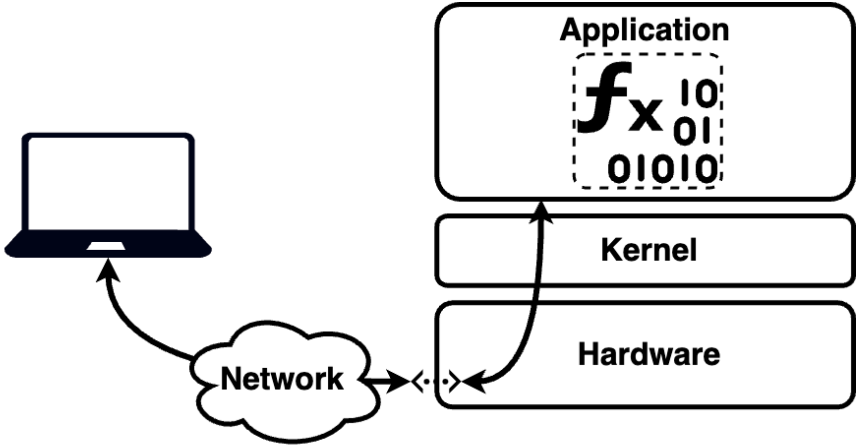
Tutorial: Observability into Application-level Metrics with ***eBPF***

Mohammadreza Rezvani, Muntaka Ibnath, Daniel Wong

2025 IEEE International Symposium on Workload Characterization

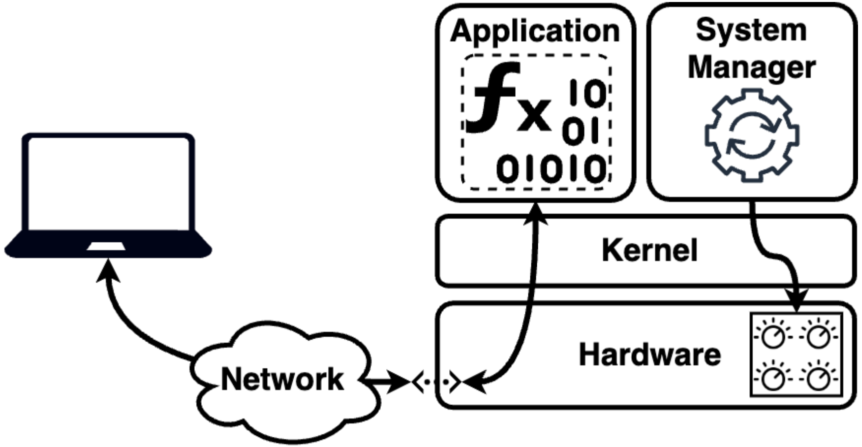
Modern Data Centers

- Dynamic environment of data centers, demands ***efficient management***.



System Management

- Dynamic environment of data centers, demands ***efficient management***.
- System manager tune performance knobs
  - ***Cores, Frequency, Cache***, …
- System manager requires ***observability*** into applications.

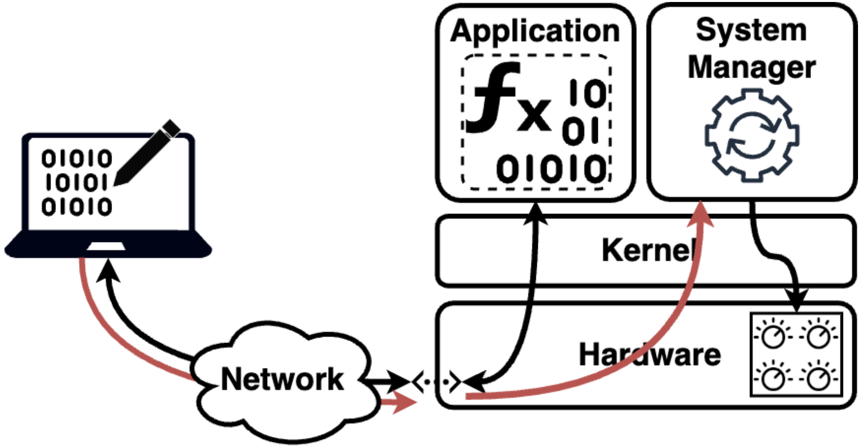


Observability Challenges

I. Observabilities fulfilled ***by clients*** as performance metrics.

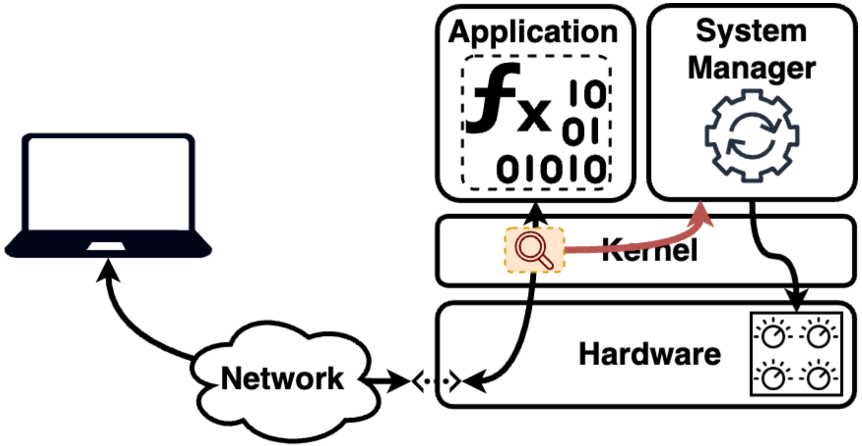
II. Application must be ***instrumented*** to provide observability.

III. Timely action incurs significant ***overhead***.



Idea: Increase Observability From Kernel Interactions

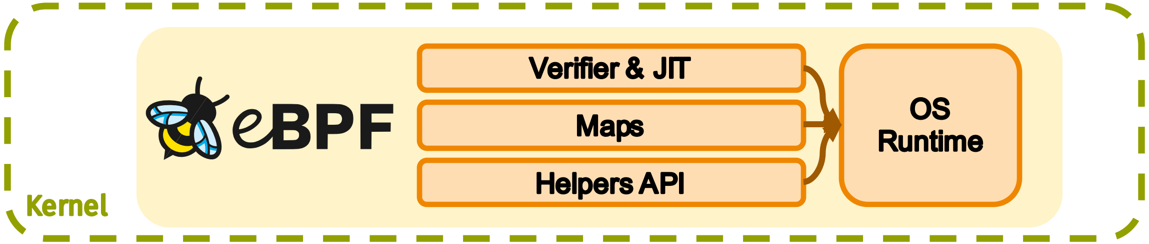
- Benefits of observing ***kernel interactions***
  - ***No need for instrumentation***
  - ***Open doors for data to be used in kernel***



How To Observe Kernel Activities?

- Run sandboxed programs in kernel*

- eBPF allows safe custom programs to be attached to parts of kernels in order to monitor the event, gather information, or do somethings based on the event



*ebpf.io

Emerging eBPF Use Cases

- Connection security between application deployed using container management
- eBPF based OS scheduler (sched_ext)
- Load balancing
- Anomaly detection

Today's agenda

Step 1:

- Gain high-level understanding of ***Linux kernel***
- Learn about ***eBPF***, its *benefits*, and some *use cases*

Step 2:

- Dissect and run **eBPF** programs

Step 3:

- Introduction to Server-Client Workloads

Step 4:

- Tracing syscalls using kprobes

Step 5:

- Tracing user-space functions using uprobes

What is eBPF?

What is eBPF?

