KernelScript: Unifying eBPF, Userspace, and Kernel Extensions in One Language

Cong Wang

Founder and CEO, Multikernel Technologies, Inc.

eBPF Is Miserable to Write

- eBPF runs your code safely inside the Linux kernel: packet filtering, tracing, security, all at kernel speed
- It powers Cilium, Falco, Katran, bpftrace, and most modern observability
- But to actually write it, you fight C, libbpf, and the verifier, plus pages of boilerplate

> *"The power is real. The developer experience is not best."*

KernelScript

Three Programs, Not One

- The eBPF program itself: restricted C, verifier rules, BPF helpers
- A userspace loader: libbpf calls, map setup, lifecycle management
- Often a kernel module too: custom kfuncs, BTF registration, kernel API's

> *"Three languages, three files, three build systems, for one project."*

Every Tool Is a Compromise

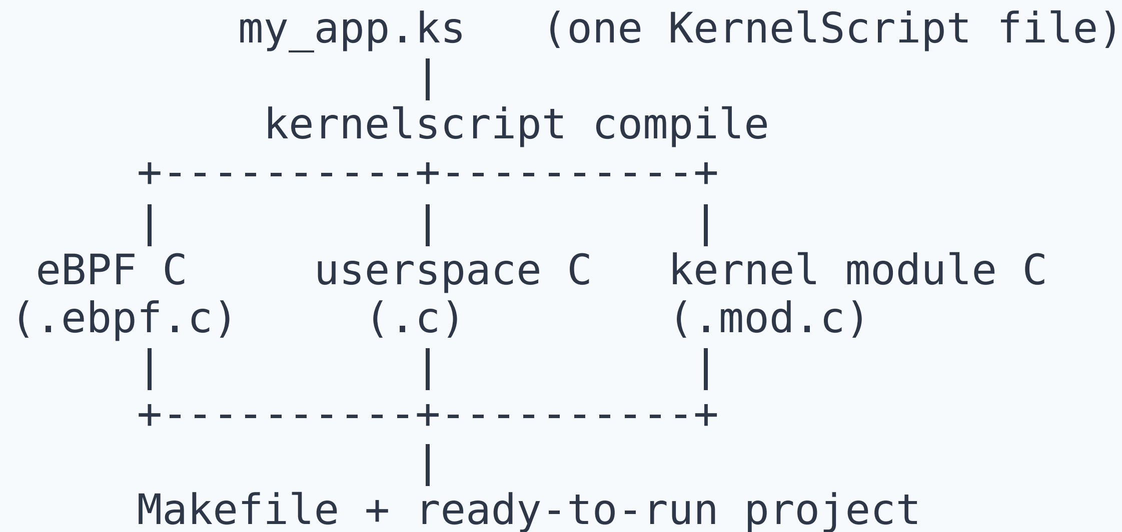
- Raw C + libbpf: full power, brutal learning curve, endless boilerplate
- bpftrace: easy, but tracing only, no XDP or TC, no real data structures
- Rust / Go eBPF libraries: still make you write the eBPF parts in C
- Nothing unifies the kernel side, the userspace side, and the kernel module side

One Language, Three Targets

KernelScript: a type-safe language for eBPF-centric development

- Write eBPF programs, userspace coordination, and kernel functions in one file
- The compiler emits efficient C for each target automatically
- Readable syntax designed around how eBPF actually works
- A modern alternative to procfs and debugfs for kernel customization

One Source, Three Outputs



```
         my_app.ks   (one KernelScript file)
                |
          kernelscript compile
     +----------+----------+
     |          |          |
 eBPF C     userspace C   kernel module C
(.ebpf.c)     (.c)        (.mod.c)
     |          |          |
     +----------+----------+
                |
     Makefile + ready-to-run project
```

- Function attributes decide the target, not separate projects
- You write logic, the compiler writes the plumbing

Your First KenrelScript Program

```
include "xdp.kh"

@xdp fn packet_filter(ctx: *xdp_md) -> xdp_action {
  var packet_size = ctx->data_end - ctx->data
  if (packet_size > 1500) {
    return XDP_DROP
  }
  return XDP_PASS
}

fn main() -> i32 {
  var prog = load(packet_filter)
  attach(prog, "eth0", 0)
  return 0
}
```

