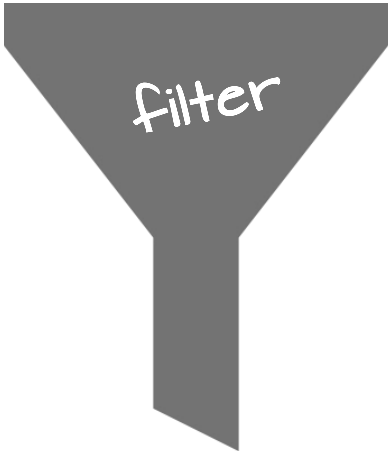
eBPF

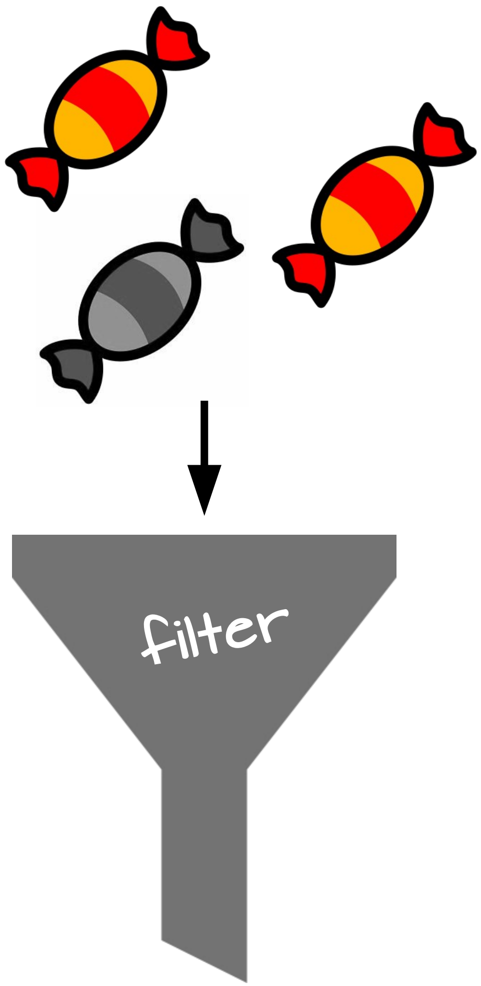
- Mechanism, current status
  - BCC
- A small eBPF trial with LTTng
- Filtering performance with experimental userspace eBPF

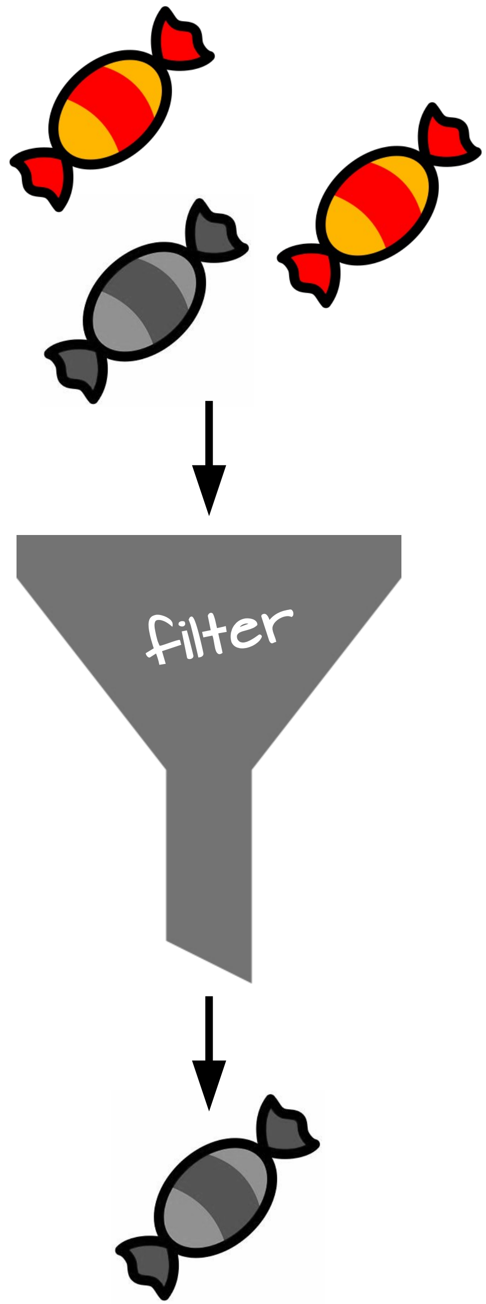
Beyond

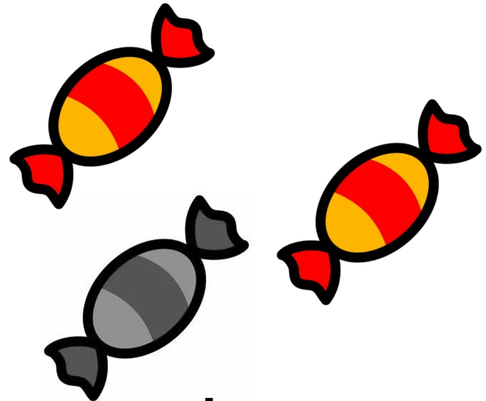
- KeBPF/UeBPF?

Filters

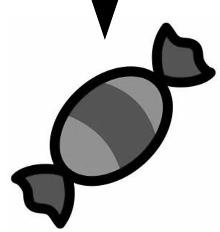

filter

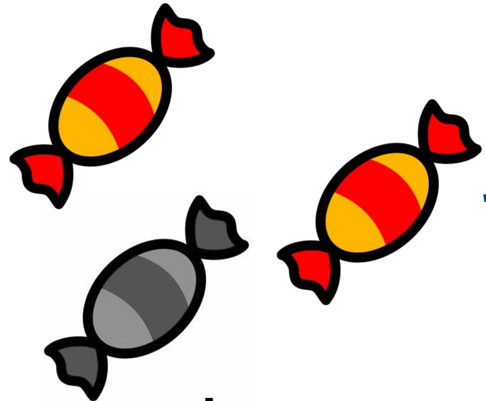

filter


filter



$$(\underbrace{(type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\ \mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

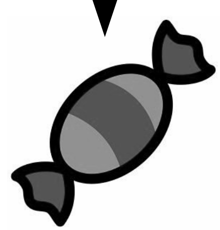




Packets

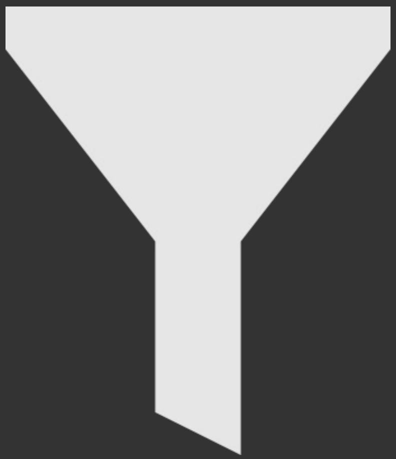
$$(\underbrace{(type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\,\mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

Predicates



Evaluating

Filters



$$\underbrace{((type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\ \mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

$$((\underbrace{type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\ \mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

Foo Evaluator

Take whole string expression and start parsing and evaluating by hand

TRUE / FALSE

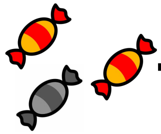
$$\underbrace{((type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\,\mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

Foo Evaluator

Take whole string expression and start parsing and evaluating by hand

TRUE / FALSE

42 billion runs



$$\underbrace{((type = ARP)}_{P1}\ \textbf{OR}\ \underbrace{(type = IP)}_{P2})\ \textbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \textbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

