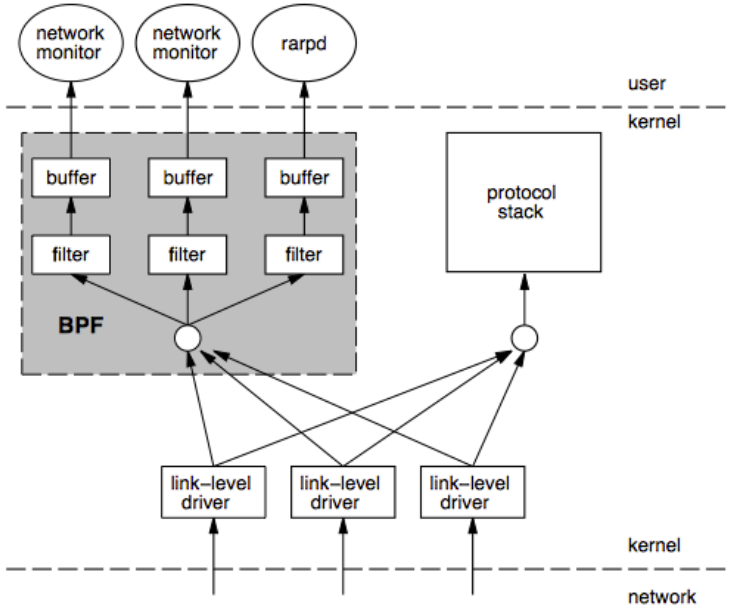
Comparing P4 and eBPF

January 8, 2021

Mihai Budiu
VMware Research Group

BPF

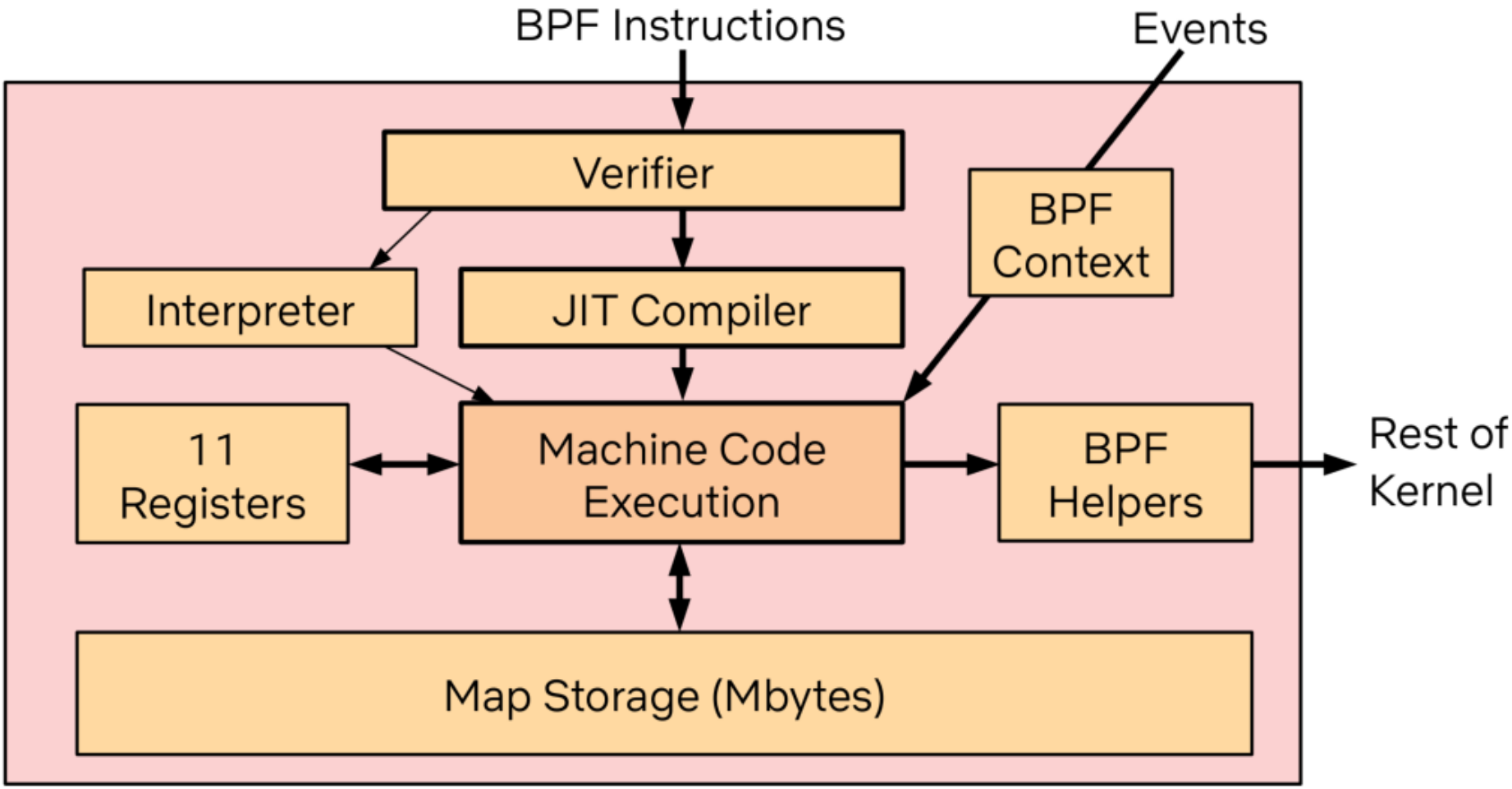
- Berkeley Packet Filters
- Steven McCanne & Van Jacobson, 1992
  http://www.tcpdump.org/papers/bpf-usenix93.pdf
- Instruction set & virtual machine (aka machine language)
- Express packet filtering policies
- Originally interpreted
- “Safe” interpreter in kernel space