- The eBPF program and its loader, in one readable file

Attributes Pick the Target

- `@xdp`, `@tc`, `@probe`, `@tracepoint`, `@perf_event`: compile to eBPF bytecode
- `@helper`: shared subroutines callable from any eBPF program
- `@kfunc`: compiles into a kernel module, callable from eBPF
- `@struct_ops`: pluggable kernel behavior, such as TCP congestion control
- plain `fn`: compiles to the userspace program
- One file, multiple targets, chosen by attribute

> *"Stop juggling toolchains. Annotate and compile."*

Maps That Feel Like Dicts

- Raw eBPF C: `SEC` macros, helper calls, null checks, pointer derefs
- KernelScript: a global variable, indexed like a Python dict
- Same eBPF map, same verifier, same performance
- Compiler emits the boilerplate

> *"If you can use a Python dictionary, you can use a KernelScript map."*

Maps That Read Naturally

```
var connection_count : hash<u32, u64>(1024)
struct PacketInfo { size: u16 }
var ip_stats : hash<u32, PacketInfo>(1024)

// declaration-as-condition: bind only if the key exists
if (var count = connection_count[ip]) {
  return count
} else {
  connection_count[ip] = 1
}

// compound assignment straight into a map entry
ip_stats[ip].size += delta
```

- One presence-checked lookup, in-place update, no write-back
- The compiler turns the idiom into correct, efficient eBPF

Pattern Matching

```
return match (protocol) {
  TCP:  tcp_classifier(ctx),    // transparent tail call
  UDP:  udp_classifier(ctx),
  ICMP: XDP_DROP,
  default: XDP_ABORTED
}
```

- Clean `match` on enums and values
- Calling another eBPF program becomes a **tail call** automatically

Errors, Everywhere the Same

```
try {
  var value = counters[key]
  if (value > 1000) { throw 1 }    // overflow: jump to catch
  return XDP_PASS
} catch 1 {
  counters[key] = 0                // reset the counter
  return XDP_DROP
}
```

- A simple `try` / `catch` replaces C's ugly `goto` cleanup ladders
- Plain integer error codes, the same model for all

Programs Cooperate for Free

KernelScript

```
pin var shared_counter : hash<u32, u32>(1024)

@xdp fn packet_counter(ctx: *xdp_md) -> xdp_action {
  shared_counter[1] = 100
  return XDP_PASS
}

@tc("ingress") fn packet_filter(ctx: *__sk_buff) -> i32 {
  shared_counter[2] = 200
  return TC_ACT_OK
}
```

- Multiple programs, one shared map, zero explicit coordination

Hand Off to Python

```
// my_app.ks
var packet_stats : array<u32, u64>(256)
@xdp fn monitor(ctx: *xdp_md) -> xdp_action { ... }

fn main() -> i32 {
  attach(load(monitor), "lo", 0)
  exec("./analyze.py")                     // replaces this process; never returns
  return 0                                 // unreachable; satisfies the signature
}
```

```
# analyze.py
import my_app as ks                        # auto-generated wrapper
print(ks.packet_stats[5])                  # live eBPF map, in Python
ks.packet_stats[0] = 100
```

- KS emits a Python wrapper exposing every map as a normal object
- Bring NumPy, Pandas, or ML straight to live eBPF data

KernelScript

Kfuncs: Kernel as a Library

```
@kfunc
fn current_time_ns() -> u64 {
  return ktime_get_ns()          // direct kernel API
}

@xdp fn xdp_main(ctx: *xdp_md) -> xdp_action {
  var ts = current_time_ns()   // eBPF calls the kfunc directly
  if (ts == 0) { return XDP_ABORTED }
  return XDP_PASS
}
```

- The kernel module, BTF registration, and loader are all generated

Kernel Types, for Free

BTF is the type information the kernel ships about itself.