extended

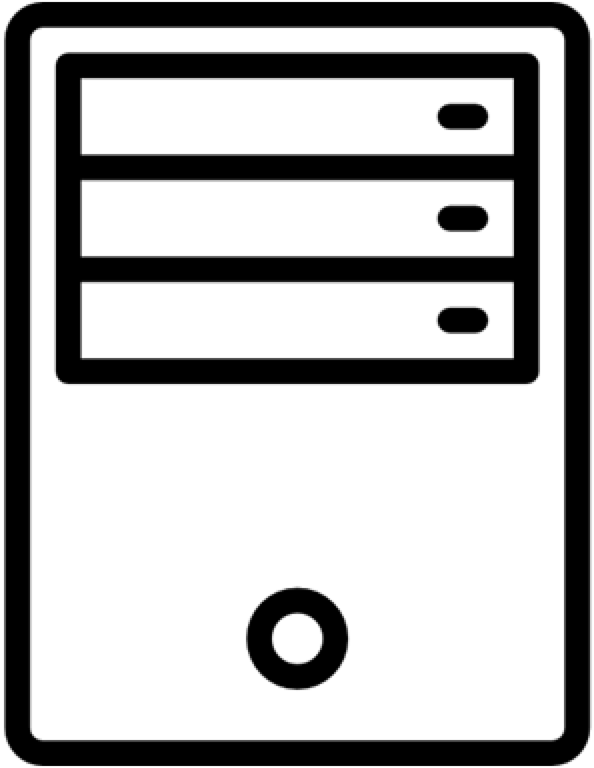
Berkeley

Packet

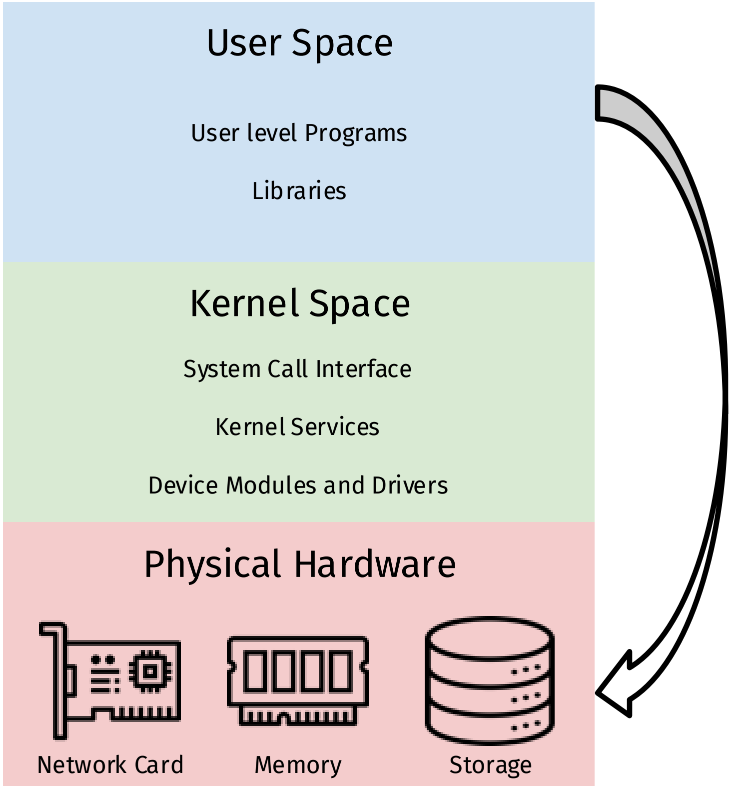
Filter

Source of picture: ebpf.io/blog/ebpf-for-all/

Operating system fundamentals

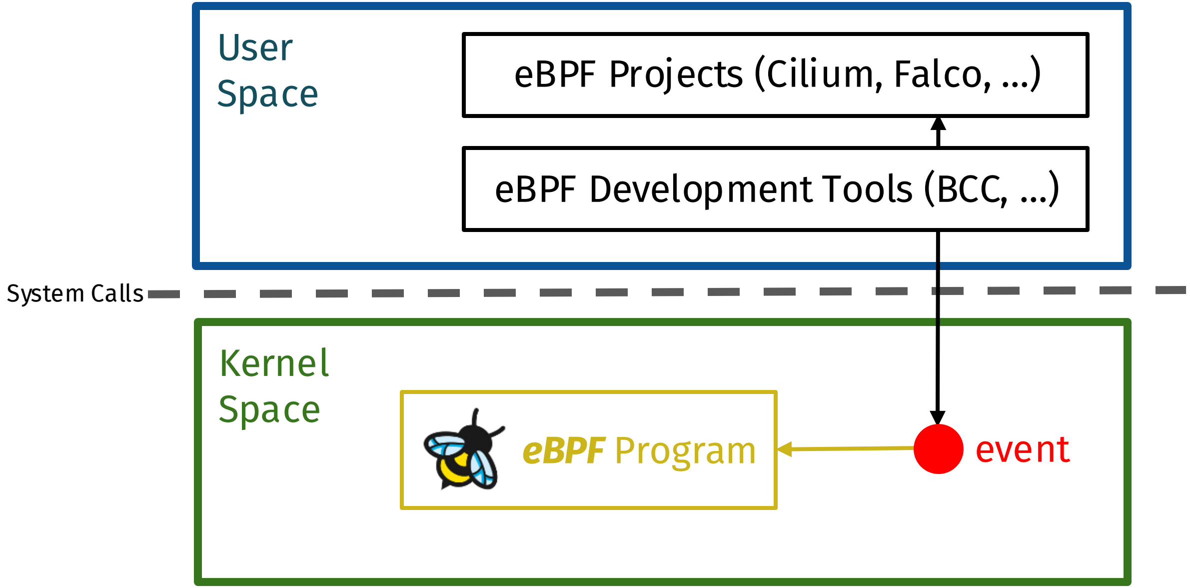


Operating system fundamentals



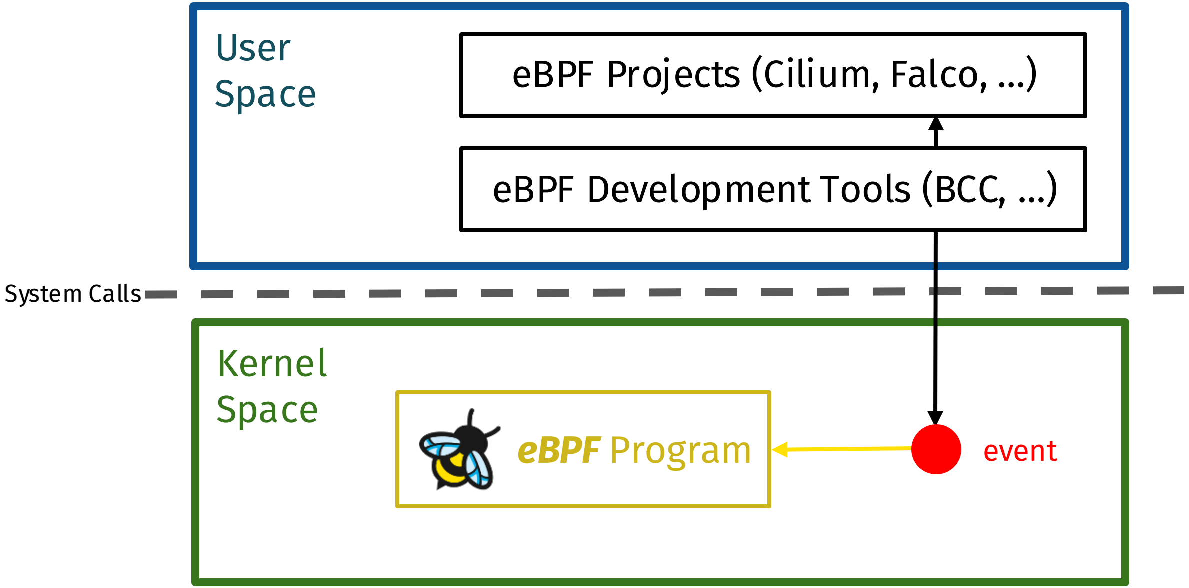
So what if we have to easily extend the capabilities of kernel?

eBPF



eBPF programs can be attached to different events

- Kprobes
- Uprobes
- Tracepoints
- Network packets
- Security Modules
- Perf events
- ...



Emerging *eBPF* use cases

- Connection security between application deployed using container management
- eBPF-based OS scheduler
- Load balancing
- Anomaly detection

Setup environment

Scan the QR code or visit the link for server access

https://forms.gle/w717jHchTXiVhw2Y8

- You will receive an email with credentials
- ***Please let us know before submitting another form***

How to use command line:

- https://tinyurl.com/cmdlinetutorial

Please ssh to the server after receiving the credentials.

Tutorial repository

https://github.com/socal-ucr/eBPF-tutorial

- Clone the repository

How to use git:

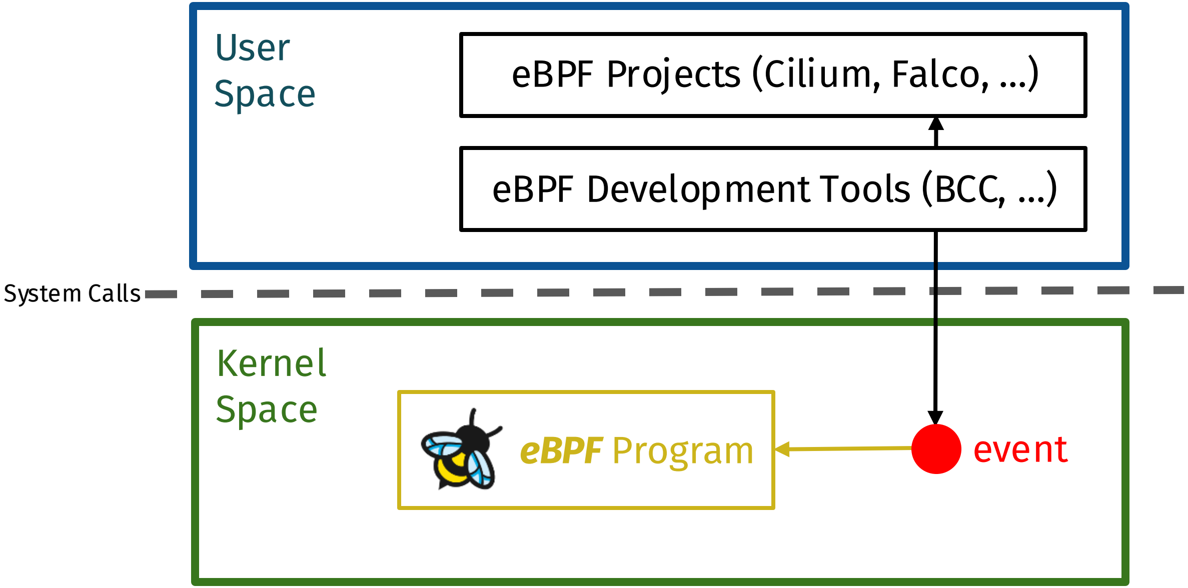
- https://tinyurl.com/git-tut

How to use vim:

