PythonBPF - Writing eBPF programs in pure Python

FOSDEM 2026

Pragyansh Chaturvedi, Varun Mallya

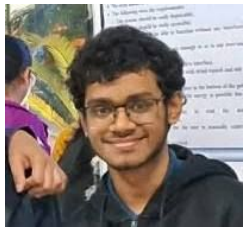
31 January 2026

PythonBPF

Introduction

- Pragyansh Chaturvedi <r41k0u@ubuntu.com>
  - Part of Ubuntu Engineering at Canonical
  - Co-maintainer of PythonBPF

- Varun Mallya <varunrmallya@gmail.com>
  - Engineering student
  - Co-maintainer of PythonBPF



PythonBPF

Introduction

- PythonBPF is a pure Python frontend for writing eBPF programs
- This sounds similar to what you can do with BCC, but is significantly different

PythonBPF

What PythonBPF does differently

- Allows a reduced Python grammar for eBPF specific code
- Abstract away typing and verifier-massaging complexities (as best as it can)

PythonBPF

PythonBPF and BCC - side by side

```
# define BPF program
prog = """
#include <linux/sched.h>

// define output data structure in C
struct data_t {
    u32 pid;
    u64 ts;
    char comm[TASK_COMM_LEN];
};
BPF_PERF_OUTPUT(events);

int hello(struct pt_regs *ctx) {
    struct data_t data = {};

    data.pid = bpf_get_current_pid_tgid();
    data.ts = bpf_ktime_get_ns();
    bpf_get_current_comm(&data.comm, sizeof(data.comm));

    events.perf_submit(ctx, &data, sizeof(data));

    return 0;
}
"""
```

```
@bpf
@struct
class data_t:
    pid: c_int64
    ts: c_int64
    comm: str(16)  # type: ignore [valid-type]


@bpf
@map
def events() -> PerfEventArray:
    return PerfEventArray(key_size=c_int64, value_size=c_int64)


@bpf
@section("tracepoint/syscalls/sys_enter_clone")
def hello(ctx: c_void_p) -> c_int64:
    dataobj = data_t()
    dataobj.pid, dataobj.ts = pid(), ktime()
    comm(dataobj.comm)
    events.output(dataobj)
    return 0  # type: ignore [return-value]
```