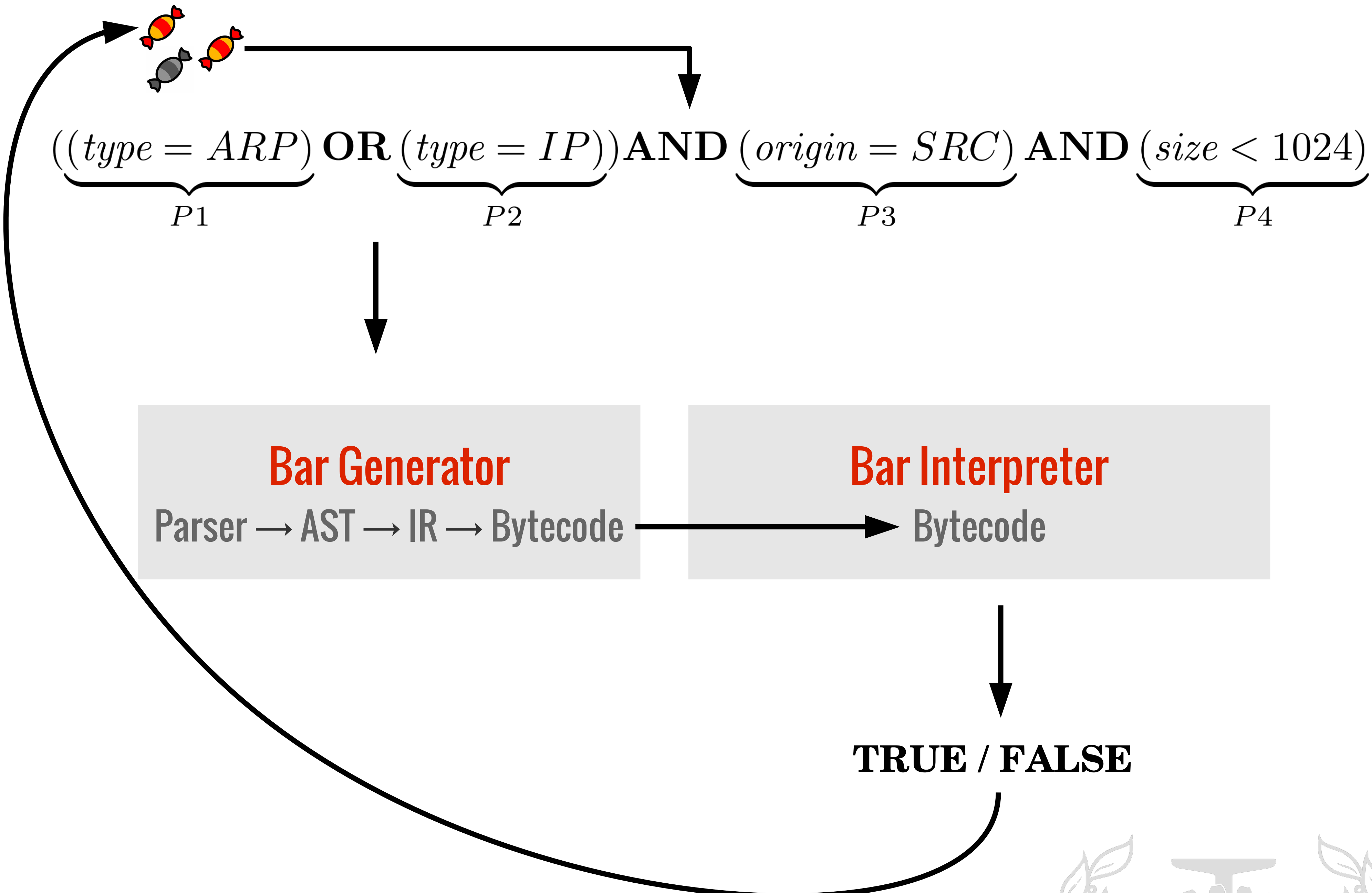
Bar Generator

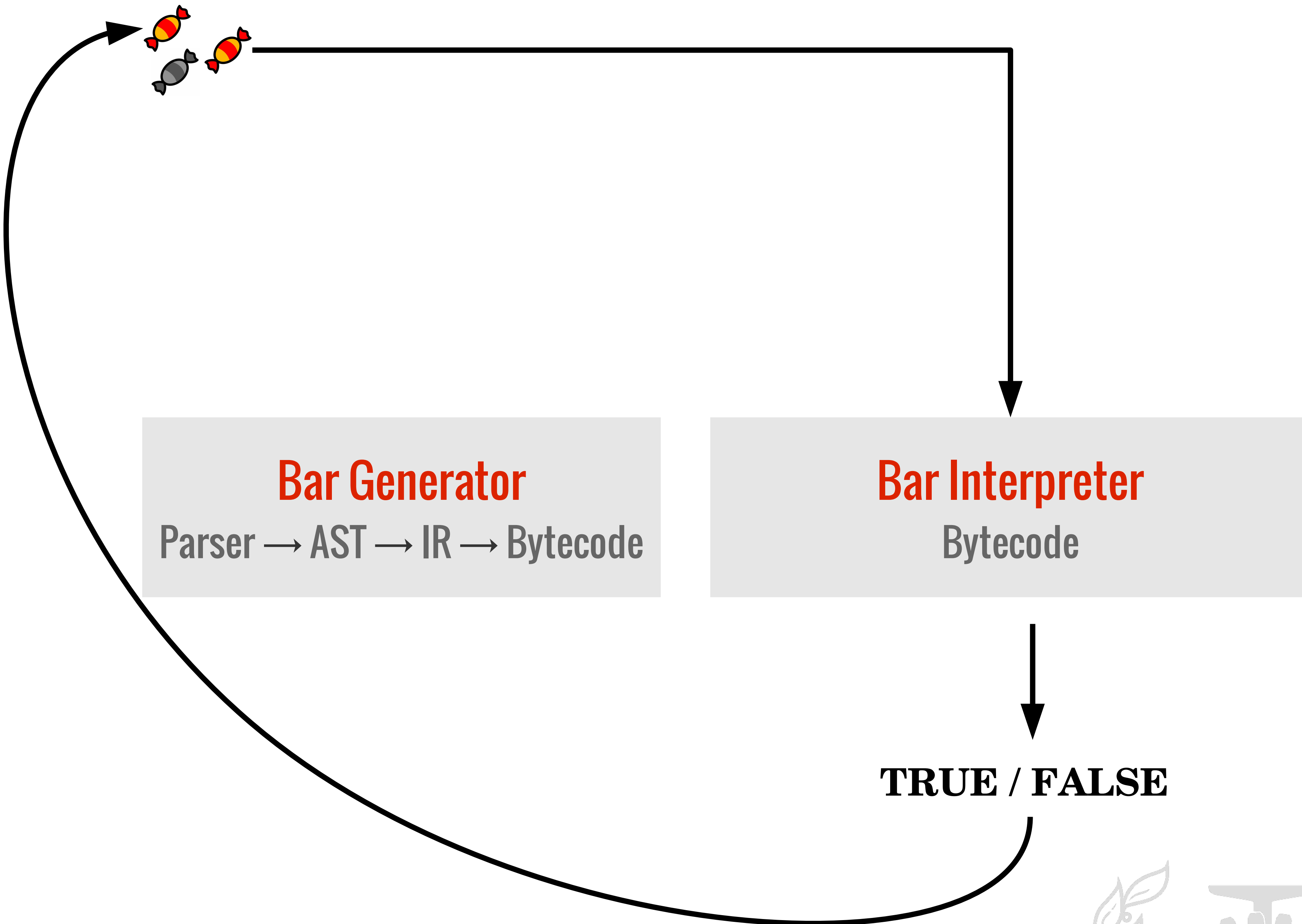
Parser → AST → IR → Bytecode

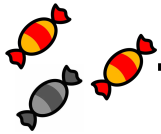
Bar Interpreter

Bytecode

TRUE / FALSE


$\underbrace{((type = ARP)}_{P1}$ **OR** $\underbrace{(type = IP)}_{P2})$ **AND** $\underbrace{(origin = SRC)}_{P3}$ **AND** $\underbrace{(size < 1024)}_{P4}$
Bar Generator
Parser → AST → IR → Bytecode
Bar Interpreter
Bytecode
TRUE / FALSE


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



$$\underbrace{((type = ARP)}_{P1}\ \textbf{OR}\ \underbrace{(type = IP)}_{P2})\,\textbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \textbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

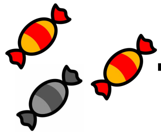
Bar Generator

Parser → AST → IR → Bytecode

Bar Interpreter

Bytecode

TRUE / FALSE



$$\underbrace{((type = ARP)}_{P1}\ \textbf{OR}\ \underbrace{(type = IP)}_{P2})\ \textbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \textbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

Bar Generator

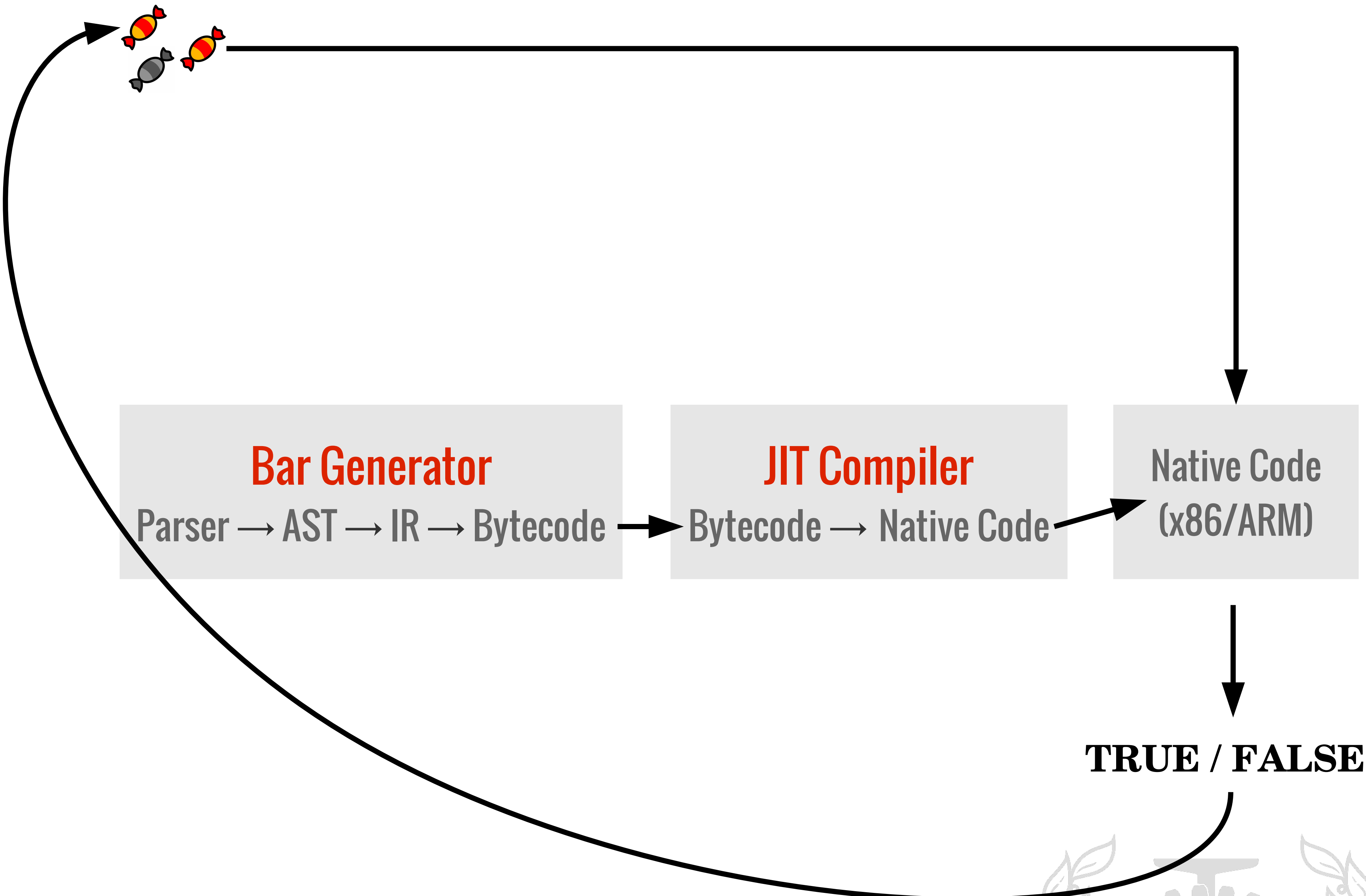
Parser → AST → IR → Bytecode

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

$$\underbrace{((type = ARP)}_{P1}\ \mathbf{OR}\ \underbrace{(type = IP)}_{P2})\mathbf{AND}\ \underbrace{(origin = SRC)}_{P3}\ \mathbf{AND}\ \underbrace{(size < 1024)}_{P4}$$

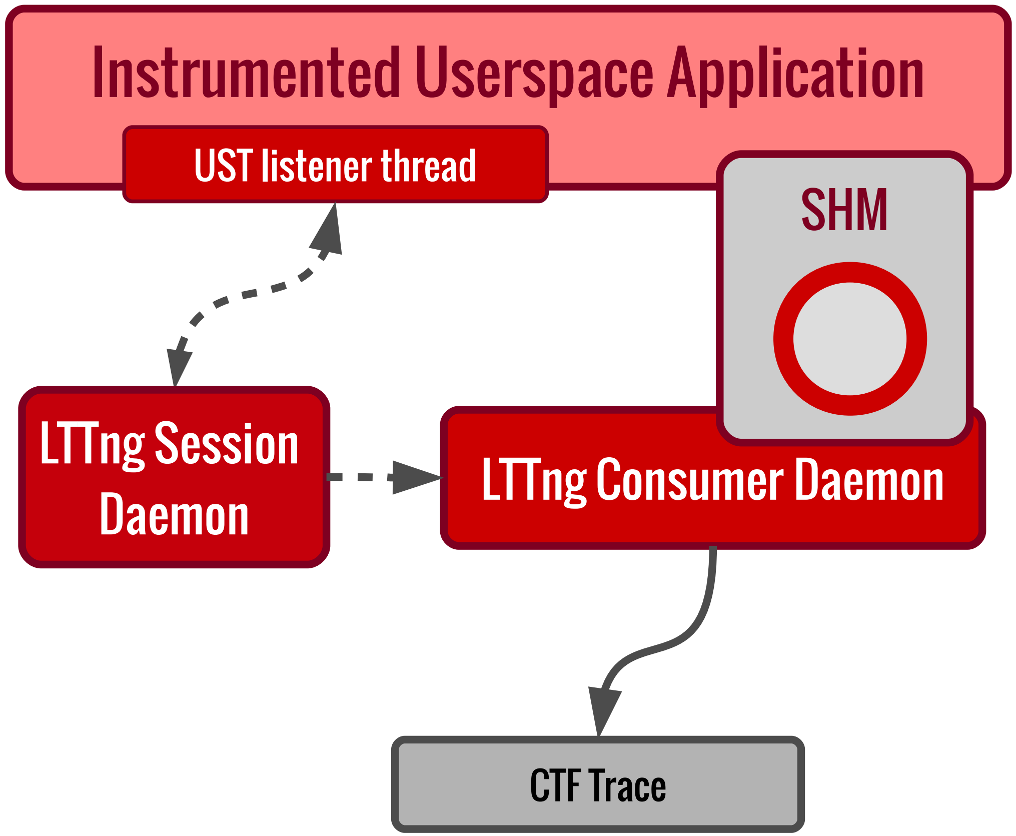
$$\downarrow$$

?