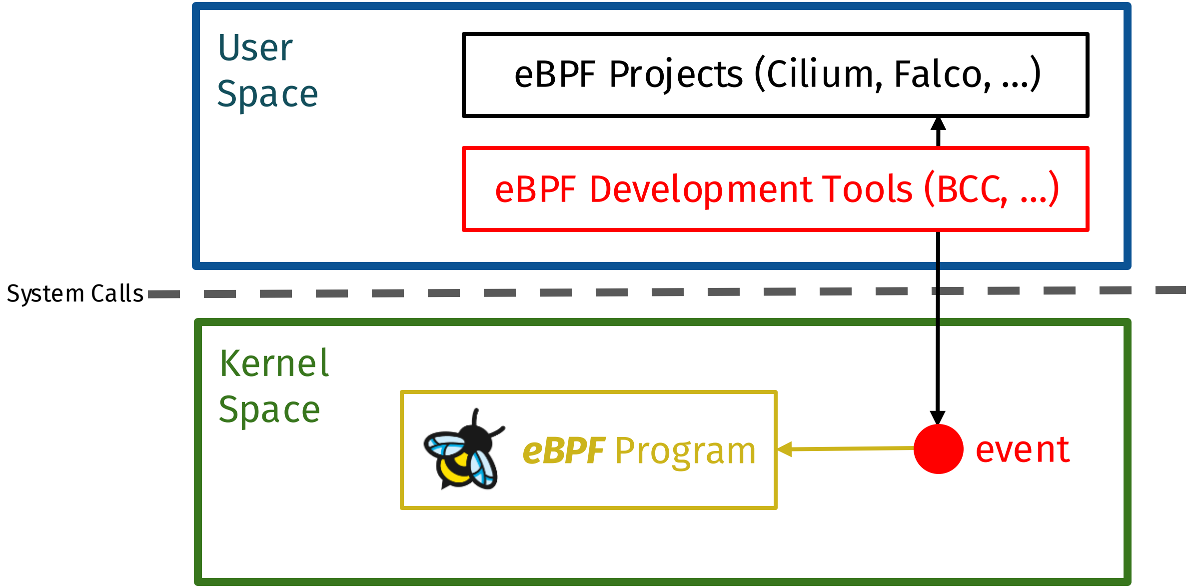
- https://tinyurl.com/vim-tut

eBPF Development

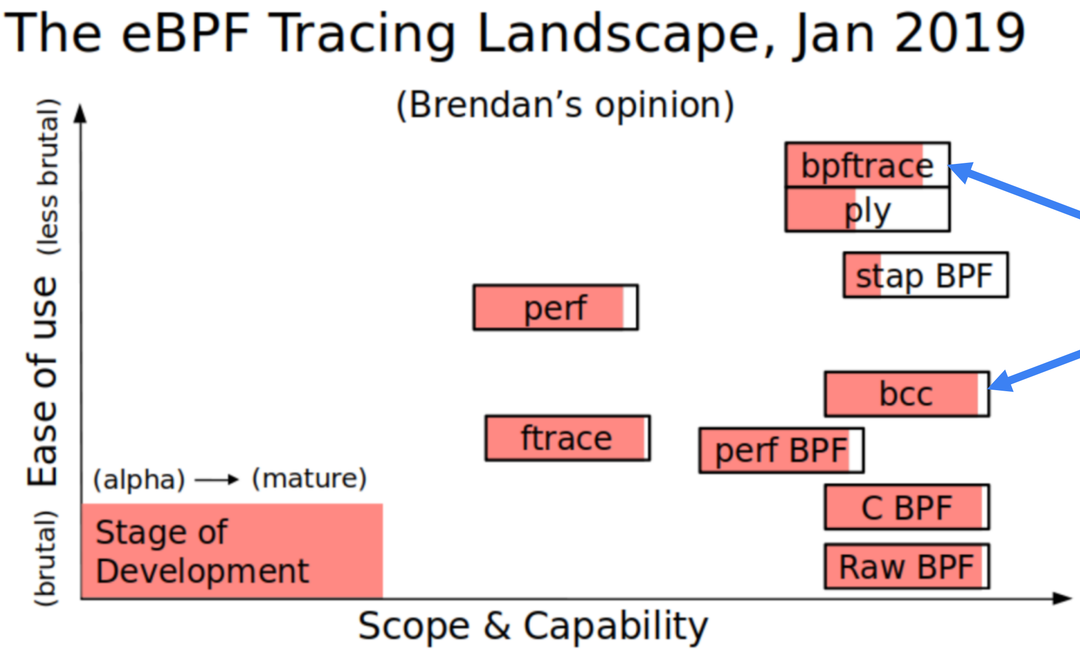
eBPF Development Tools



eBPF Development Tools



eBPF Front Ends

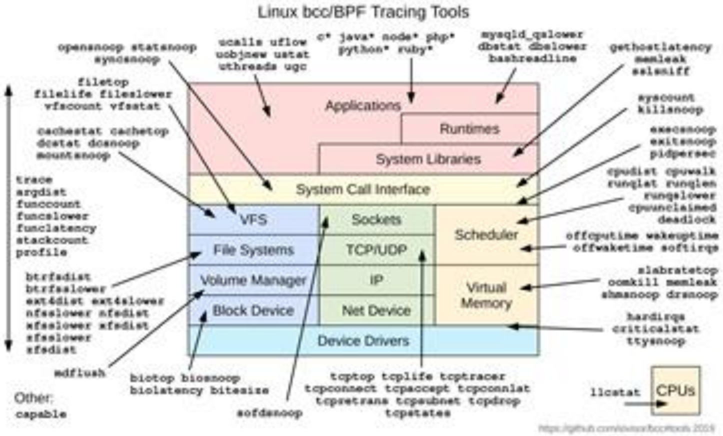


This tutorial will use theses front ends

Brendan Gregg, "**Linux Extended BPF (eBPF) Tracing Tools**", https://www.brendangregg.com/ebpf.html

eBPF Development Tools - ***BPF Compiler Collection (BCC)***

- Helps developers write, load, and attach eBPF programs to kernel events
- Biggest advantages
  - Write ***eBPF*** logic in ***C*** and ***control*** logic in ***Python*** or ***Lua***
  - ***BCC compiler*** translates embedded C code into eBPF bytecode.
  - ***Data*** collection and analysis can run it ***Python***.
- Many ready to use tracing tools

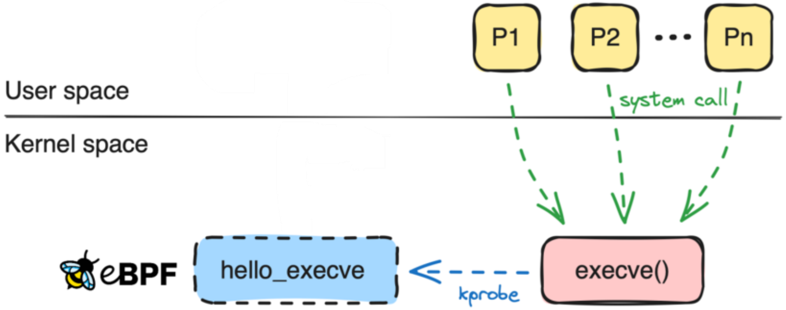


Source of picture: github.com/iovisor/bcc

Helloworld with ***BCC***

- Please navigate to **`{tutorial directory}/2-HelloWorld/BCC`** to experiment with BCC.

- ***Goal***: print from eBPF space anytime a Program uses ***execve*** system call.



- ***execve*** is responsible for executing a new program.

How to print from eBPF space? ***bpf_trace_printk***

- Please navigate to **`{tutorial directory}/2-HelloWorld/BCC`**
to experiment with BCC.

- ***bpf_trace_printk*** is a helper prints messages to the trace log of the kernel.

- bpf_trace_printk("message");

Helloworld with ***BCC***

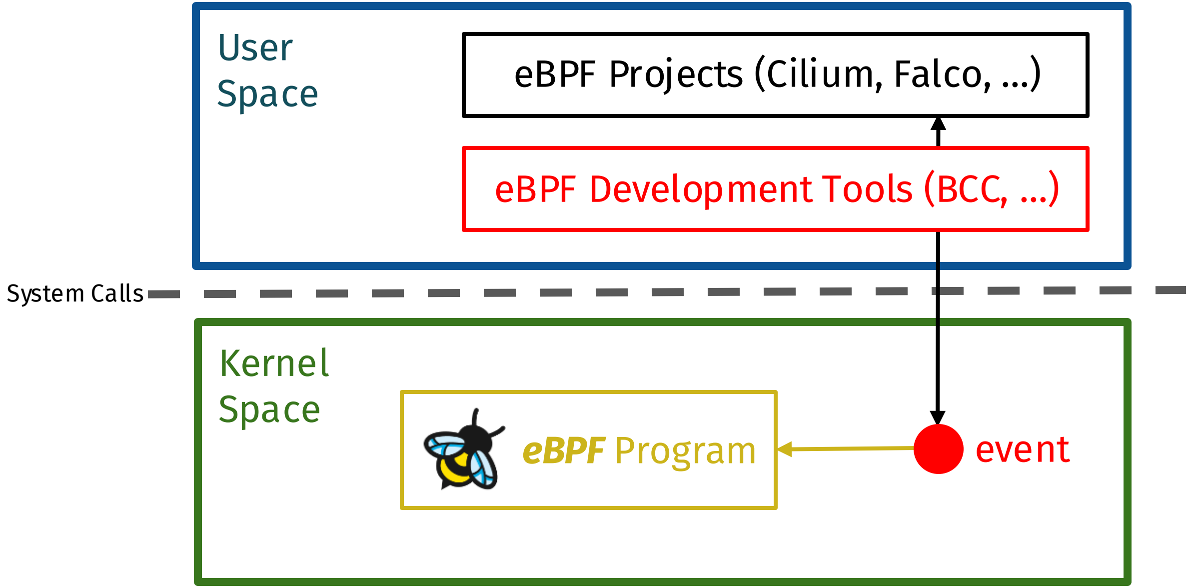
- Please navigate to **`{tutorial directory}/2-HelloWorld/BCC`** to experiment with BCC

- Please fill the hello function to print "Hello World!" in eBPF space.