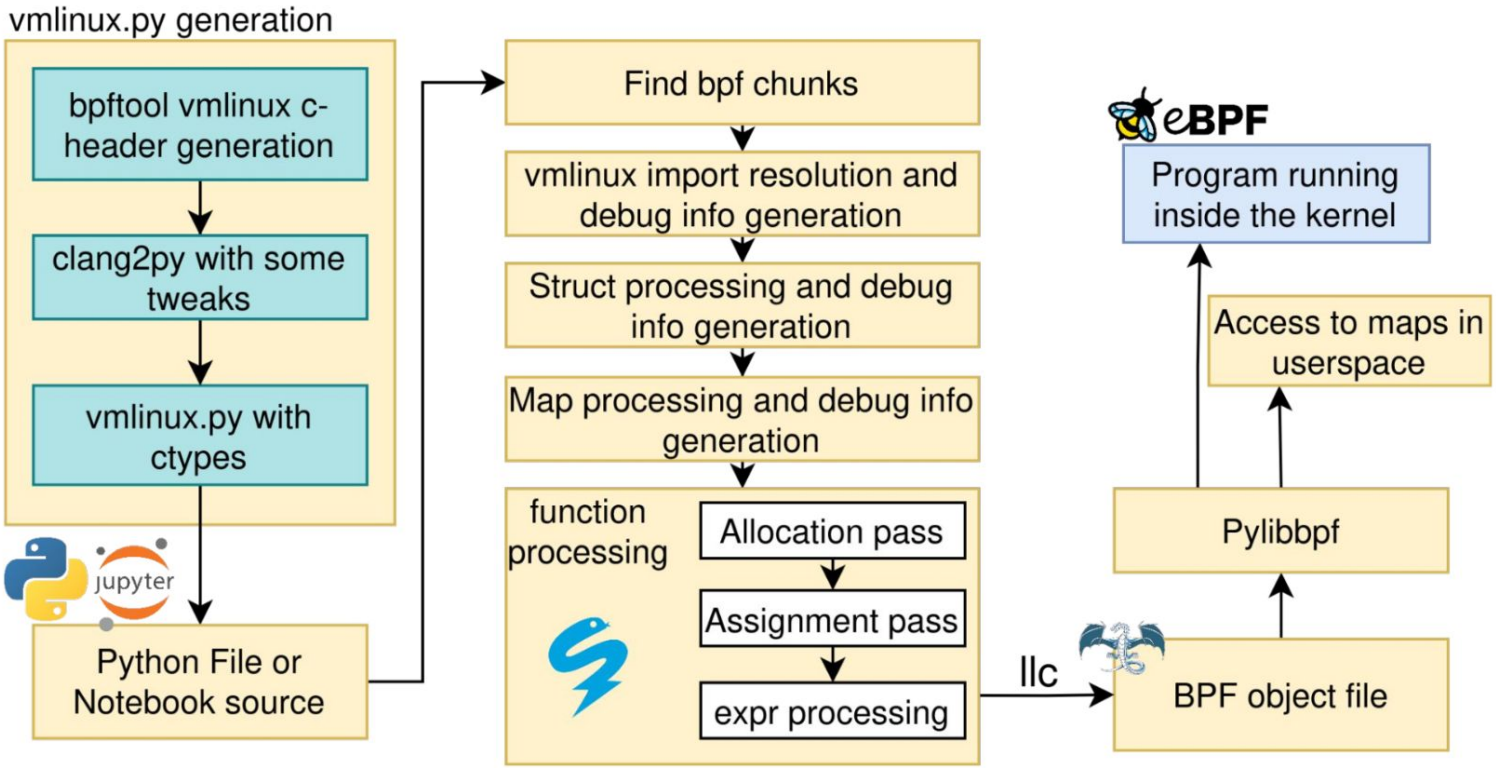
ythonBPF

Basic overview of a PythonBPF program

```python
from pythonbpf import bpf, map, struct, section, bpfglobal, BPF
from pythonbpf.helper import ktime, pid, comm
from pythonbpf.maps import PerfEventArray
from ctypes import c_void_p, c_int64


@bpf
@struct
class data_t:
    pid: c_int64
    ts: c_int64
    comm: str(16)  # type: ignore [valid-type]


@bpf
@map
def events() -> PerfEventArray:
    return PerfEventArray(key_size=c_int64, value_size=c_int64)


@bpf
@section("tracepoint/syscalls/sys_enter_clone")
def hello(ctx: c_void_p) -> c_int64:
    dataobj = data_t()
    dataobj.pid, dataobj.ts = pid(), ktime()
    comm(dataobj.comm)
    events.output(dataobj)
    return 0  # type: ignore [return-value]
```

PythonBPF

Compilation flow



PythonBPF

A walkthrough - disksnoop

```
from ctypes import c_int32, c_int64, c_uint64

from vmlinux import struct_pt_regs, struct_request

from pythonbpf import bpf, bpfglobal, compile, map, section
from pythonbpf.helper import ktime
from pythonbpf.maps import HashMap


@bpf
@map
def start() -> HashMap:
    return HashMap(key=c_uint64, value=c_uint64, max_entries=10240)
```

PythonBPF

A walkthrough - disksnoop

```
@bpf
@section("kprobe/blk_mq_end_request")
def trace_completion(ctx: struct_pt_regs) -> c_int64:
    req_ptr = ctx.di
    req = struct_request(ctx.di)
    data_len = req.__data_len
    cmd_flags = req.cmd_flags
    req_tsp = start.lookup(req_ptr)
    if req_tsp:
        delta = ktime() - req_tsp
        delta_us = delta // 1000
        print(f"{data_len} {cmd_flags:x} {delta_us}\n")
        start.delete(req_ptr)

    return 0  # type: ignore [return-value]
```

PythonBPF

A walkthrough - disksnoop

```
@bpf
@section("kprobe/blk_mq_start_request")
def trace_start(ctx1: struct_pt_regs) -> c_int32:
    req = ctx1.di
    ts = ktime()
    start.update(req, ts)
    return 0  # type: ignore [return-value]


@bpf
@bpfglobal
def LICENSE() -> str:
    return "GPL"


compile()
```

PythonBPF

Demos and Examples

- These are the video demos: https://youtu.be/eFVhLnWFxtE?t=352 (Highly recommend!)
  - An interactive TUI based container monitor
  - Syscall anomaly detection for the Spotify snap
  - Kernel stack symbolization using blazesym
  - BCC's vfsreadlat example - ported to ipynb, live TUI and interactive live web dashboard
- You can try and tweak the examples yourself at https://github.com/pythonbpf/Python-BPF#try-it-out

PythonBPF

The internals

- As seen in previous examples, the @bpf decorator is what signals the compiler that the enclosed code is to be compiled as BPF code. We call them BPF chunks
- Then there are specifiers for structs, maps, and global strings.