EBPF

- Extended BPF, Linux only
- Project leader: Alexei Starovoitov, Facebook
- Larger register set
- JIT + verifier instead of interpreter
- Maps (“tables”) for kernel/user communication
- Whitelisted set of kernel functions that can be called from EBPF (and a calling convention)
- “Execute to completion” model
- C -> EBPF LLVM back-end (gcc **not** yet)
- Used for packet processing and code tracing

BPF Runtime



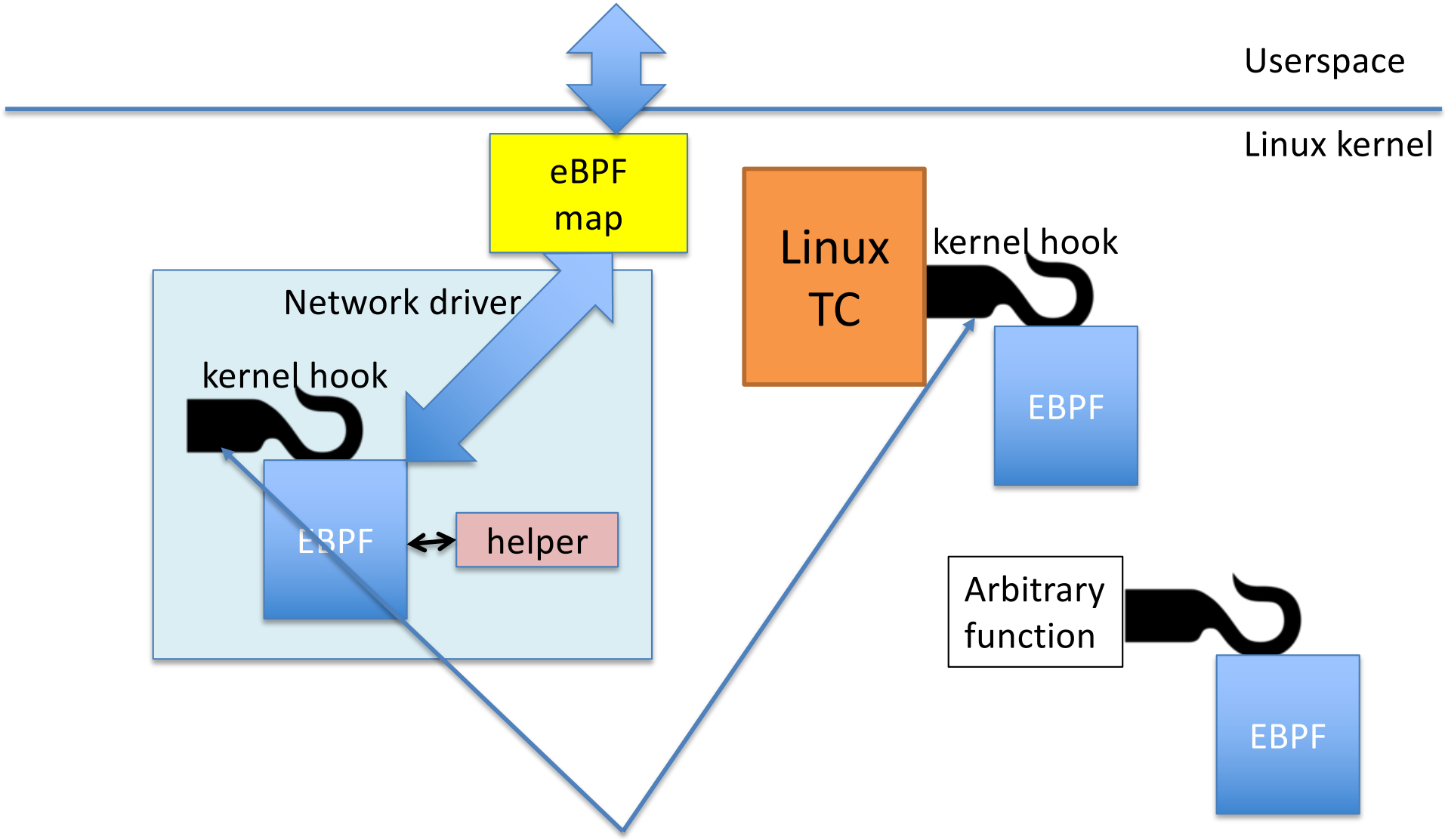
Slide from Brendan Gregg

It’s all about trust

- The main goal is **kernel safety**, not correctness
- Trusted computing base: entirely in kernel
  - BPF loader
  - Kernel verifier (must be simple)
  - Kernel JIT (must be simple)
  - Whitelisted helper functions

- **Not** trusted: compiler, eBPF program writer

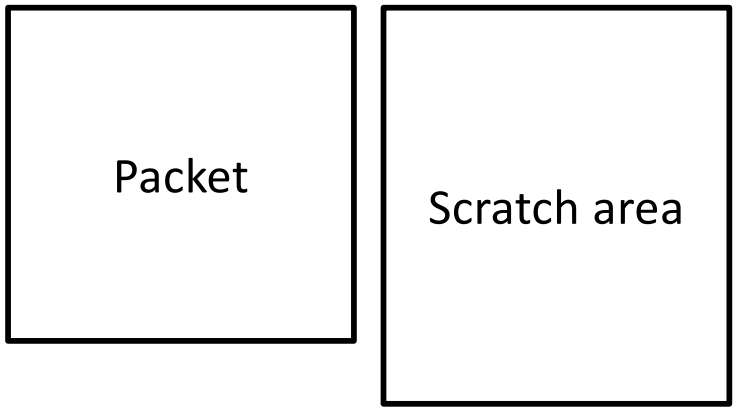
EBPF's world



Each hook provides different capabilities for the EBPF programs.

Note: XDP is just another hook, which allows packet access very early.