LTTng's

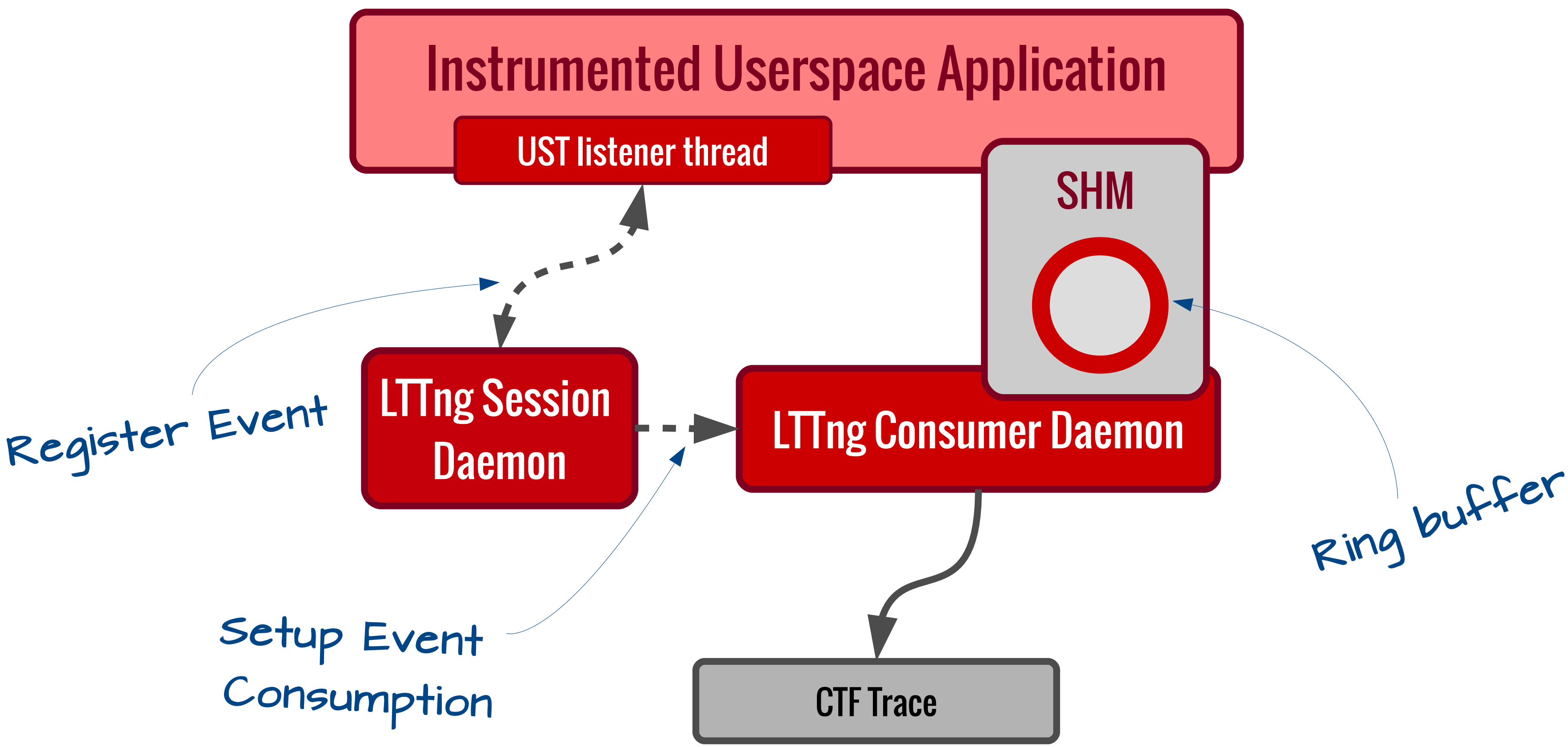
Trace Filtering

LTTng-UST

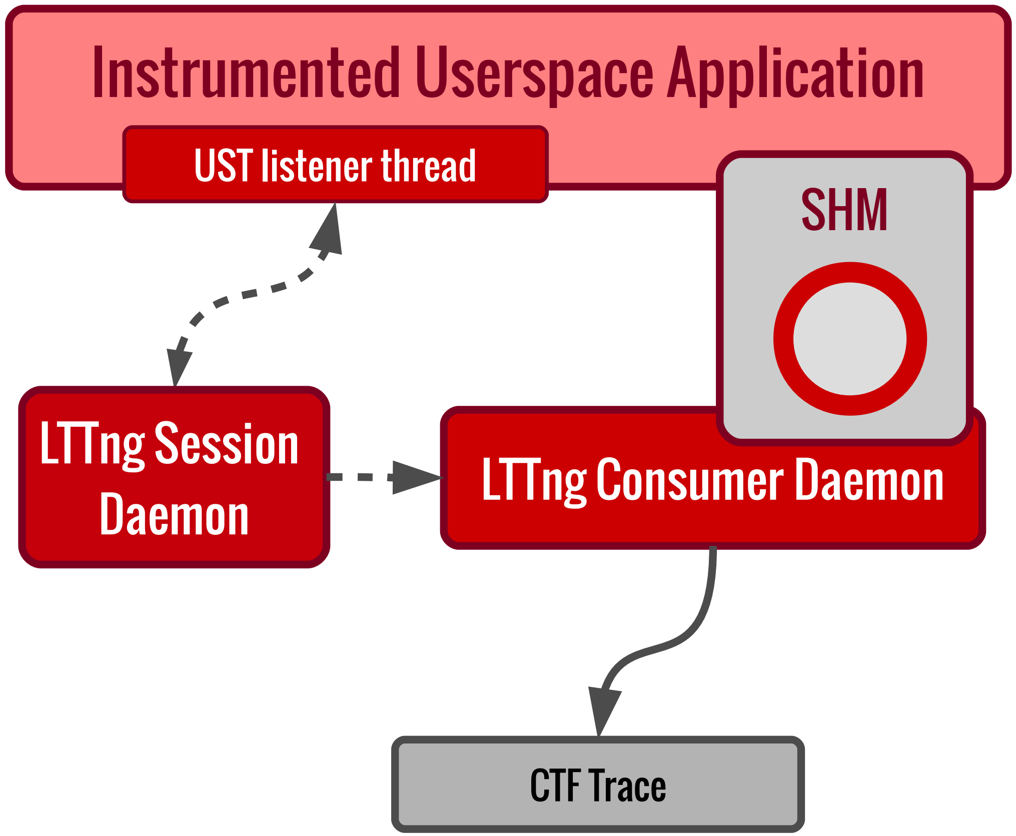


Instrumented Userspace Application
UST listener thread
SHM
LTTng Session Daemon
LTTng Consumer Daemon
CTF Trace

LTTng-UST

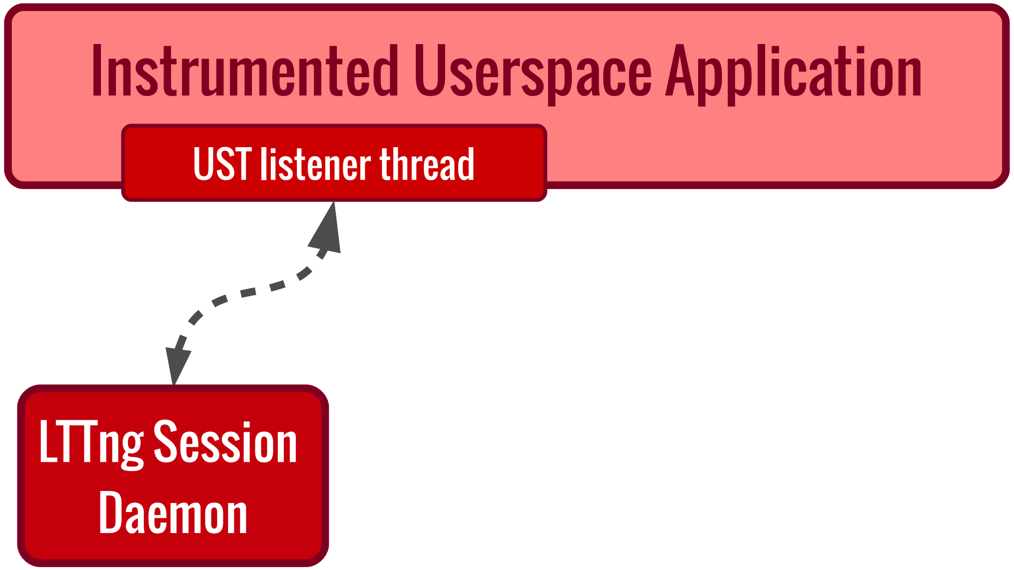

Instrumented Userspace Application
UST listener thread
SHM
Register Event
LTTng Session Daemon
LTTng Consumer Daemon
Ring buffer
Setup Event Consumption
CTF Trace

LTTng-UST

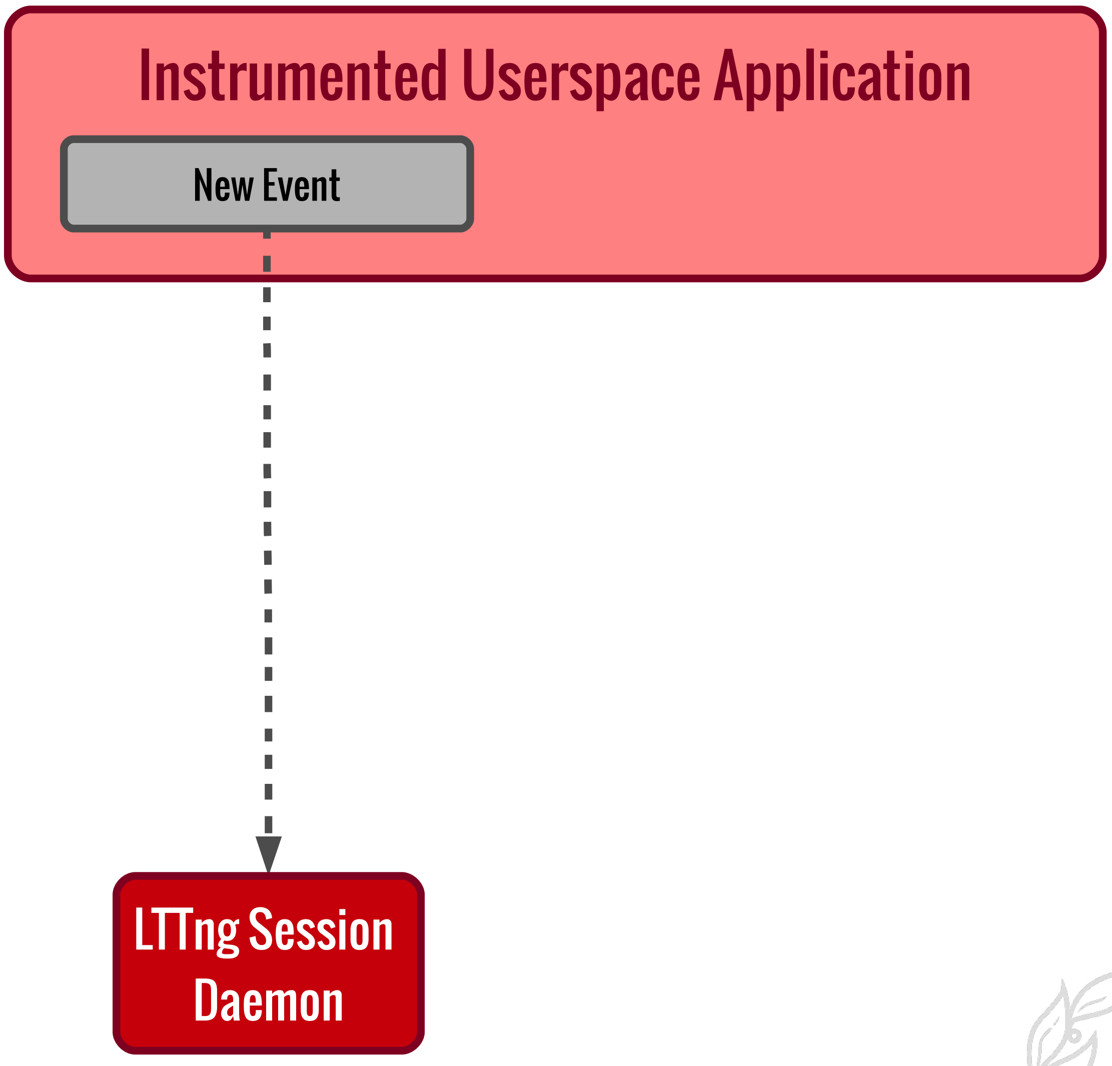


Instrumented Userspace Application
UST listener thread
SHM
LTTng Session Daemon
LTTng Consumer Daemon
CTF Trace