`sudo python3 helloWorld.py`

```
b'           <...>-70101   [000] ....1 82841.999068: bpf_trace_printk: Hello World!'
b'           <...>-70103   [001] ....1 82844.558208: bpf_trace_printk: Hello World!'
b'           <...>-70105   [000] ....1 82853.858909: bpf_trace_printk: Hello World!'
```

eBPF Development Tools



eBPF Development Tools - ***bpftrace***

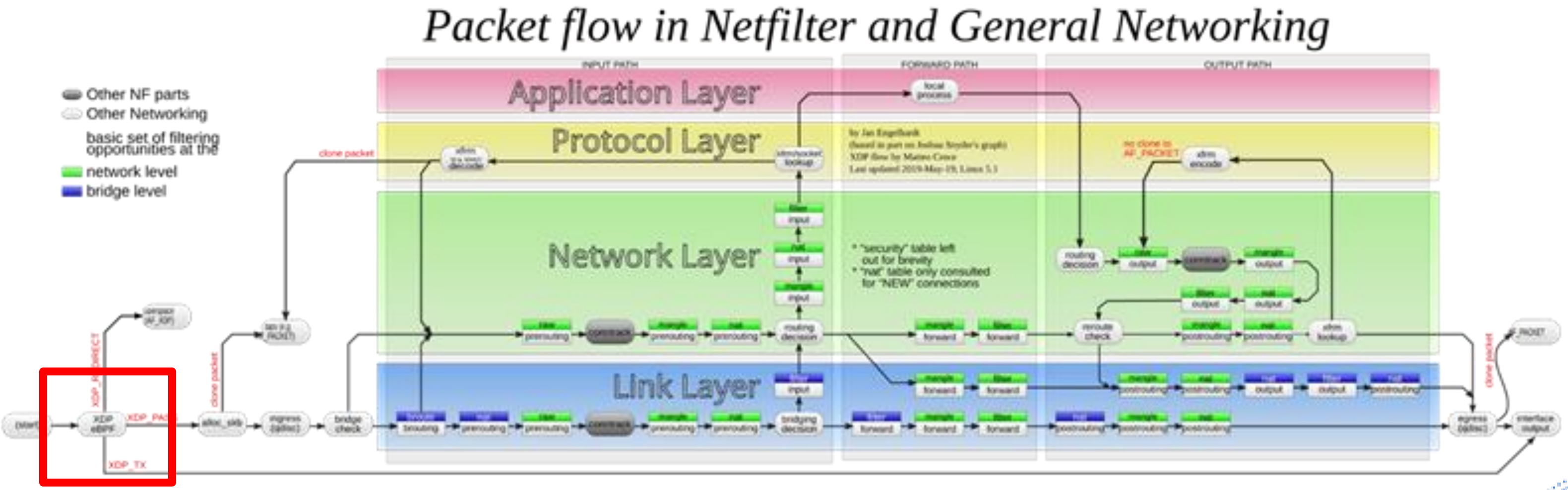
- ***bpftrace*** is a command-line tool that makes eBPF accessible.

- Biggest advantages
  - **No Setup Needed**: No need for wrappers in C & Python. Can even accept 1 liner programs
    - Syscall count by thread name:
    - `bpftrace -e 'tracepoint:raw_syscalls:sys_enter { @[comm] = count(); }'`
  - **Fast Debugging**: Perfect for on-the-fly performance analysis.

- Please navigate to **`{tutorial directory}/2-HelloWorld/bpftrace`** to see this tool in action

eBPF Development Tools - ***bpftrace***

- Please navigate to **`{tutorial directory}/2-HelloWorld/bpftrace`** to see this tool in action



eBPF Development Tools - *bpftrace*

- Please navigate to **`{tutorial directory}/2-HelloWorld/bpftrace`** to see this tool in action

- Please move forward until you reach ***Inspecting the Compiled eBPF Object File*** section

bpftrace - Inspecting the Compiled eBPF Object File

- The ***file*** utility is commonly used to determine the contents of a file.

```
file helloWorld.bpf.o
helloWorld.bpf.o: ELF 64-bit LSB relocatable, eBPF, version 1
(SYSV), with debug_info, not stripped
```

- This shows it's an ***ELF (Executable and Linkable Format)*** file, containing ***eBPF code***, for a ***64-bit platform*** with ***LSB (least significant bit) architecture***.

bpftrace - eBPF bytecode with llvm-objdump

`llvm-objdump-18 -S helloWorld.bpf.o`

```
helloWorld.bpf.o:       file format elf64-bpf
Disassembly of section xdp:
0000000000000000 <helloWorld>:
;     bpf_printk("Hello World %d", counter);
       0:       18 06 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r6 = 0x0 ll
       2:       61 63 00 00 00 00 00 00 r3 = *(u32 *)(r6 + 0x0)
       3:       18 01 00 00 00 00 00 00 00 00 00 00 00 00 00 00 r1 = 0x0 ll
       5:       b7 02 00 00 0f 00 00 00 r2 = 0xf
       6:       85 00 00 00 06 00 00 00 call 0x6
;     counter++;
       7:       61 61 00 00 00 00 00 00 r1 = *(u32 *)(r6 + 0x0)
       8:       07 01 00 00 01 00 00 00 r1 += 0x1
       9:       63 16 00 00 00 00 00 00 *(u32 *)(r6 + 0x0) = r1
;     return XDP_PASS;
      10:       b7 00 00 00 02 00 00 00 r0 = 0x2
      11:       95 00 00 00 00 00 00 00 exit
```

bpftrace - Loading the Program into the Kernel

`sudo bpftool prog list name helloWorld`

```
47: xdp  name helloWorld  tag d35b94b4c0c10efb  gpl
        loaded_at 2025-10-08T22:00:10+0000  uid 0
        xlated 96B  jited 68B  memlock 4096B  map_ids 6,7
        btf_id 52
```

bpftrace - Loading the Program into the Kernel

`sudo bpftool prog show id {ID} --pretty`


```
{
    "id": 47,
    "type": "xdp",
    "name": "helloWorld",
    "tag": "d35b94b4c0c10efb",
    "gpl_compatible": true,
    "loaded_at": 1759960810,
    "uid": 0,
    "orphaned": false,
    "bytes_xlated": 96,
    "jited": true,
    "bytes_jited": 68,
    "bytes_memlock": 4096,
    "map_ids": [6,7
    ],
    "btf_id": 52
}
```

bpftrace - Loading the Program into the Kernel

`sudo bpftool prog dump xlated name helloWorld`

```
int helloWorld(struct xdp_md * ctx):
; bpf_printk("Hello World %d", counter);
   0: (18) r6 = map[id:18][0]+0
   2: (61) r3 = *(u32 *)(r6 +0)
   3: (18) r1 = map[id:19][0]+0
   5: (b7) r2 = 15
   6: (85) call bpf_trace_printk#-118144
; counter++;
   7: (61) r1 = *(u32 *)(r6 +0)
   8: (07) r1 += 1
   9: (63) *(u32 *)(r6 +0) = r1
; return XDP_PASS;
   10: (b7) r0 = 2
   11: (95) exit
```

Introduction to Server-Client Workloads

NVIDIA Triton Inference Server and Client

- NVIDIA Triton Workload
- Setting up the server: Here
- Setting up the client: Here
- Triton exposes both HTTP/REST and GRPC endpoints
  - HTTP port: 8000
  - gRPC port: 8001

- gRPC
  - gRPC = Remote Procedure Call by Google
  - Open-source, high-performance RPC framework
  - Uses HTTP/2 as transport, supports bi-directional streaming
  - Data serialized with Protocol Buffers (Protobuf)
  - Widely used in microservices, ML serving (e.g., Triton Inference Server)

Check whether the workloads are running properly

- Run the following commands:
  - ***docker start triton-server***
  - ***docker start triton-client***

  These commands will start the containers that we have already set up for you

- Now check whether the containers are running properly
  - ***docker ps***

```
(bpfenv) ebpftutorial@iiswc-tutorial-main-instance-ubuntu24:~$ docker ps
CONTAINER ID   IMAGE                                   COMMAND                  CREATED          STATUS         PORTS     NAMES
8e7f45bcec7e   nvcr.io/nvidia/tritonserver:24.08-py3-sdk   "/opt/nvidia/nvidia_…"   19 hours ago   Up 17 hours             triton-client
74dc7b886d00   nvcr.io/nvidia/tritonserver:24.08-py3       "/opt/nvidia/nvidia_…"   19 hours ago   Up 19 hours             triton-server
(bpfenv) ebpftutorial@iiswc-tutorial-main-instance-ubuntu24:~$
```

Check whether the workloads are running properly

- Execute the triton-client docker container in a **separate terminal**
  - ***docker exec -it triton-client /bin/bash***
- Inside this folder, there are some simple client codes for both HTTP and gRPC
  - ***/workspace/client/src/python/examples***
- We have also put two simple codes for the starter
  - http_test.py ---> HTTP client code
  - grpc_test.py ---> gRPC client code