- `kernelscript init` reads BTF and pulls in the kernel structs and kfuncs you need
- Typed access to real kernel structures, no hand-copied headers, no guessed layouts
- `--kfuncs` and `--btf-vmlinux-path` target a specific kernel

A Verifier-Friendly Type System

- Fixed-size arrays (`u8[64]`) instead of unbounded ones
- Simple type aliases and structs, no complex generics
- Program handles are typed values: you cannot `attach()` before `load()` succeeds
- The verifier rejects less, because the language steers you away from trouble

Example: Per-IP Packet Quota

```
var packet_counts : hash<u32, u64>(1024)
config network { limit: u32 }

@xdp fn packet_quota(ctx: *xdp_md) -> xdp_action {
  var src_ip = 0x7F000001
  var count: u64 = 1
  if (var prev = packet_counts[src_ip]) { count = prev + 1 }
  packet_counts[src_ip] = count
  if (count > network.limit) { return XDP_DROP }
  return XDP_PASS
}

fn main() -> i32 {
  network.limit = 100                    // tune from userspace
  attach(load(packet_quota), "eth0", 0)
  return 0
}
```

The Compiler Does the Grunt Work

- Tail calls: just call another program; arrays and `bpf_tail_call()` are generated
- Dynptr: use simple pointer operations; complex dynptr APIs handled behind the scenes
- Userspace loaders, map management, and lifecycle: generated
- Makefiles and build system: generated

> *"You describe the behavior. The compiler handles the BPF mechanics."*

KernelScript Workflow

```
kernelscript init xdp hello_world      # initialize a project
cd hello_world
kernelscript compile hello_world.ks    # generate C + Makefile
make                                   # build eBPF + userspace
sudo ./hello_world                     # run it
```

- One workflow from empty directory to a running eBPF program

What You Get Out

```
my_project/
  my_project.ks          # your source
  my_project.c           # userspace program
  my_project.ebpf.c      # eBPF C code
  my_project.mod.c       # kernel module (if you used @kfunc)
  Makefile               # build system
  README.md              # usage instructions
```

- A complete, **buildable** project from a single source file

Not Just Packets

- `@xdp`, `@tc`: networking and packet processing
- `@probe`, `@tracepoint`: kernel tracing and observability
- `@perf_event`: hardware and software performance counters
- `struct_ops`: pluggable kernel behavior, such as TCP congestion control

> *"One language for the entire eBPF subsystem."*

How It Stacks Up

Feature	Raw C + libbpf	Rust eBPF	bpftrace	KernelScript
Syntax	Complex C	Complex Rust	Simple but limited	Clean & readable
Multi-program	Manual	Manual	No	Automatic
Userspace code	Manual	Manual	N/A	Generated
Build system	Manual	Cargo	N/A	Generated
Program types	All	Most	Tracing only	All

Who Is This For?

- Developers who want eBPF's power without a kernel-engineering detour
- Teams tired of maintaining loader code and build glue by hand
- Anyone customizing Linux behavior: networking, security, observability
- A modern alternative to procfs and debugfs for app-specific kernel tuning

Status: Beta Testing

- Syntax, APIs, and features can still change without notice
- Built for experimentation and early feedback, not production yet
- The compiler already handles XDP, TC, probes, perf events, kfuncs, struct_ops
- Please suggest syntax and shape the language together

Try It By Yourself

```
sudo apt install libbpf-dev libelf-dev zlib1g-dev opam bpftool

git clone https://github.com/multikernel/kernelscript.git
cd kernelscript
opam install . --deps-only --with-test
eval $(opam env) && dune build && dune install

# Start development
kernelscript init xdp hello_world

# ...
```

- And browse `examples/`, dozens of working programs to learn from

Future Work

- More eBPF program types: LSM, cgroup, sockops, etc.
- More struct_ops families beyond TCP congestion control
- `kernelscript deploy`: declarative, config-driven deployment of eBPF projects
- Kernel Bundle Image (KBI) support for packaging and distribution

> *"From single-file source to first-class kernel artifact."*

Questions and Feedback

Contact: cwang@multikernel.io

KernelScript project: github.com/multikernel/kernelscript

KBI project: github.com/multikernel/kbi