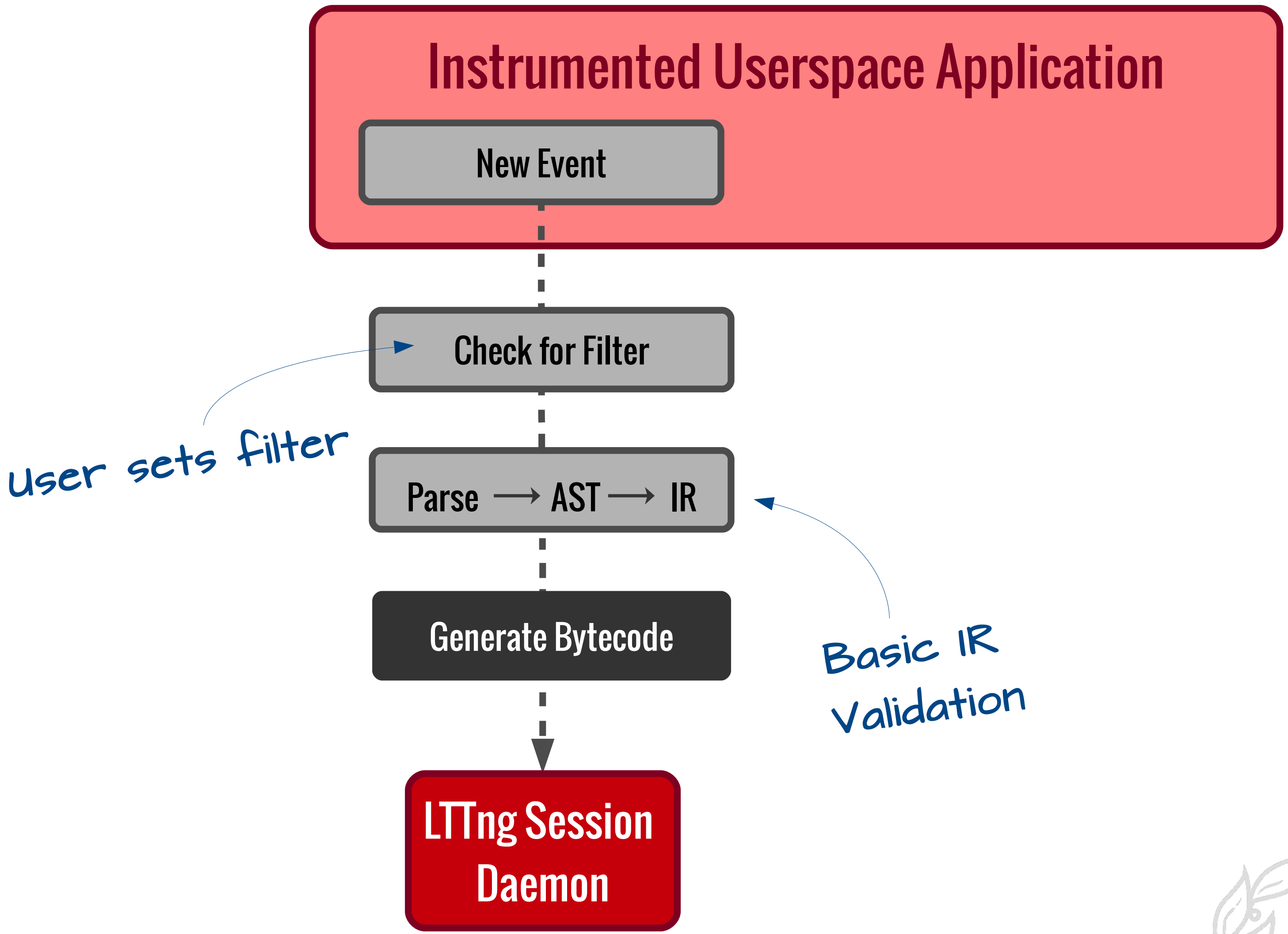
LTTng-UST


Instrumented Userspace Application
UST listener thread
LTTng Session Daemon

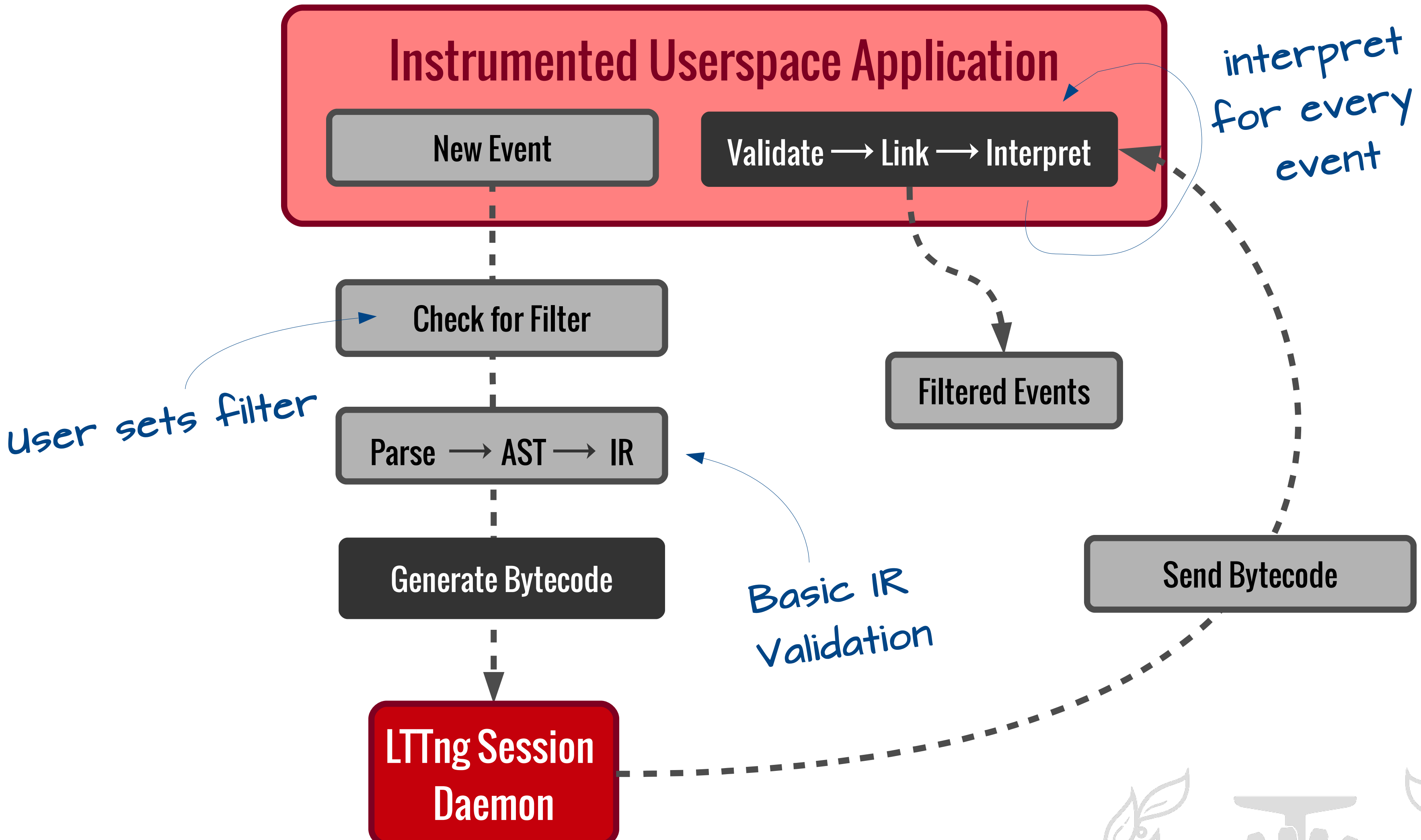
LTTng-UST Filtering



LTTng-UST Filtering



LTTng-UST Filtering



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
```

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
```

Generating Bytecode

Filter Bytecode Generation

`generate_filter()`

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

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

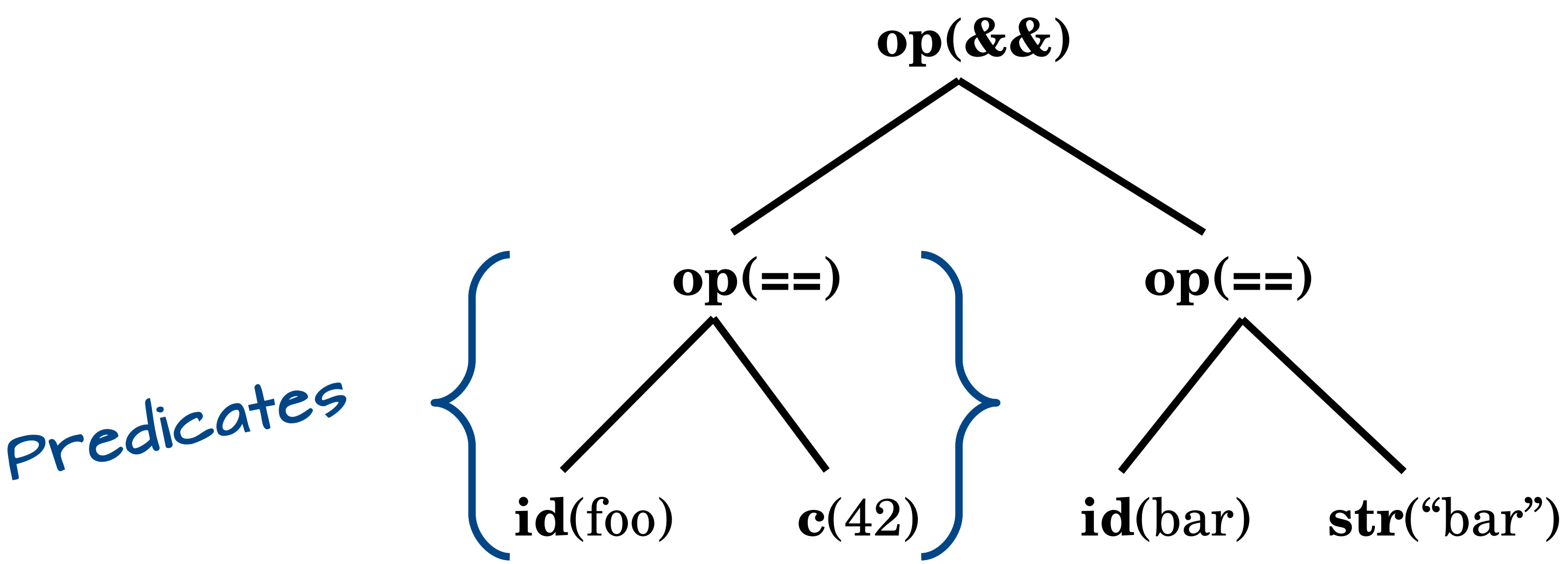
- Allocate/initialize parser, AST, create root node

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

- Run `yyparse()`, `yylex()`
- Generate syntax tree

Filter Bytecode Generation

Syntax Tree



Filter Bytecode Generation

```
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

```
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

```
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

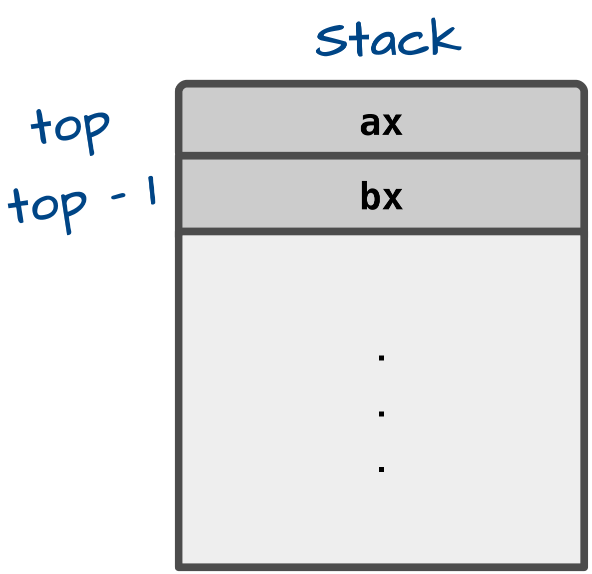
```
lttng_filter_specialize_bytecode(runtime);
```

- 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)




```
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

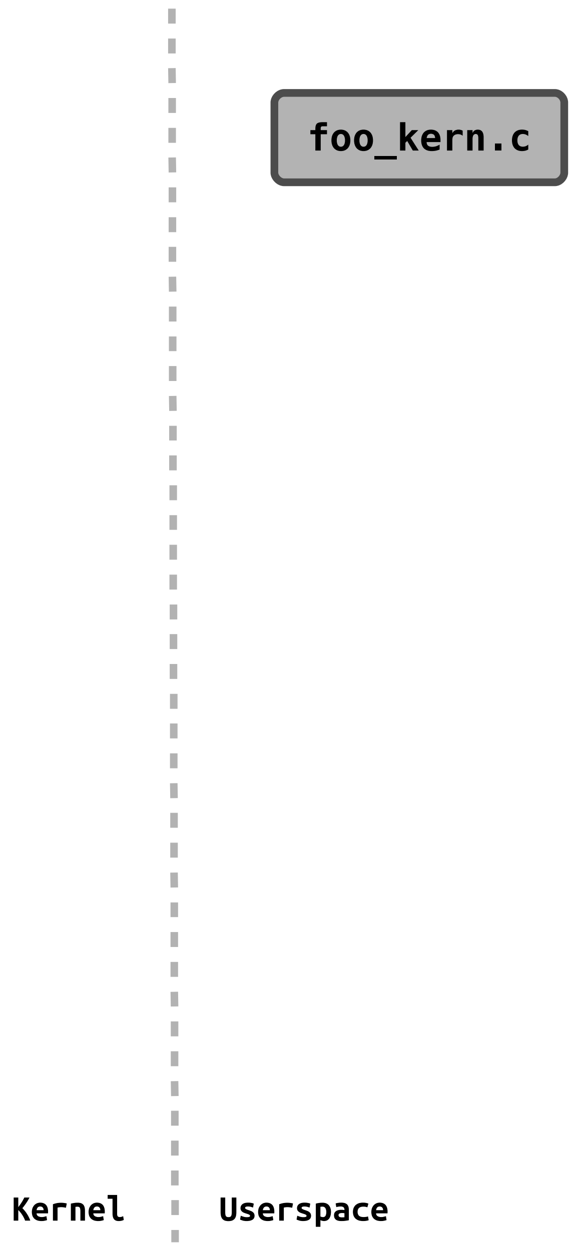
Berkeley Packet Filter (BPF)