Sending simple requests from client to server

- From the client docker container, we can send requests to the server
- This time we have the triton server and client both on the same machine, but we can have them on different machines and test
- To test whether the workloads are functioning, run:
  - ***python3 <client-python-filename> <duration-in-seconds>***

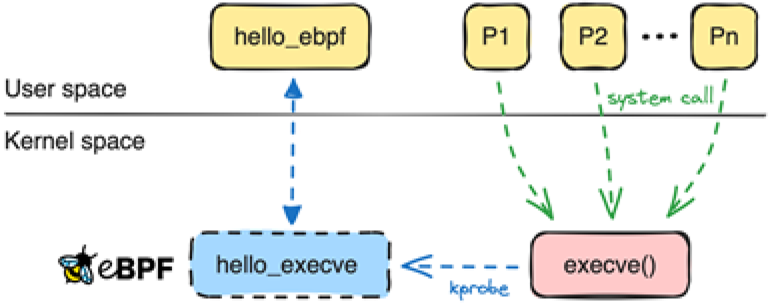
```
root@iiswc-tutorial-main-instance-ubuntu24:/workspace/client/src/python/examples# python3 grpc_test.py 10
Connected to Triton server at localhost:8001
Running for 10 seconds at 1 requests/second...
Sending request...
Response received successfully.
Sending request...
Response received successfully.
Sending request...
Response received successfully.
Sending request...
Response received successfully.
Sending request...
Response received successfully.
```

Tracing System Calls with eBPF kprobes

What is a kprobe?

- Dynamically trace kernel functions at runtime
- Attach the eBPF program to the system call using kprobe
- eBPF program runs whenever the particular syscall is called

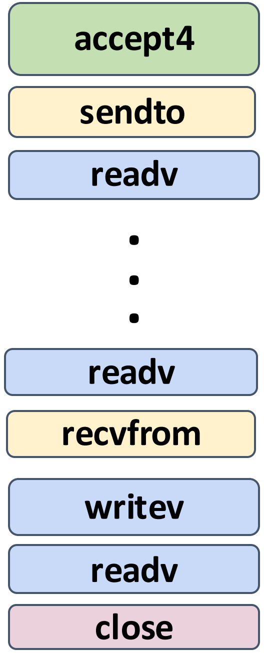
    - kprobe: before syscall execution
    - kretprobe: after syscall execution



https://edgedelta.com/company/blog/applied-introduction-ebpf-go

What we are trying to do

- What are the syscalls we want to trace?
    1. recvfrom
    2. sendto
- Container specific filtering
- Collect:
    a. PID
    b. Timestamp
    c. Process name (comm)
    d. File Descriptor(fd)
    e. Syscall name



Syscalls used by triton http server

Let's have a look at the syscall synopsis

- Before starting to write the code for tracing these syscalls, we need to have a look at their synopses

recvfrom(2) - Linux man page

Name

recv, recvfrom, recvmsg - receive a message from a socket

Synopsis

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t recv(int sockfd, void *buf, size_t len, int flags);

ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags,
                 struct sockaddr *src_addr, socklen_t *addrlen);

ssize_t recvmsg(int sockfd, struct msghdr *msg, int flags);
```

[recvfrom]

sendto(2) - Linux man page

Name

send, sendto, sendmsg - send a message on a socket

Synopsis

```c
#include <sys/types.h>
#include <sys/socket.h>

ssize_t send(int sockfd, const void *buf, size_t len, int flags);

ssize_t sendto(int sockfd, const void *buf, size_t len, int flags,
               const struct sockaddr *dest_addr, socklen_t addrlen);

ssize_t sendmsg(int sockfd, const struct msghdr *msg, int flags);
```

[sendto]

Writing a kprobe

Please navigate to ***{tutorial directory}/4-kprobe/*** to experiment with kprobe

Step 1: Getting the container PID

This function returns the server container PID, so that we can only probe the syscalls made by the container

```
from bcc import BPF
import os

def get_pid(container_name):
        cmd = f"docker inspect --format '{{{{.State.Pid}}}}' {container_name}"
        return int(os.popen(cmd).read().strip())
```

Writing a kprobe

Step 2: Define the eBPF program to attach to the syscall

This function name must follow **syscall__<syscall-name>** convention

```
int syscall__recvfrom(struct pt_regs *ctx, int fd, void *buf, size_t len, int flags, struct sockaddr *src_addr, __u32 *addrlen)
{
        u32 pid = bpf_get_current_pid_tgid() >> 32;
        if (pid != TRITON_PID) return 0;

        char comm[TASK_COMM_LEN];
        bpf_get_current_comm(&comm, sizeof(comm));
        bpf_trace_printk("recvfrom pid=%d fd=%d comm=%s\n", pid, fd, comm);

        return 0;
}
```

Must be same as the signature

This line collects the process id

Retrieves the name of the current process

Prints the syscall info

Writing a kprobe

Step 3: Sending the triton PID to the eBPF code

```
bpf_text = r"""
................
//bpf code goes here
................
""".replace("TRITON_PID", str(TRITON_PID))
```

Now the TRITON_PID variable will be accessible to the eBPF code

Step 4: Load the eBPF code and attach kprobes

```
b = BPF(text=bpf_text, cflags=["-w"])
```

Loads the eBPF code

```
b.attach_kprobe(event="__x64_sys_recvfrom", fn_name="syscall__recvfrom")
```

Attaches the syscall specific function to the syscall

Writing the kprobe

Step 5: Run the kprobe code

sudo python3 <filename> -c <container-name>
Which is in our case:
sudo python3 kprobe.py -c triton-server

On a different terminal, execute the triton-client docker container and send some http requests

Step 6: Example Output

```
(bpfenv) ebpftutorial@iiswc-tutorial-main-instance-ubuntu24:~/eBPF-tutorial/4-kprobes$ sudo python3 kprobe.py -c triton-server
Tracing Triton (PID 1447) inside container triton-server
Printing kprobe messages (Ctrl+C to stop)...
b'    tritonserver-1593    [000] ....1 202459.521438: bpf_trace_printk: sendto pid=1447 fd=10 comm=tritonserver'
b''
b'    tritonserver-1585    [001] ....1 202459.521519: bpf_trace_printk: recvfrom pid=1447 fd=9 comm=tritonserver'
b''
b'    tritonserver-1570    [000] ....1 202459.643989: bpf_trace_printk: sendto pid=1447 fd=10 comm=tritonserver'
b''
b'    tritonserver-1585    [000] ....1 202459.644114: bpf_trace_printk: recvfrom pid=1447 fd=9 comm=tritonserver'
b''
b'    tritonserver-1593    [000] ....1 202460.520534: bpf_trace_printk: sendto pid=1447 fd=10 comm=tritonserver'
b''
b'    tritonserver-1585    [000] ....1 202460.520599: bpf_trace_printk: recvfrom pid=1447 fd=9 comm=tritonserver'
```

Tracing User Space Functions with eBPF uprobes

What we are trying to do

To write a uprobe:

- Find the tritonserver binary
- Pick some functions of interest from the source code of c grpc here to uprobe
- Find the symbols for the functions of interest
- Container specific filtering

Please navigate to ***{tutorial directory}/5-uprobe/*** to experiment with uprobe

Writing a uprobe

Step 1: To attach uprobes, you need the absolute path of the triton server binary

```
docker info | grep "Docker Root Dir"
# Example: /var/lib/docker
```

Find exactly where docker is installed

```
find /var/lib/docker/overlay2 -type f -name "tritonserver"

# Example result:
/var/lib/docker/overlay2/6f3e90...dd0f/diff/opt/tritonserver/bin/tritonserver
```

Docker stores all the binary files inside the "overlay2" folder

This is the binary file path

Note:

There might be more than one binary file path . The *merged/* path may disappear or be remounted. If the container restarts or stops:

- The *merged/* mount is unmounted automatically.
- The *diff/* directory remains on disk.

Writing a uprobe

Step 2: Select the function that we want to probe

For example, we want to trace the following function from the gRPC source code

- grpc_chttp2_maybe_complete_recv_trailing_metadata

Look at the function signature

Writing a uprobe

Step 3: Look for the function symbol in our target binary

sudo objdump -t <tritonserver-binary-path> | grep <function-name>

```
(bpfenv) ebpftutorial@iiswc-tutorial-main-instance-ubuntu24:~$ sudo objdump -t /var/lib/docker/overlay2/b49f62eb49d20d5cd2c849cd4b81e4777bb042a4924376786e74cc113f8b210f/diff/opt/tritonserver/
bin/tritonserver | grep grpc_chttp2_maybe_complete_recv_trailing_metadata
00000000000deaa9 l     F .text  0000000000000018              _Z49grpc_chttp2_maybe_complete_recv_trailing_metadataP21grpc_chttp2_transportP18grpc_chttp2_stream.cold
00000000003d3800 g     F .text  00000000000001fc              _Z49grpc_chttp2_maybe_complete_recv_trailing_metadataP21grpc_chttp2_transportP18grpc_chttp2_stream
```

You can run:

echo <symbol> | c++filt This demangles the symbol so that we can match it to the function signature

For this case:

echo _Z49grpc_chttp2_maybe_complete_recv_trailing_metadataP21grpc_chttp2_transportP18grpc_chttp2_stream | c++filt

This is the function symbol we need

Output:

grpc_chttp2_maybe_complete_recv_trailing_metadata(grpc_chttp2_transport*, grpc_chttp2_stream*)

Writing the uprobe

Step 4: Define the eBPF program to attach to the function

This function gets triggered whenever the targeted function gets called