PythonBPF

The internals - type deduction

- Due to the weakly typed nature of Python, PythonBPF has to employ type deduction for reliably lowering it to LLVM IR, while hiding this from the user. We need to see if a data container can be converted to a desired type, and do a best effort attempt to do it.
- This also introduces complexity in the local variable allocations, where we use a strategy of calculating the maximum allocations we might need, and the maximum scratch space (in terms of stack frames) we need for each type.
- We constantly test and improve our type deduction as this is an area most of our papercuts come from.

PythonBPF

The internals - type deduction

```c
/* completion: compute latency and print data_len, cmd_flags, latency_us */
SEC("kprobe/blk_mq_end_request")
int trace_completion(struct pt_regs *ctx)
{
    __u64 reqp = (__u64)(ctx->di);
    __u64 *tsp;
    __u64 now_ns;
    __u64 delta_ns;
    __u64 delta_us = 0;
    bpf_printk("%lld", reqp);
    tsp = bpf_map_lookup_elem(&start_map, &reqp);
    if (!tsp)
        return 0;

    now_ns = bpf_ktime_get_ns();
    delta_ns = now_ns - *tsp;
    delta_us = delta_ns / 1000;

    /* read request fields using CO-RE; needs vmlinux.h/BTF */
    __u32 data_len = 0;
    __u32 cmd_flags = 0;

    /* __data_len is usually a 32/64-bit; use CORE read to be safe */
    data_len = ( __u32 ) BPF_CORE_READ((struct request *)reqp, __data_len);
    cmd_flags = ( __u32 ) BPF_CORE_READ((struct request *)reqp, cmd_flags);

    /* print: "<bytes> <flags_hex> <latency_us>" */
    bpf_printk("%u %x %llu\n", data_len, cmd_flags, delta_us);

    /* remove from map */
    bpf_map_delete_elem(&start_map, &reqp);

    return 0;
```

```python
@bpf
@section("kprobe/blk_mq_end_request")
def trace_completion(ctx: struct_pt_regs) -> c_int64:
    req_ptr = ctx.di
    req = struct_request(ctx.di)
    data_len = req.__data_len
    cmd_flags = req.cmd_flags
    req_tsp = start.lookup(req_ptr)
    if req_tsp:
        delta = ktime() - req_tsp
        delta_us = delta // 1000
        print(f"{data_len} {cmd_flags:x} {delta_us}\n")
        start.delete(req_ptr)

    return 0  # type: ignore [return-value]
```

PythonBPF

The internals - vmlinux.py

- We write BTF info into `vmlinux.h` using `bpftool`, then run that through clang2py (with some tweaks) to get `vmlinux.py` with the kernel headers Pythonized.
- Internally, there is a handler that checks which structs and enums from vmlinux.py are being used in the program being compiled and resolves all the dependencies required to interface with that struct and generates IR for it as well.
- Once we get the headers and parse them into the required data structures, we generate Debug Info and add it to the IR.
- We then inject the parsed data into the symbol table and add handlers to the required locations where we expect vmlinux elements to be used.

PythonBPF

Where PythonBPF shines

- Users need to be less worried about typing
- Users are shielded from common verifier massaging
- A concise, Pythonic syntax, which allows users to use Python devtools in their IDE
- Allows using the Python ecosystem – allows for better data analysis and visualization
- Ideal for beginners and quick prototyping

PythonBPF

Next challenges

- PythonBPF is not feature complete (yet)
  - For it to be a serious industry choice, it has to support all the language constructs and all maps, helpers and kfuncs
  - One suggestion was to port the kernel selftests – but that seems to be a huge undertaking
- Establishing a feedback loop with our early adopters
  - Users whose needs and use cases are very clearly defined, and can give constant feedback
  - Grad students?

PythonBPF

Our Goals

- Make PythonBPF a prime choice for users to write BPF code
- Due to the choice of language, make BPF accessible to a much wider audience

PythonBPF

Summary

- PythonBPF is a reduced Python grammar for writing BPF programs in pure Python
- There is active development ongoing to increase coverage of supported BPF features and hardening the type deduction.
- PythonBPF and pylibbpf: https://github.com/pythonbpf/Python-BPF and https://github.com/pythonbpf/pylibbpf
- Project docs: https://python-bpf.readthedocs.io/en/latest
- Video Demos: https://youtu.be/eFVhLnWFxtE?t=352
- Try it out: https://github.com/pythonbpf/Python-BPF#try-it-out
- We welcome contributions, reviews, bug reports and RFCs!

PythonBPF

Thanks! Questions?

-

PythonBPF