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

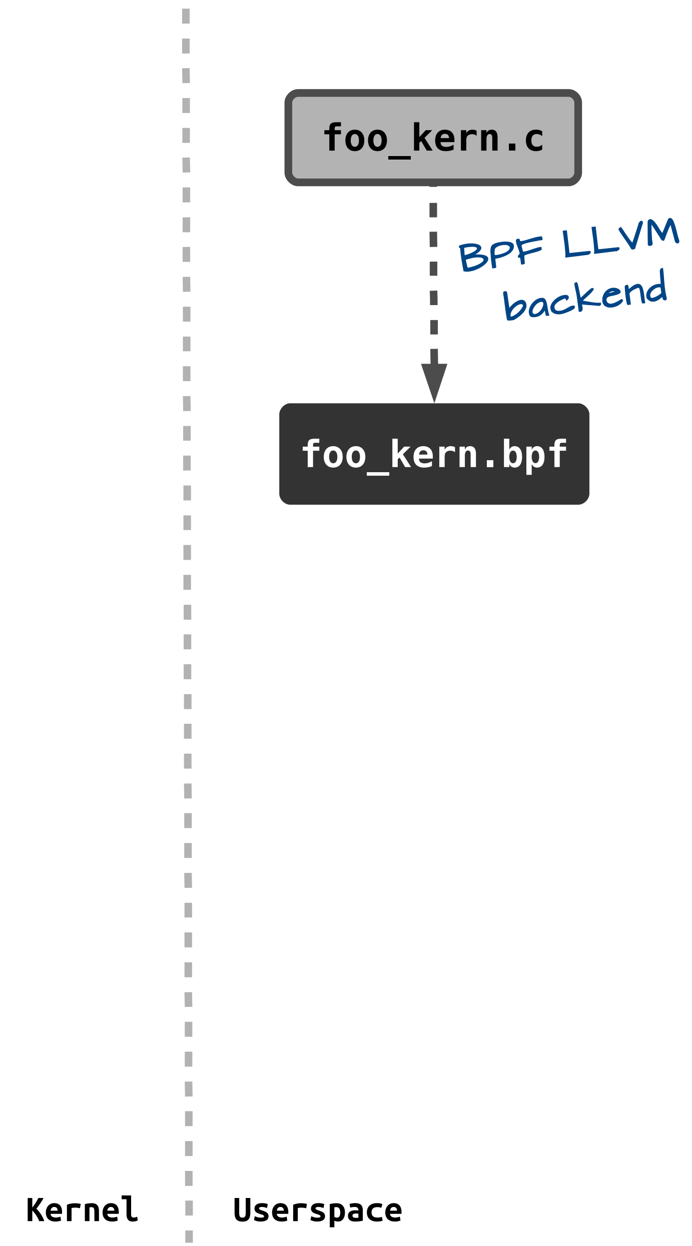
Current Status of BPF

- 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

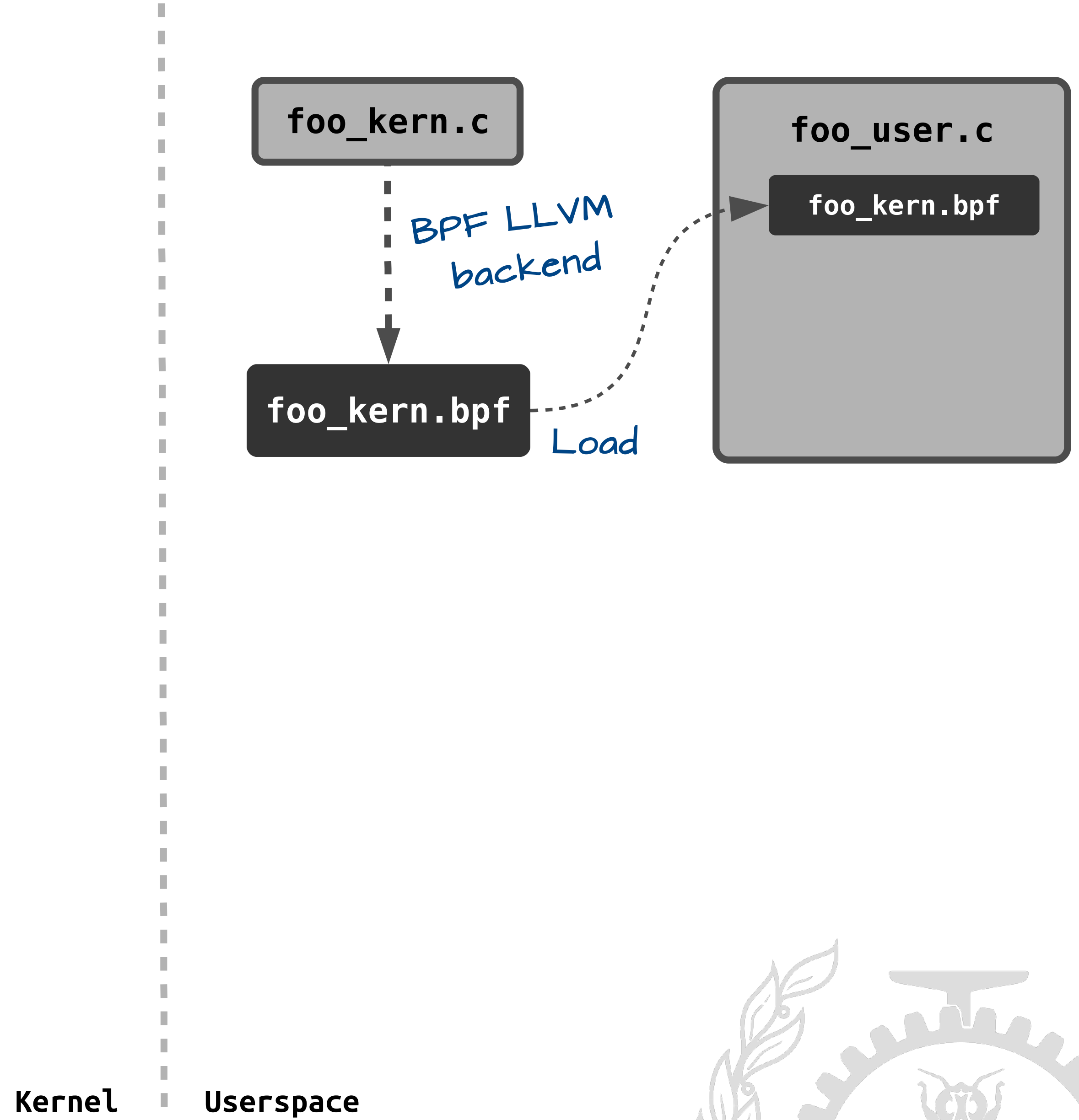
Example eBPF Session



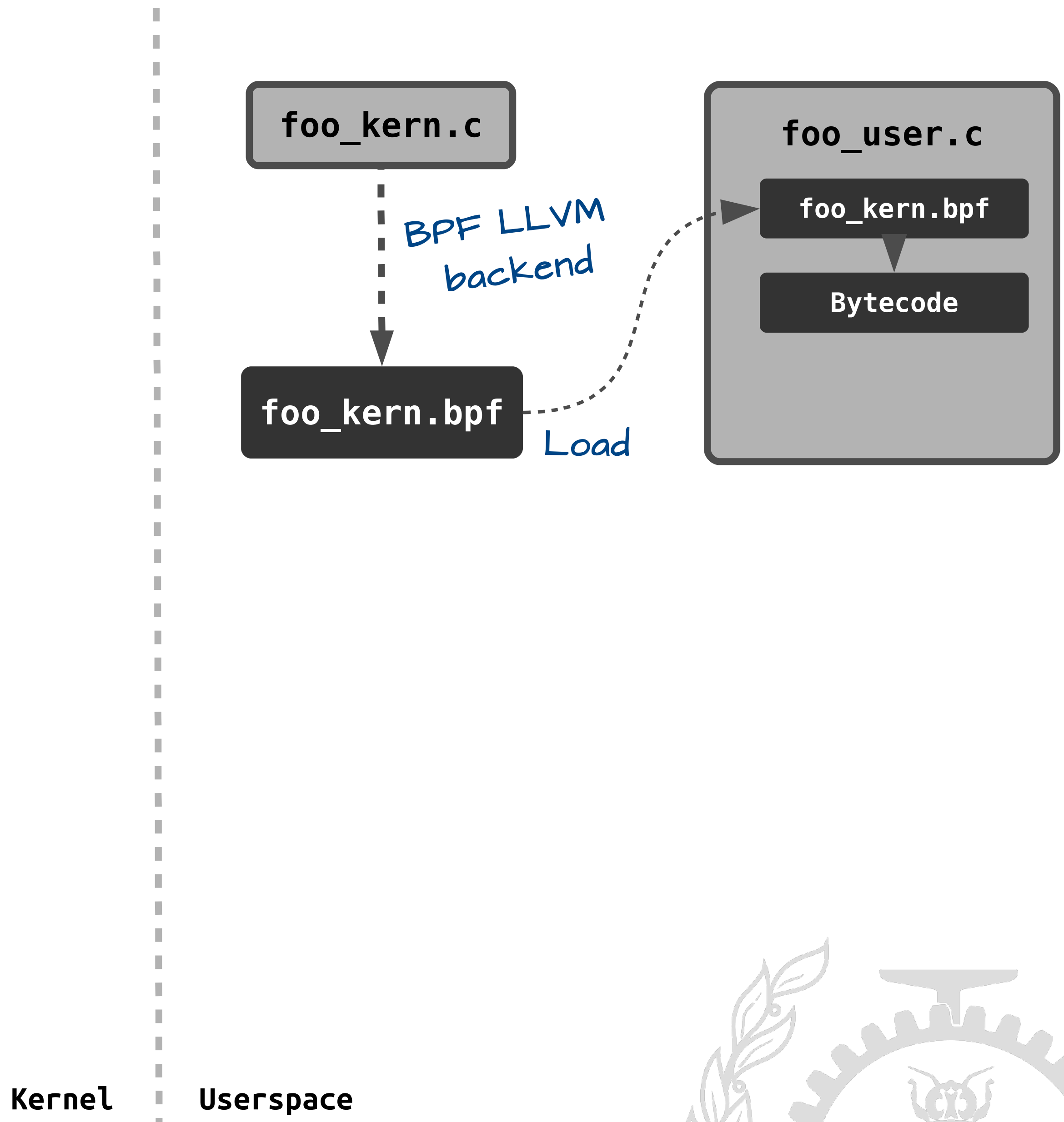
Example eBPF Session



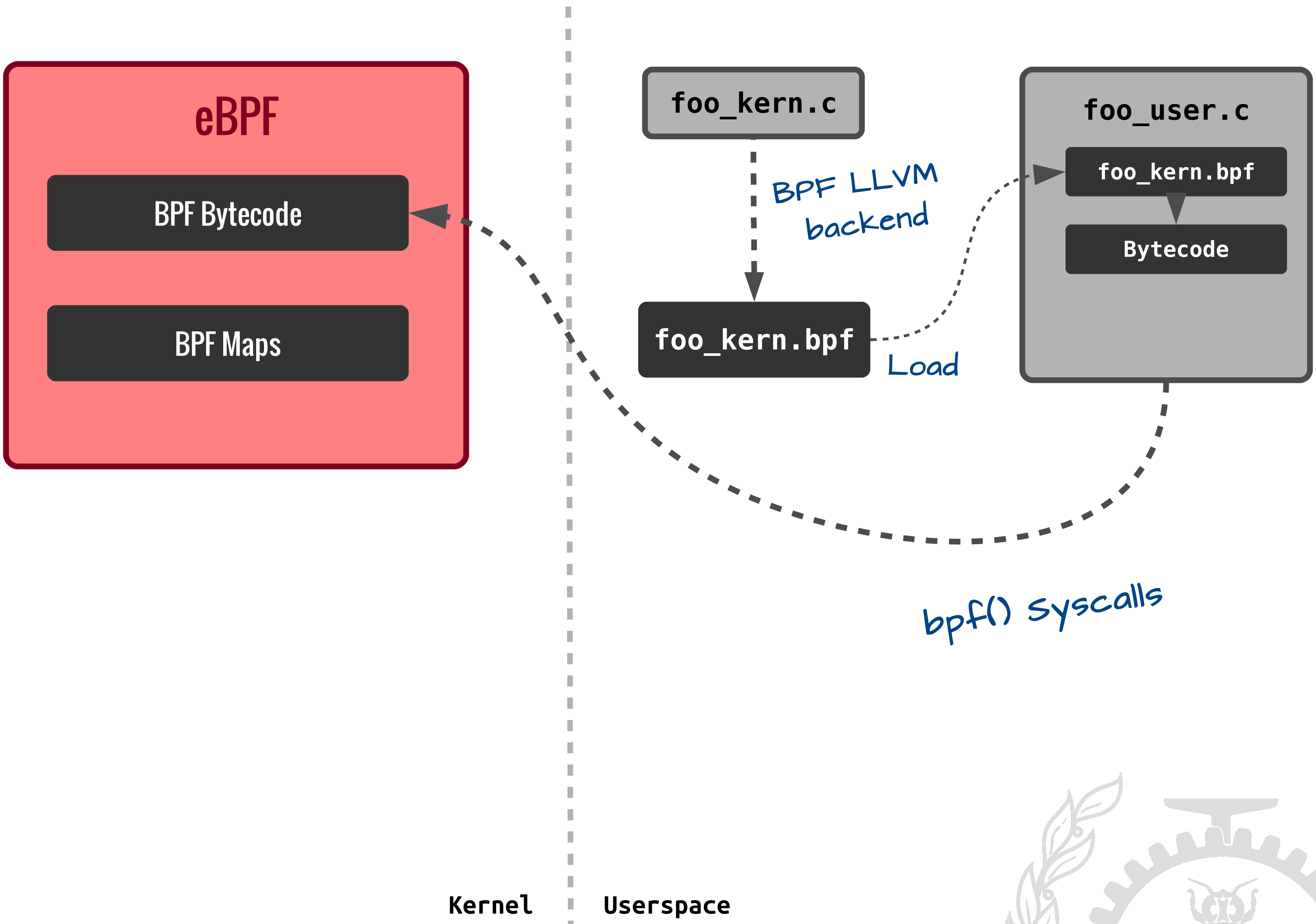
Example eBPF Session



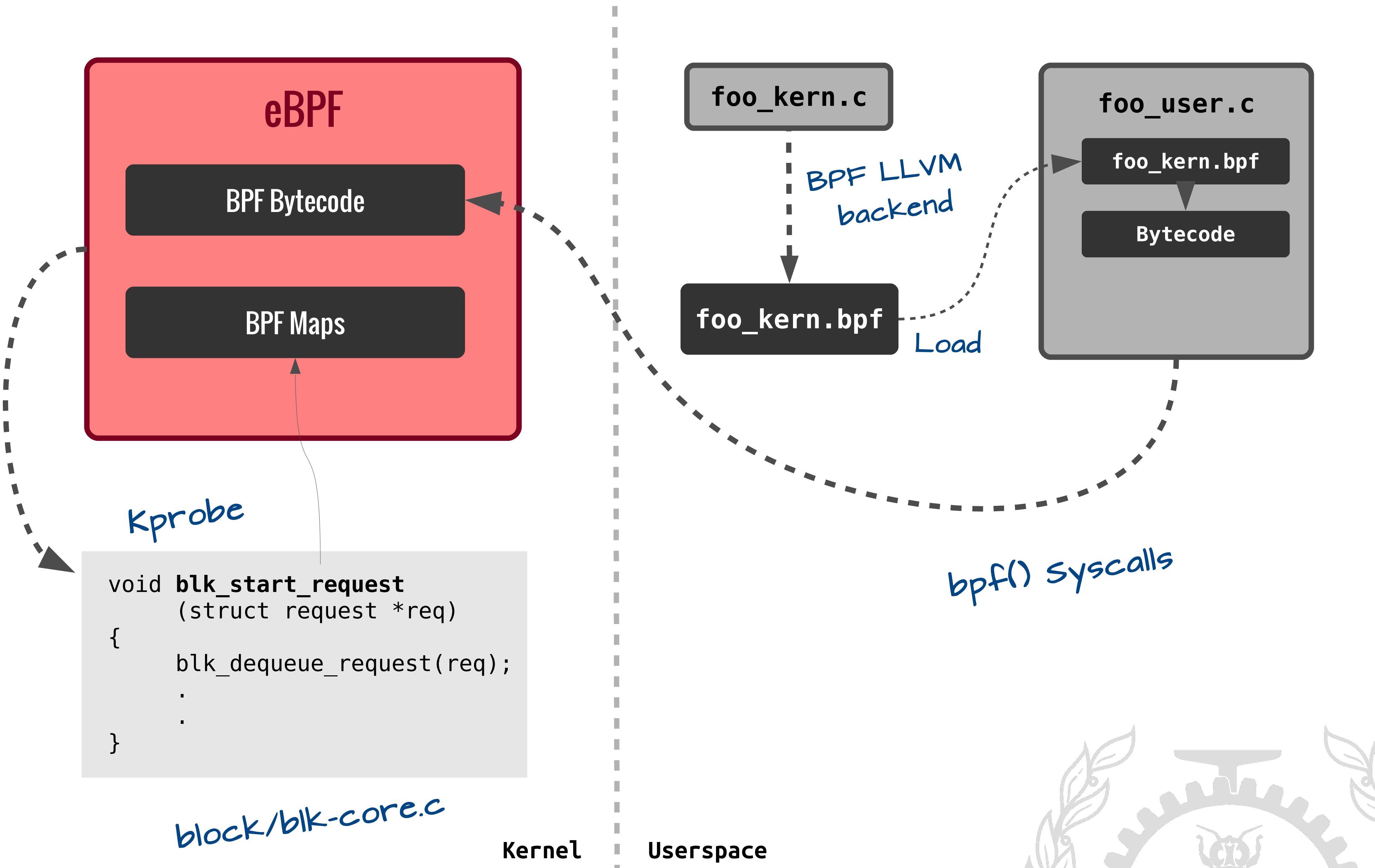
Example eBPF Session



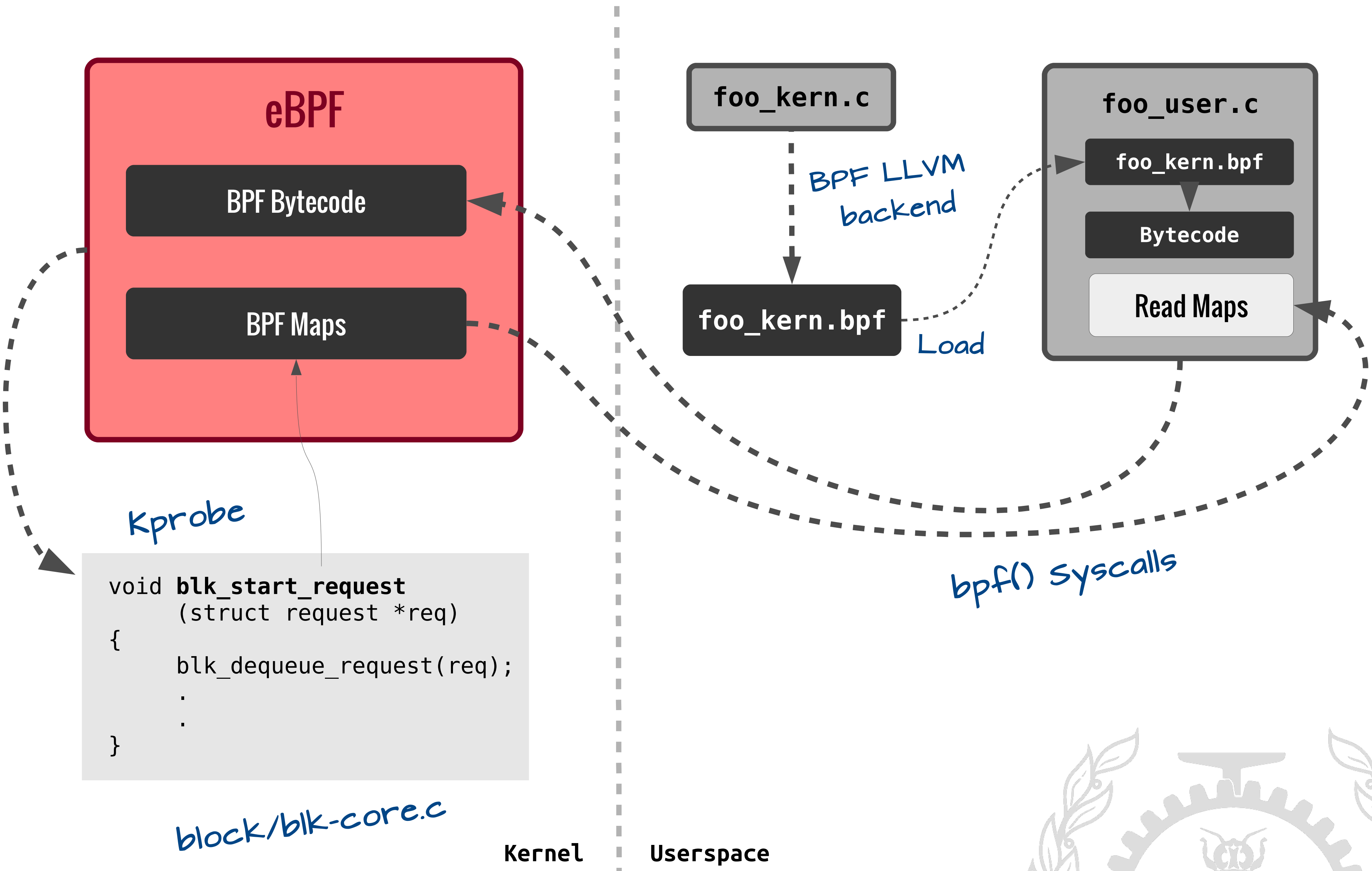
Example eBPF Session



Example eBPF Session



Example eBPF Session



Sample eBPF Filter

eBPF Filter on LTTng Kernel Event

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

eBPF Bytecode :

R2 = ctx

R3 = *(dev->name)
R4 = 0x6f6c

```
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)
        mov    %r14,-0x218(%rbp)
        mov    %r15,-0x210(%rbp)
 27:    xor    %eax,%eax
 29:    xor    %r13,%r13
        mov    0x0(%rdi),%rsi
        mov    0x0(%rsi),%rdx
        mov    0x8(%rdi),%rcx
 38:    cmp    %rcx,%rdx
```