```
int trace_metadata_func(struct pt_regs *ctx) {
    u32 pid = bpf_get_current_pid_tgid() >> 32;
    if (pid != PID) return 0;

    bpf_trace_printk("gRPC Metadata Function Called - PID: %d\n", pid);
    return 0;
}
```

Step 5: Attach uprobe

```
def attach_uprobe(bpf, triton_binary):
    func_symbol="_Z49grpc_chttp2_maybe_complete_recv_trailing_metadataP21grpc_chttp2_transportP18grpc_chttp2_stream"
    bpf.attach_uprobe(name=triton_binary, sym=func_symbol, fn_name="trace_metadata_func")
```

For uprobe attachment, the appropriate binary path and function symbol is crucial

Writing the uprobe

Step 6: Run the uprobe code

sudo python3 <filename> -c <container-name>

For our case:
sudo python3 uprobe.py -c triton-server

On a different terminal, execute the triton-client docker container and send some gRPC requests

Step 7: Example Output

```
(bpfenv) ebpftutorial@iiswc-tutorial-main-instance-ubuntu24:~/eBPF-tutorial/5-uprobes$ sudo python3 uprobe.py -c triton-server
Initializing tracing for container 'triton-server' (PID 1447)
Tracing gRPC function calls... Press Ctrl+C to stop.
b'    tritonserver-1584    [001] ....1 243829.110709: bpf_trace_printk: gRPC Metadata Function Called - 1447: 1447'
b''
b'    tritonserver-1582    [000] ....1 243829.123963: bpf_trace_printk: gRPC Metadata Function Called - 1447: 1447'
b''
b'    tritonserver-1570    [001] ....1 243829.393247: bpf_trace_printk: gRPC Metadata Function Called - 1447: 1447'
b''
b'    tritonserver-1582    [000] ....1 243830.087826: bpf_trace_printk: gRPC Metadata Function Called - 1447: 1447'
b''
b'    tritonserver-1582    [001] ....1 243830.090625: bpf_trace_printk: gRPC Metadata Function Called - 1447: 1447'
b''
```

Characterizing In-Kernel Observability of Latency-Sensitive Request-level Metrics with eBPF

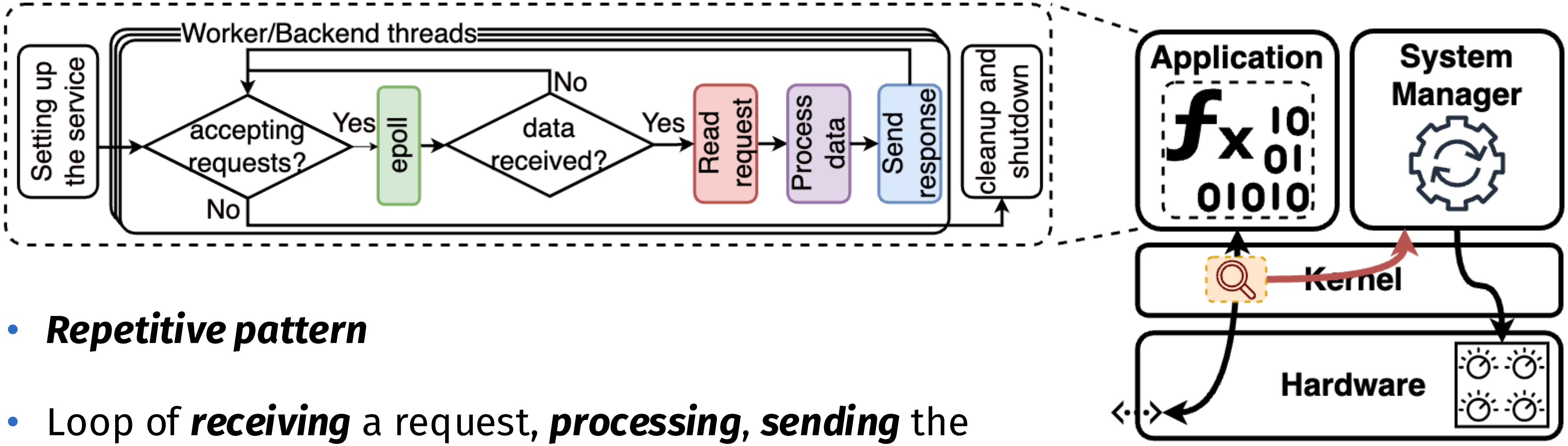
Mohammadreza Rezvani, Ali Jahanshahi,
Daniel Wong

Systems Optimization + Computer Architecture Lab

ISPASS'24 – Best Paper Nominee

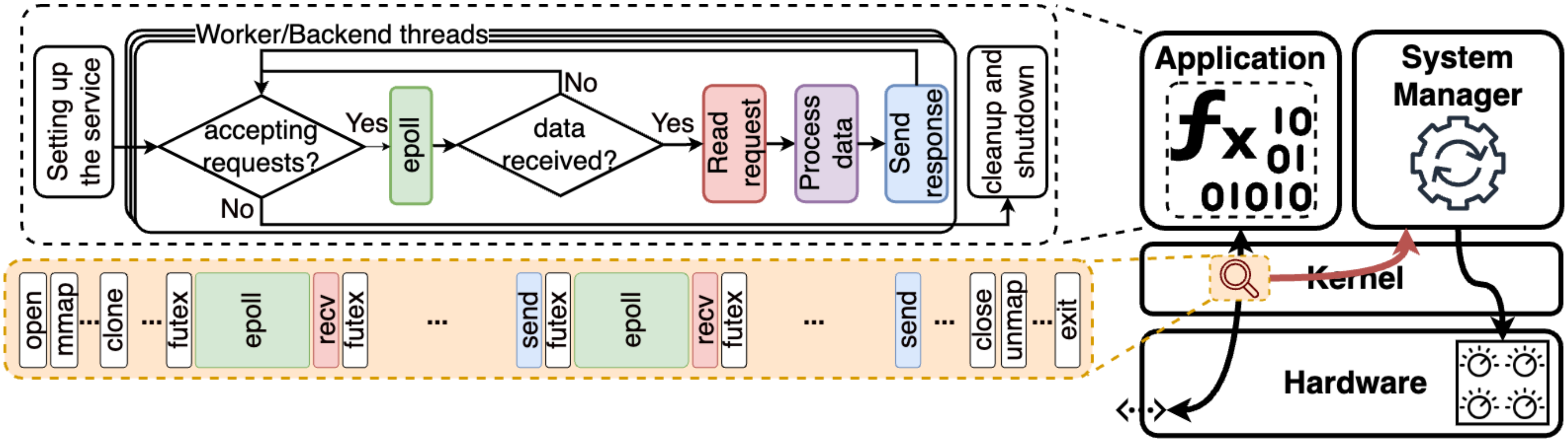
UC RIVERSIDE

Goal: Characterizing In-Kernel Observability With eBPF

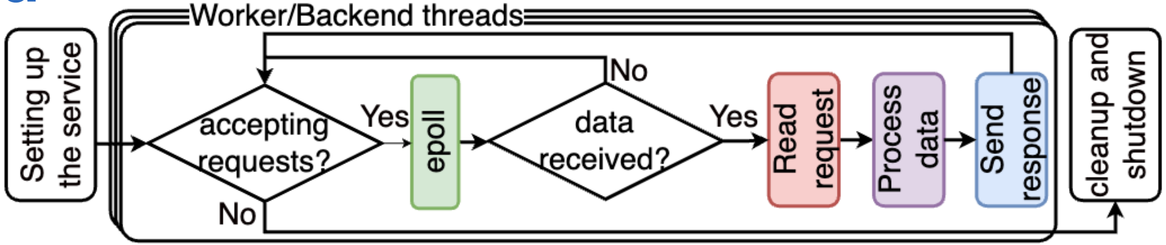


- ***Repetitive pattern***
