bpftrace Cheatsheet

smallnest

1 Basic Syntax

Concept	Description
Probe	Format: provider:probe
Action	Code within { ... }
Built-in Variables	$1, $2 (arguments), pid, tid, uid, gid, comm, cpu, curtask, rand, args, arg0, arg1, ..., argN (function arguments)
Maps	@name[key] = value;
Filtering	/filter/ { actions }

2 Common Probes

Probe	Description
kprobe:function	Kernel function entry
kretprobe:function	Kernel function return
uprobe:/path/to/binary:function	User-space function entry
uretprobe:/path/to/binary:function	User-space function return
tracepoint:subsystem:event	Static tracepoint
profile:hz:rate	Periodic sampling
interval:ms:rate	Timer
software:event:count	Software events (e.g., cpu-clock, task-clock)
hardware:event:count	Hardware events (e.g., cache-misses, cpu-cycles)

3 Listing Probes

Command	Description
`bpftrace -l`	List all probes
`bpftrace -l "tracepoint:*"`	List all tracepoint probes
`bpftrace -l "kprobe:vfs_*"`	List all vfs-related kprobe probes
`bpftrace -lv "tracepoint:syscalls:sys_enter_*"`	List all sys_enter-related tracepoint probes and their arguments

4 Built-in Functions

Function	Description
printf(format, args...)	Print formatted string
time(format)	Current time, formatted output
count()	Counter
sum(int n)	Sum
avg(int n)	Average
min(int n)	Minimum
max(int n)	Maximum
hist(int n)	Histogram
lhist(int n, int min, int max, int step)	Linear histogram
kstack(), ustack()	Kernel and user stack trace
ntop(int n)	Convert IP address to string
reg(char *name)	Read CPU register value

5 Advanced Features

Feature	Description
Wildcards	Use * in probe definitions (e.g., kprobe:vfs_*)
Frequency Counting	Use count() with maps for frequency analysis
Associative Arrays	@map[key1, key2, ...] = value;
Strings	Use str() for conversion, strcmp() for comparison
Aggregations	clear(@map), print(@map), zero(@map)
Join	join(char *arr[]) to concatenate array elements
Timestamps	nsecs, elapsed

6 Examples

6.1 System Call Count

```
#!/usr/bin/env bpftrace

tracepoint:raw_syscalls:sys_enter
{
    @syscalls[comm] = count();
}

interval:s:5
{
    print(@syscalls);
    clear(@syscalls);
}
```

6.2 Function Latency Measurement

```
#!/usr/bin/env bpftrace

kprobe:vfs_read
{
    @start[tid] = nsecs;
}
```

```
kretprobe:vfs_read
/@start[tid]/
{
    @duration = hist(nsecs - @start[tid]);
    delete(@start[tid]);
}

interval:s:10
{
    print(@duration);
    clear(@duration);
}
```

6.3 Memory Allocation Tracking

```
#!/usr/bin/env bpftrace

uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc
/comm == "target_process"/
{
    @bytes[ustack] = sum(arg0);
}

interval:s:30
{
    print(@bytes);
    clear(@bytes);
}
```

7 One-Liners

bpftrace One-Liner	Explanation
`bpftrace -e 'tracepoint:syscalls:sys_enter_* { @[probe] = count(); }'`	Count all system calls
`bpftrace -e 'kprobe:vfs_read { @bytes = sum(arg2); }'`	Sum of bytes read
`bpftrace -e 'kprobe:vfs_write { @bytes[comm] = sum(arg2); }'`	Sum of bytes written by process
`bpftrace -e 'tracepoint:block:block_rq_issue { @[args->dev] = count(); }'`	Count I/O requests per block device
`bpftrace -e 'software:major-faults:1 { @[comm] = count(); }'`	Count major page faults per process
`bpftrace -e 'kprobe:do_nanosleep { @[comm] = count(); }'`	Count sleeps per process
`bpftrace -e 'tracepoint:sched:sched_process_exec { @[args->filename] = count(); }'`	Count executions per binary
`bpftrace -e 'kprobe:tcp_retransmit_skb { @[sock_ip(args->sk)] = count(); }'`	Count TCP retransmits per IP
`bpftrace -e 'tracepoint:irq:irq_handler_entry { @[args->name] = count(); }'`	Count calls per interrupt handler
`bpftrace -e 'kprobe:kmalloc { @bytes = hist(arg1); }'`	Histogram of kernel memory allocations
`bpftrace -e 'kprobe:do_sys_open { @[comm] = count(); }'`	Count file opens per process

`bpftrace -e 'uprobe:/lib/x86_64-linux-gnu/libc.so.6:malloc { @bytes = hist(arg0); }'`	Histogram of user-space memory allocations
`bpftrace -e 'profile:hz:99 @[comm] = count(); '`	Samples 99 times per second and counts the number of calls for each process name.
`bpftrace -e 'interval:s:5 printf("Every 5 seconds"); '`	Prints "Every 5 seconds" every 5 seconds.

8 Best Practices

- Use filters to reduce overhead and focus on relevant events
- Combine multiple probes to correlate events
- Use maps and aggregations for efficient data collection
- Be cautious with stack traces and string operations (high overhead)
- Test scripts on non-production systems first
- Use interval probes to periodically output and clear data