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

Make some space on stack

Save callee saved regs

Clear A and X

Compare R3, R4

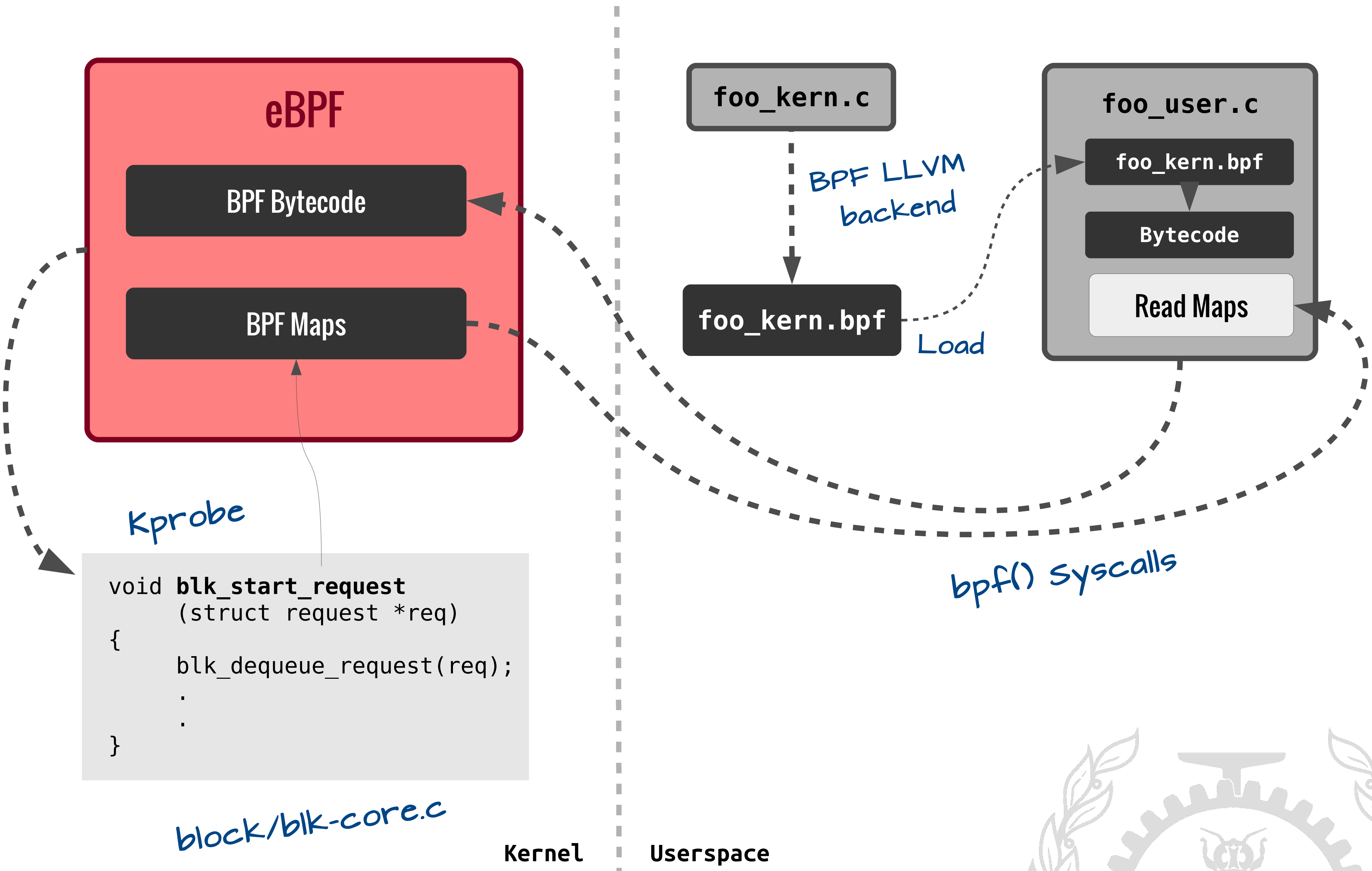
Load ctx args to R3 and R4

Jump to **TRUE**

Restore regs

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

Example eBPF Session

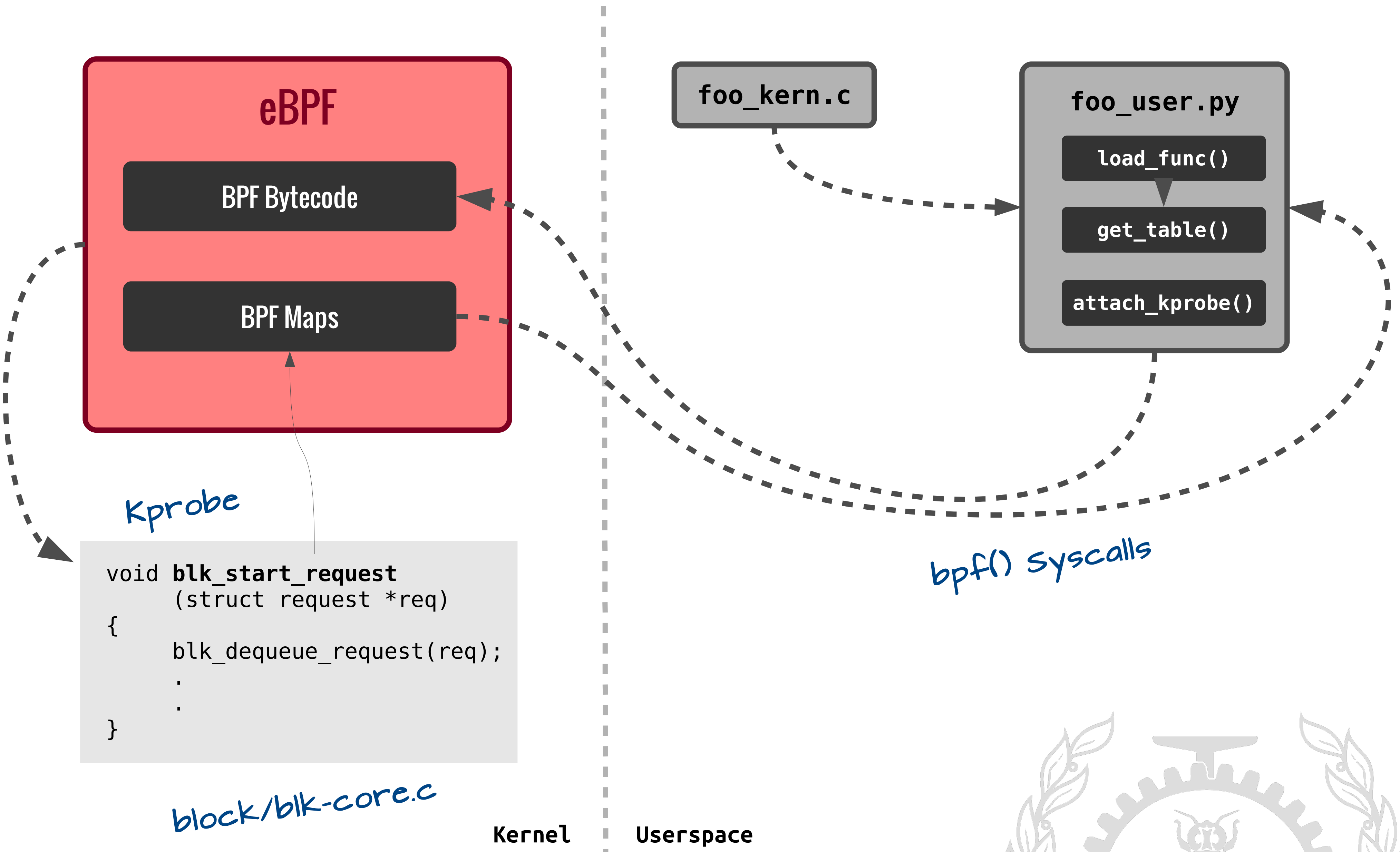


BPF COMPILER COLLECTION

Yes, `'bcc'` **exists!**

https://github.com/iovisor/bcc

Example bcc Session



Example bcc Session

task_switch.c

```
#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;
}
```

Kernel side BPF program

task_switch.py

```
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))
```

eBPF

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)

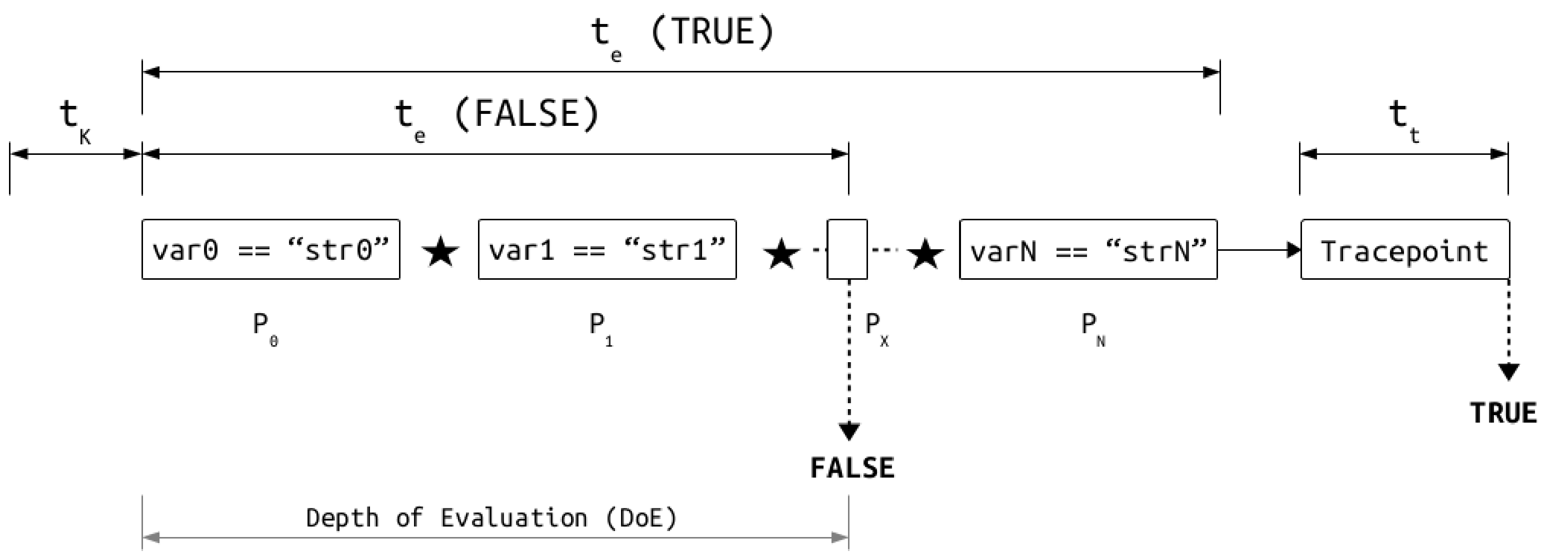
- Experimental *libebpf* to provide filtering in userspace tracing
- Includes side-effects through communication with modified KeBPF
- Easy switch between JIT/interpret for performance analysis
- Includes LLVM BPF backend.
- Load bytecode from eBPF binaries

Performance Analysis

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

Experiments

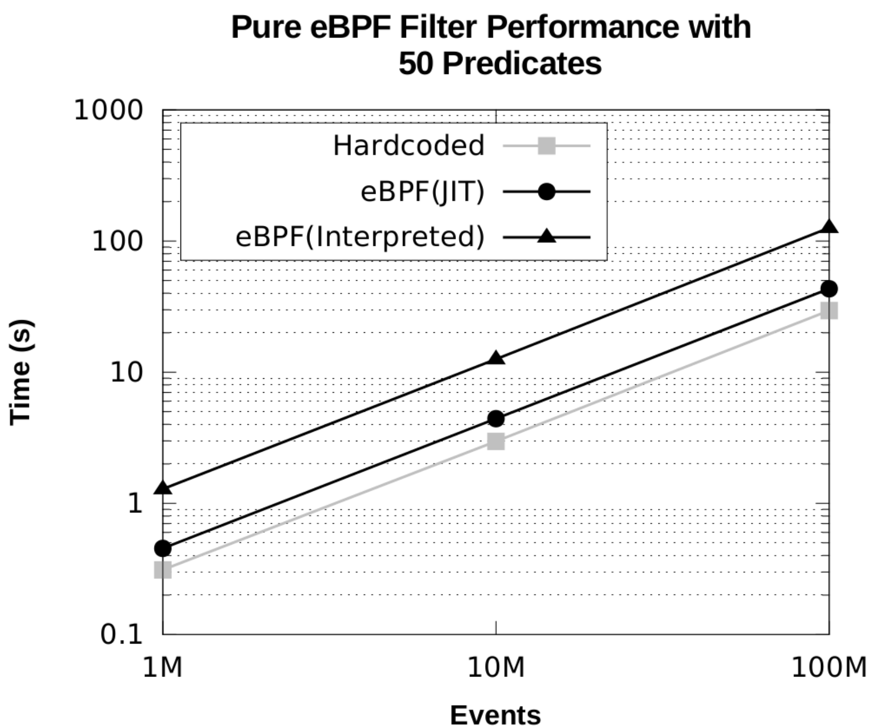
Performance Analysis

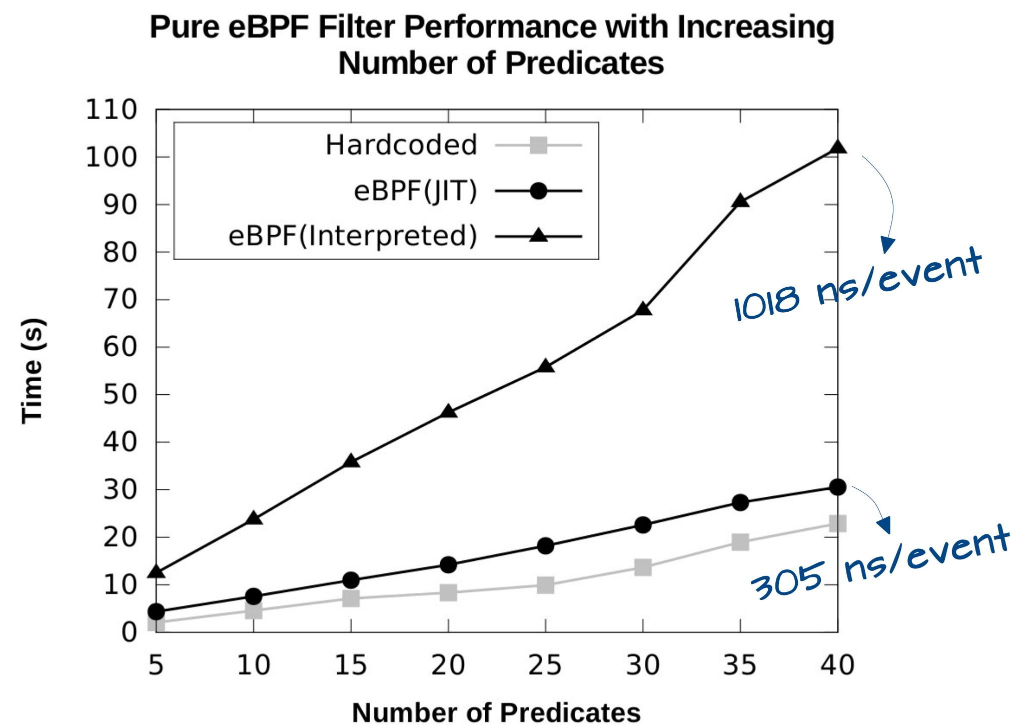


- Pure filter evaluation.
  - TRUE/FALSE biased AND chain with varying predicates
- Measure $t_e + t_t$ with varying *DoE* (Biased TRUE)