- Loop of ***receiving*** a request, ***processing***, ***sending*** the result and ***waiting***.
- Except processing, handling a process requires ***interaction*** with kernel that we can ***observe***.

Interaction With Kernel Is Done Through System Calls



Q1: Which System Calls Provide Observability into workload



Network

- **Receiving request**
  - read, recv, recvfrom, recvmsg, …
- **Sending response**
  - write, send, sendto, sendmsg, …

Processing/Waiting

- **Processing**
  - Doesn’t require interaction with kernel
- **Waiting**
  - epoll, epoll_wait, select

Q2: What application-level metrics are observable through system call activities?

- Observing Throughput by estimating **Request Per Second (RPS)**
- Observing Latency by **Observing Saturation** and estimating **Saturation Slack**

Evaluation Methodology

Benchmarks

- **TailBench** [1]
  - Img_dnn, xapian, silo, specjbb, moses
- **CloudSuite** [2]
  - Data Caching and Web search
- **Triton** [3]
  - HTTP Protocol and GRPC protocol

System Specification

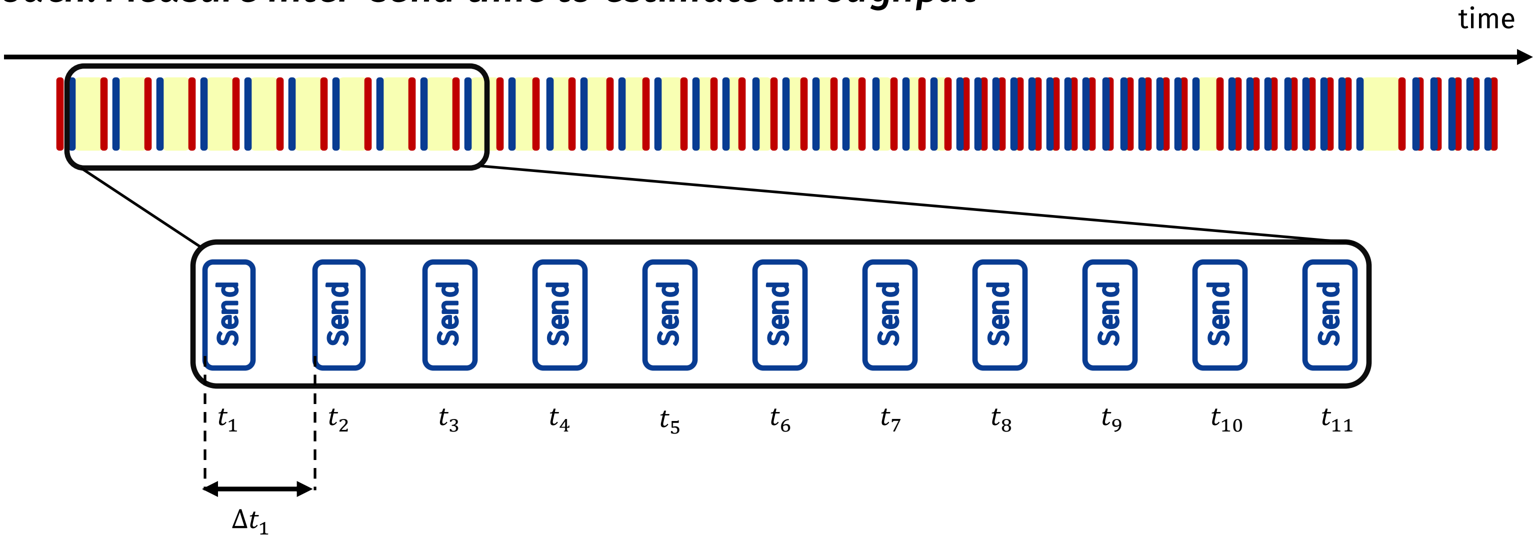
- **AMD based server**
  - AMD EPYC 7302
  - Ubuntu 20.04.1
- **Intel based server**
  - Intel Xeon CPU E5-2620
  - Red Hat 4.8.5-36

[1] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," 2016 IEEE International Symposium on Workload Characterization (IISWC), Providence, RI, USA, 2016, pp. 1-10, doi: 10.1109/IISWC.2016.7581261.
[2] Palit, T., Shen, Y. and Ferdman, M., 2016, April. Demystifying cloud benchmarking. In 2016 IEEE international symposium on performance analysis of systems and software (ISPASS) (pp. 122-132). IEEE.
[3] "NVIDIA triton inference server," https://developer.nvidia.com/nvidiatriton-inference-server, Mar. 2020, accessed: 2022-11-16.

How to observe RPS?

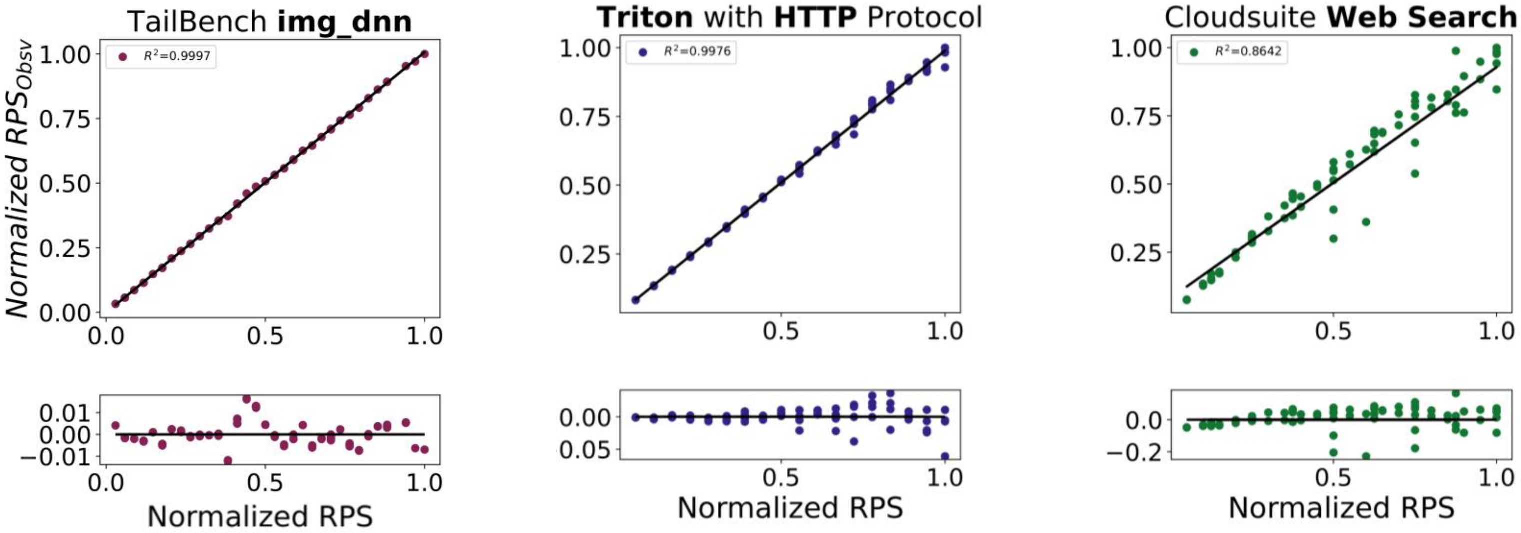
Intuition: Every response through socket will use a send system call

Approach: Measure inter-send time to estimate throughput



$$RPS_{Obsv} = \frac{r}{t_r^{send} - t_1^{send}} = \frac{1}{\overline{\Delta t^{send}}}$$

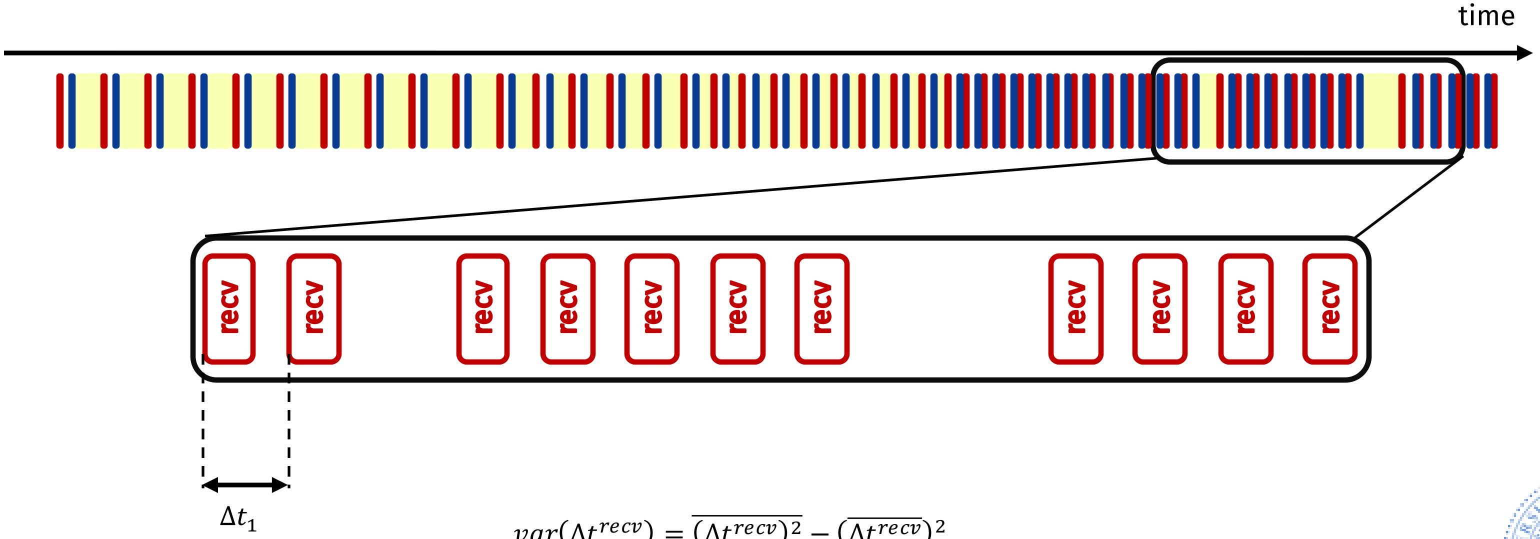
Observed RPS Correlates Well With Actual RPS



How Do We Observe System Saturation?