BPF Memory Safety

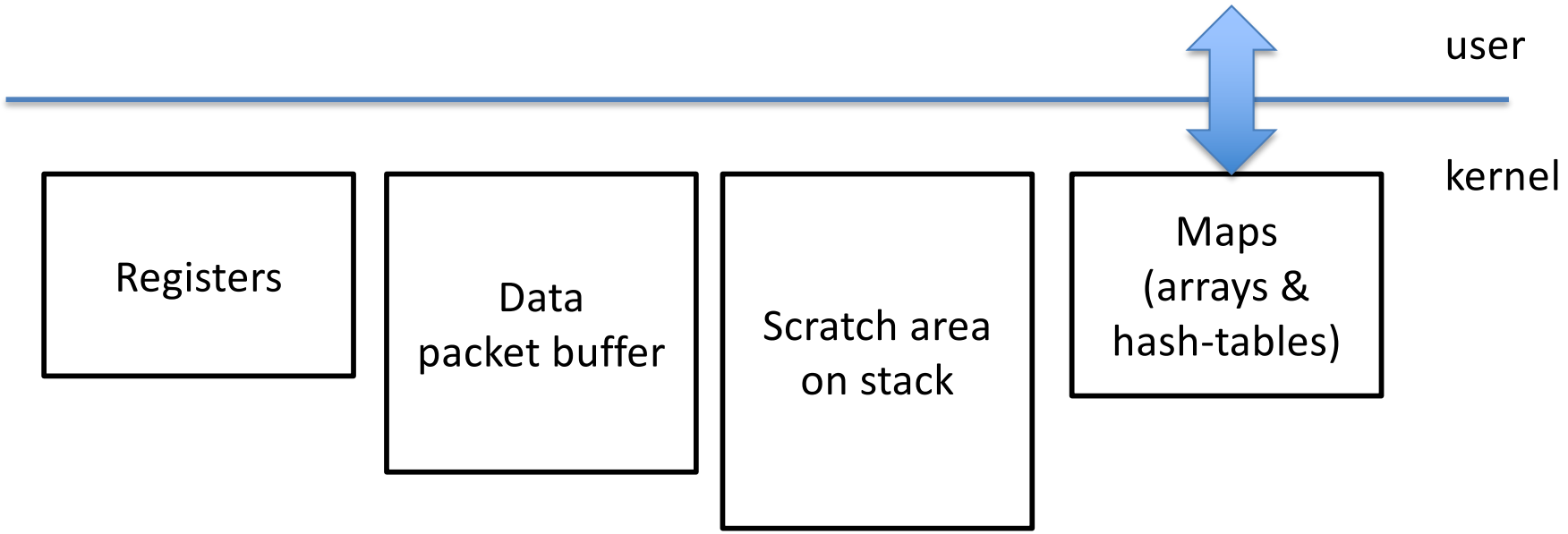


- All memory operations (load/store) are bounds-checked
- Program is terminated on out-of-bounds access

BPF Code Safety

- Code is read-only
- Enforced by static code verifier
- Originally backwards branches were prohibited
  - “Tail calls” are limited to 32
  - Static loop counts are supported
- Branch targets are bounds checked

EBPF Memory Model



(Tracing eBPF programs are allowed to ***read*** (dereference) arbitrary pointers, using the bpf_probe_read_*() helpers, but cannot cause page faults)

EBPF Maps

Userspace-only:

```
int bpf_create_map(int map_id, int key_size, int value_size, int max_entries);
int bpf_delete_map(int map_id);
```

User and kernel:

```
int bpf_update_elem(int map_id, void *key, void *value);
void* bpf_lookup_elem(int map_id, void *key);
int bpf_delete_elem(int map_id, void *key);
int bpf_get_next_key(int map_id, void *key, void *next_key);
```

All of these are multi-core atomic (using RCU)

- Per-core variants possible

Can be attached to file descriptors

- lifetime not connected eBPF programs

BPF Type Format

- Metadata describing BPF program
  - i.e., dynamic type information about tables and data structures
  - BPF loader rewrites BPF program to adjust offsets to currently running kernel
- Generated by LLVM (aka DWARF)
- Enables portable BPF programs (cross-compiled)

Packet Processing Model