Experiments

Performance Analysis

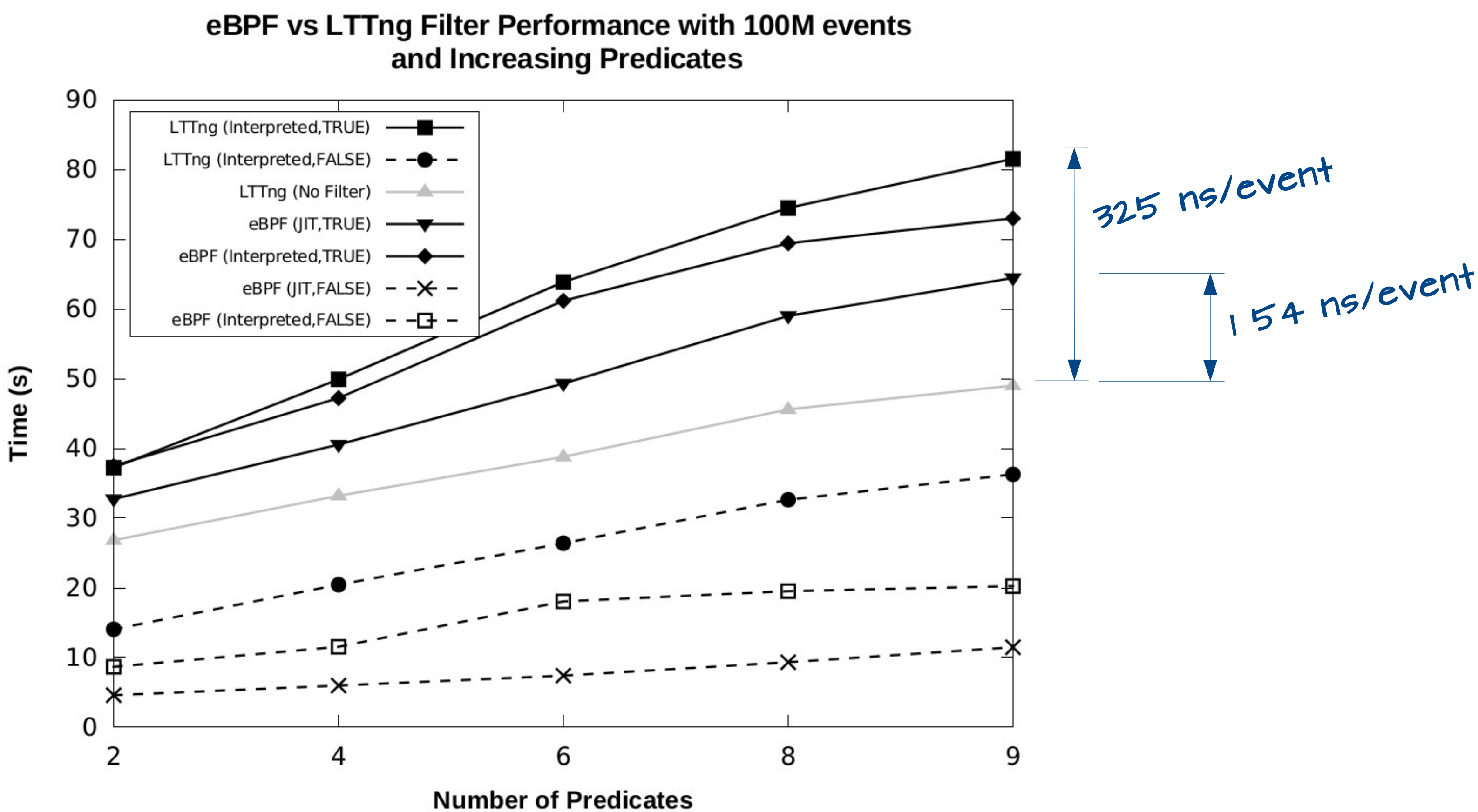




- Steady gain in **3x** range for JIT vs Interpreted with increasing events (**3.1x** to **3.3x**)

Experiments

Performance Analysis



- eBPF JIT*ed* filter is **3.1x** faster than LTTng's interpreted bytecode and eBPF's interpreted filter is **1.8x** faster than LTTng's interpreted version

Learnings

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

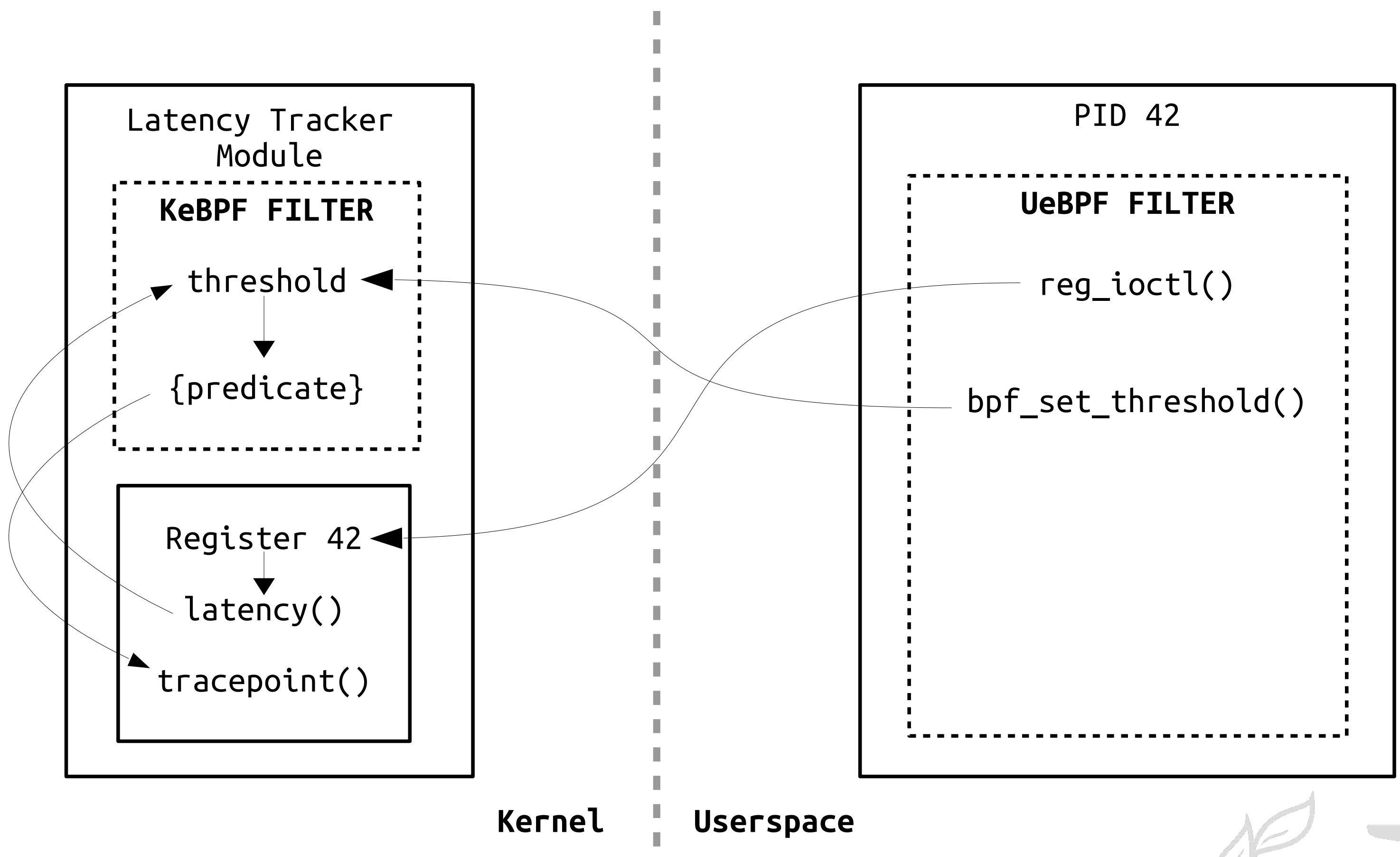
Beyond

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

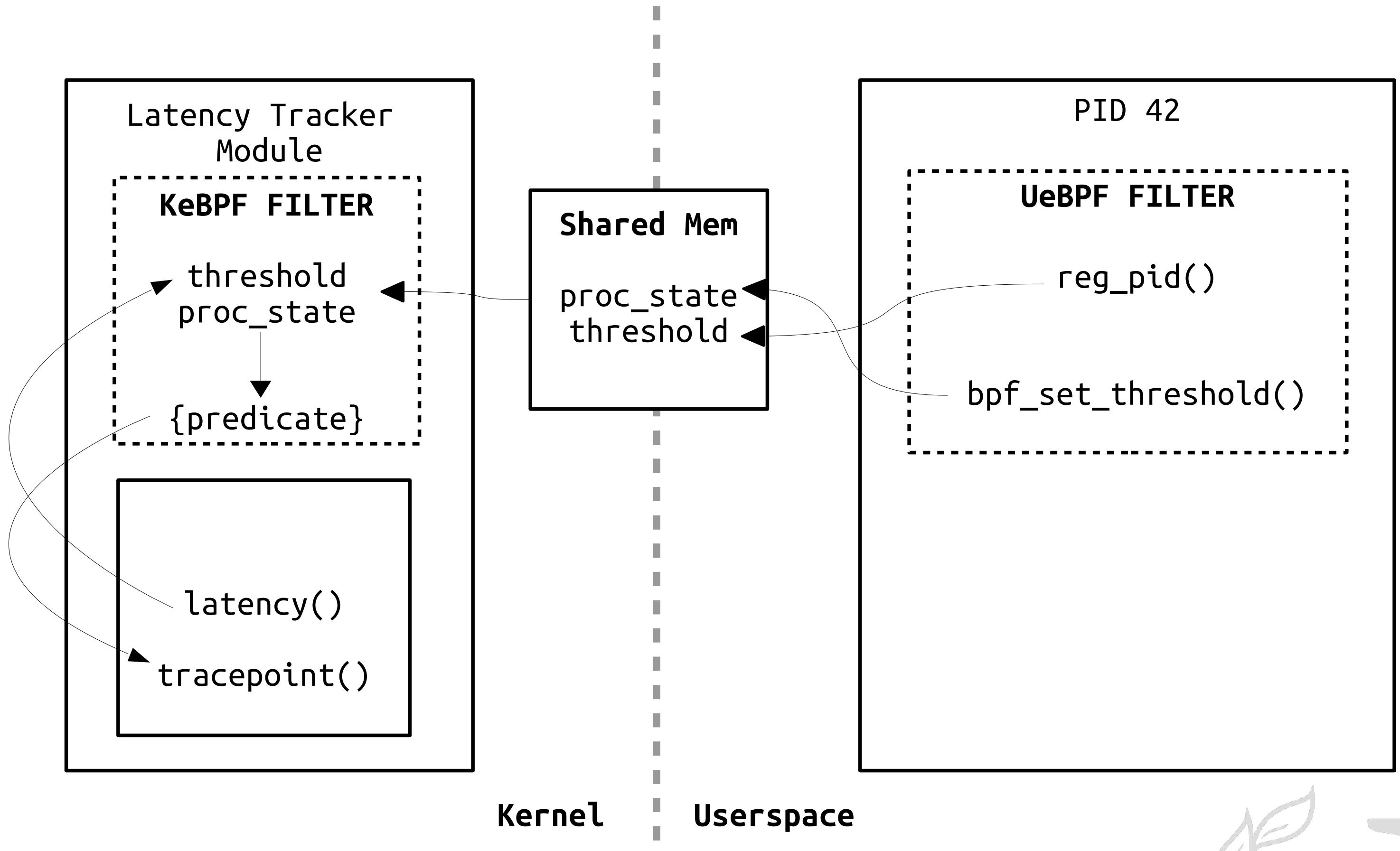
Beyond

KeBPF ↔ UeBPF Syscall Latency Tracking



Beyond

KeBPF ↔ UeBPF Syscall Latency Tracking



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

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