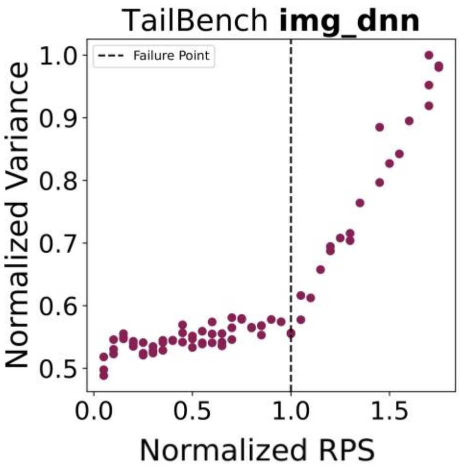
Intuition: Under saturation, system experience longer than usual delays that are noticeable in system call timings

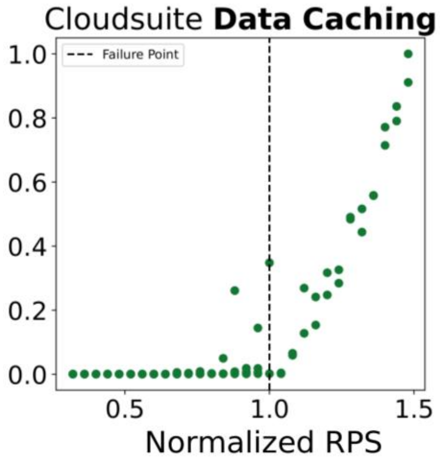
Approach: Measure variance of inter-receive time.

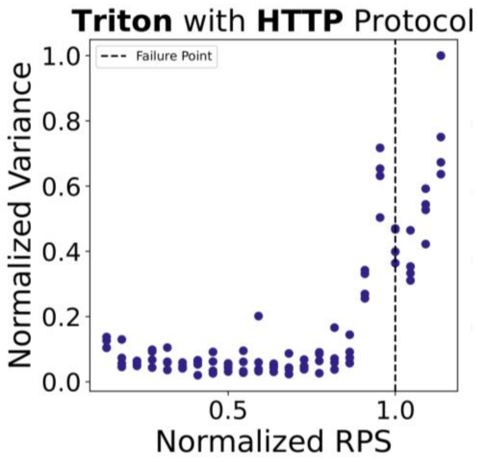


$$var(\Delta t^{recv}) = \overline{(\Delta t^{recv})^2} - (\overline{\Delta t^{recv}})^2$$

Variance Increase During Saturation



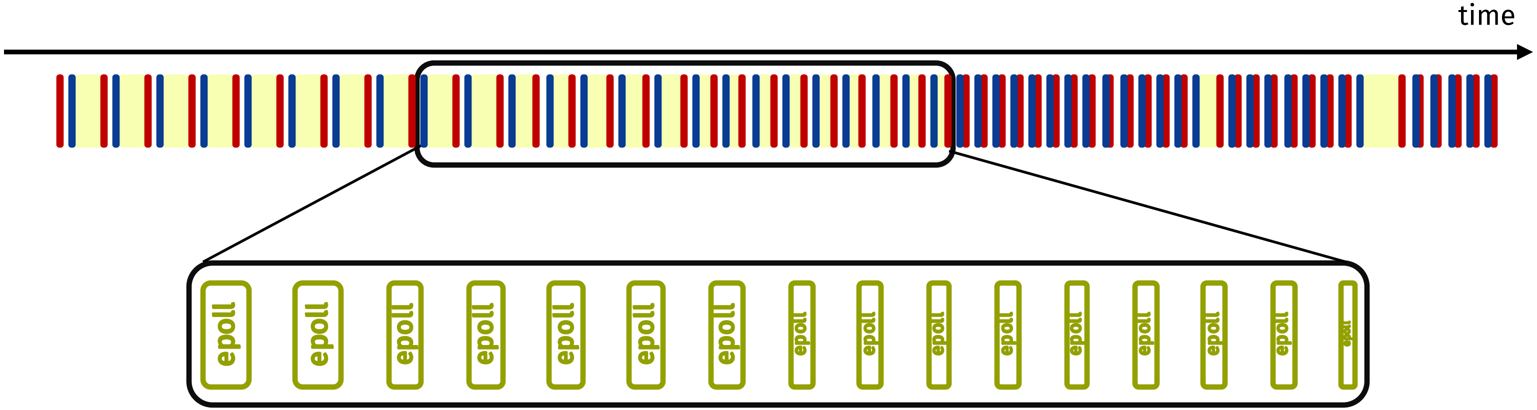




How Do We Observe saturation slack?

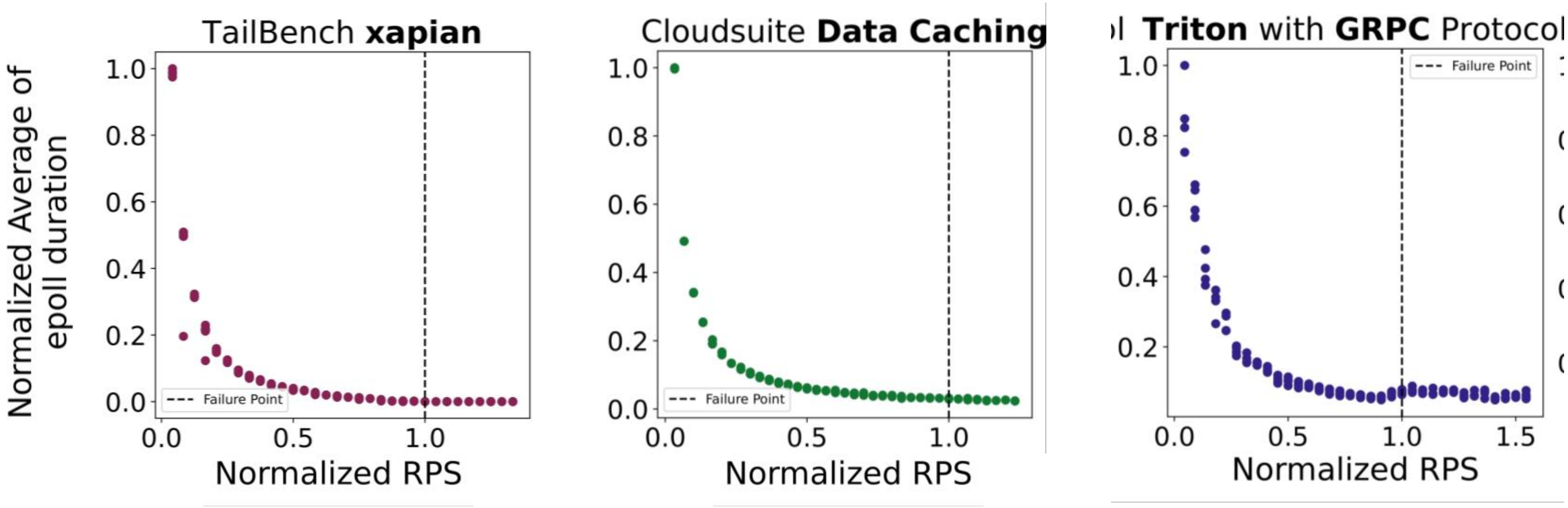
Intuition: Wait time captured by duration of epoll, decrease as load increases

Approach: Duration of epoll as proxy for slack



$$avg(duration(epoll))$$

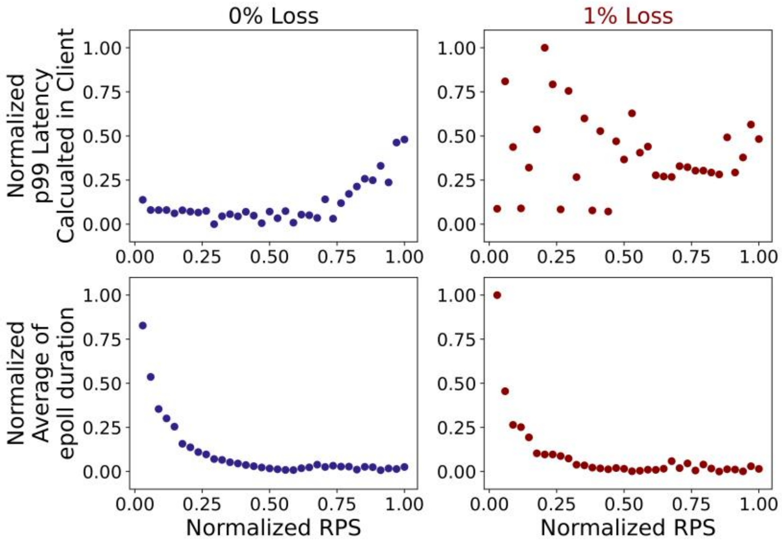
Saturation Slack Observability



How Robust is eBPF-based observability?

Network

Fluctuations of network does not affect the metrics



Network config for $RPS_{Obsv}(R^2)$	0ms Delay 0% Loss	10ms Delay 1% Loss
TailBench **img-dnn**	0.9997	0.9998
TailBench **xapian**	0.9976	0.9964
TailBench **silo**	0.9998	0.9986
TailBench **specjbb**	0.9997	0.9996
TailBench **moses**	0.9411	0.9435
CloudSuite **Data Caching**	0.9995	0.9989
CloudSuite **Web Search**	0.8642	0.8573
Trtion w/ **HTTP** Protocol	0.9976	0.9981
Trtion w/ **GRPC** Protocol	0.9711	0.9703

Use cases

01 Resource management

eBPF kernel tracing offers a non-invasive way to understand and optimize resource demands and latency drivers

02 low overhead monitoring

All our metrics can be calculated in eBPF space therefore it has super low overhead

03 Predictive Provisioning

Studying the information left in kernel will open doors for predicting application behavior.

Summary

- **non-invasive tools for observability have become increasingly important for data center management and optimization**

- **Characterizing system call activity can open up opportunities to increase observability.**

- **eBPF can provide robust observability to various set of metrics.**

- **System call activities are correlated to throughput and latency based metrics**

- **Use Cases**
  - Resource management
  - low overhead monitoring
  - Predictive Provisioning

Thank you!

Questions?