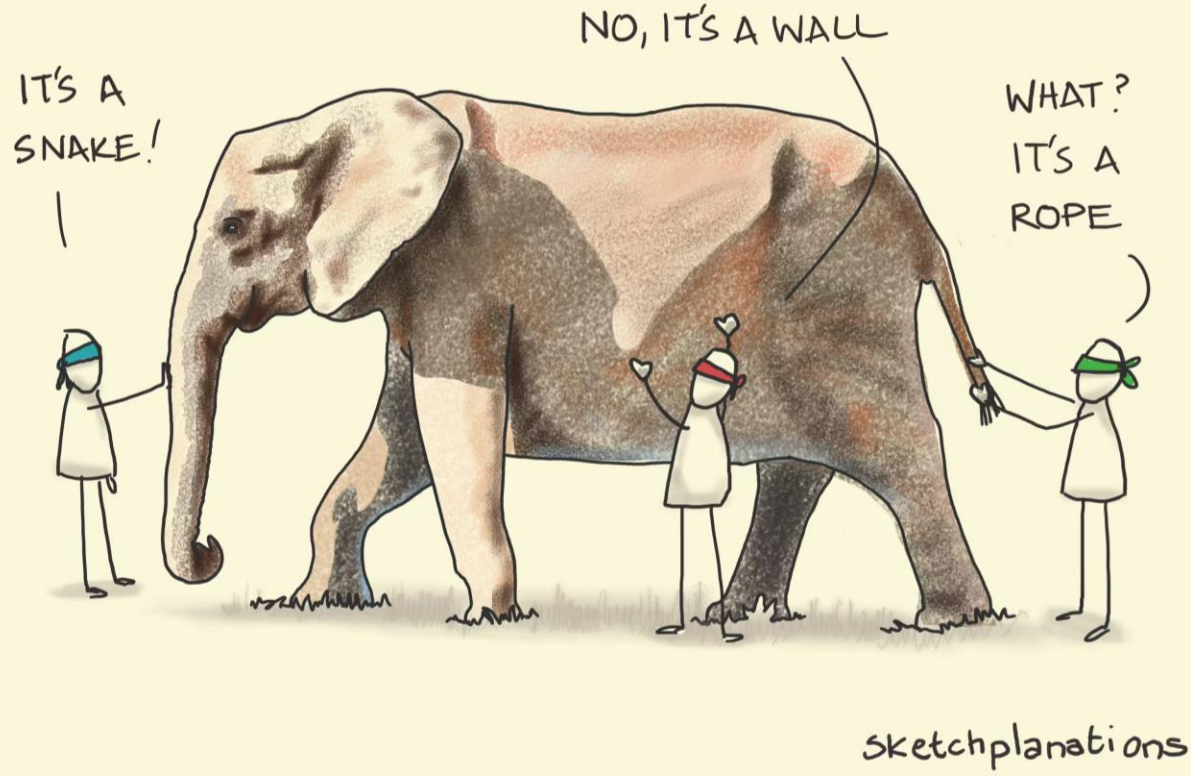
Tailoring eBPF maps for DDoS protection

Ivan Koveshnikov

Software Engineer

How are *we* using eBPF?

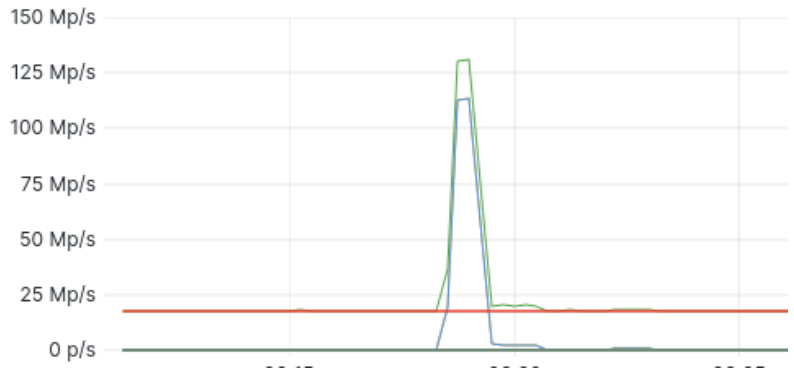
- Infrastructure as a service, middleware
- Different customers to protect, resources spawned dynamically. Different areas of networking: gaming, hosting, DNS, etc.
- Resource isolation
- Long and flexible eBPF pipelines
- Deep packet inspection, e.g. with regular expressions (Netdev 0x16 Github)
- Runtime configuration and code updates (Netdev 0x17)
- Asynchronous traffic loads, ingress >> egress
- Unidirectional and bidirectional traffic analysis

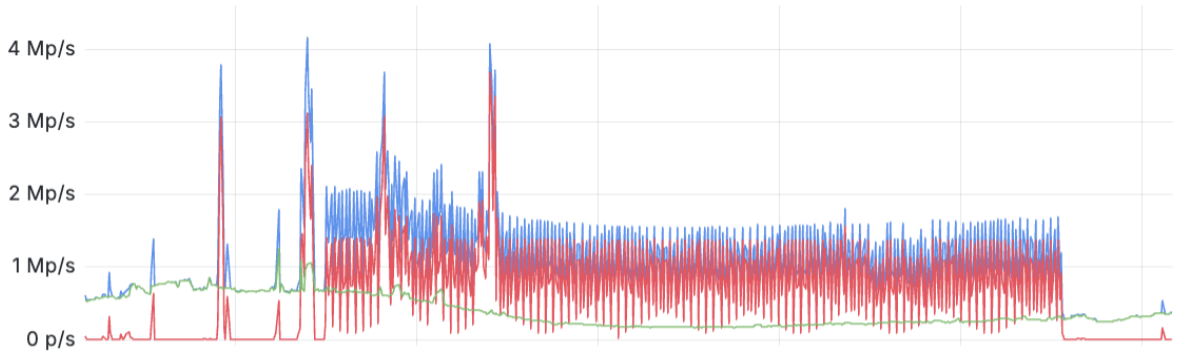


GCORE

What makes DDoS protection different?

- Low tolerance to legitimate traffic drop, but can be increased under attacks.
- Acceptable level of garbage traffic leaks may greatly vary between services.
- Filtering is not enough. Active querying, caching, connection stealing—multiple methods need to be combined.
- Raging spikes! Not only about working efficiently, but about breaking expectedly, predictably, and efficiently.





GCORE

01

Rate limiters

Surviving traffic spikes

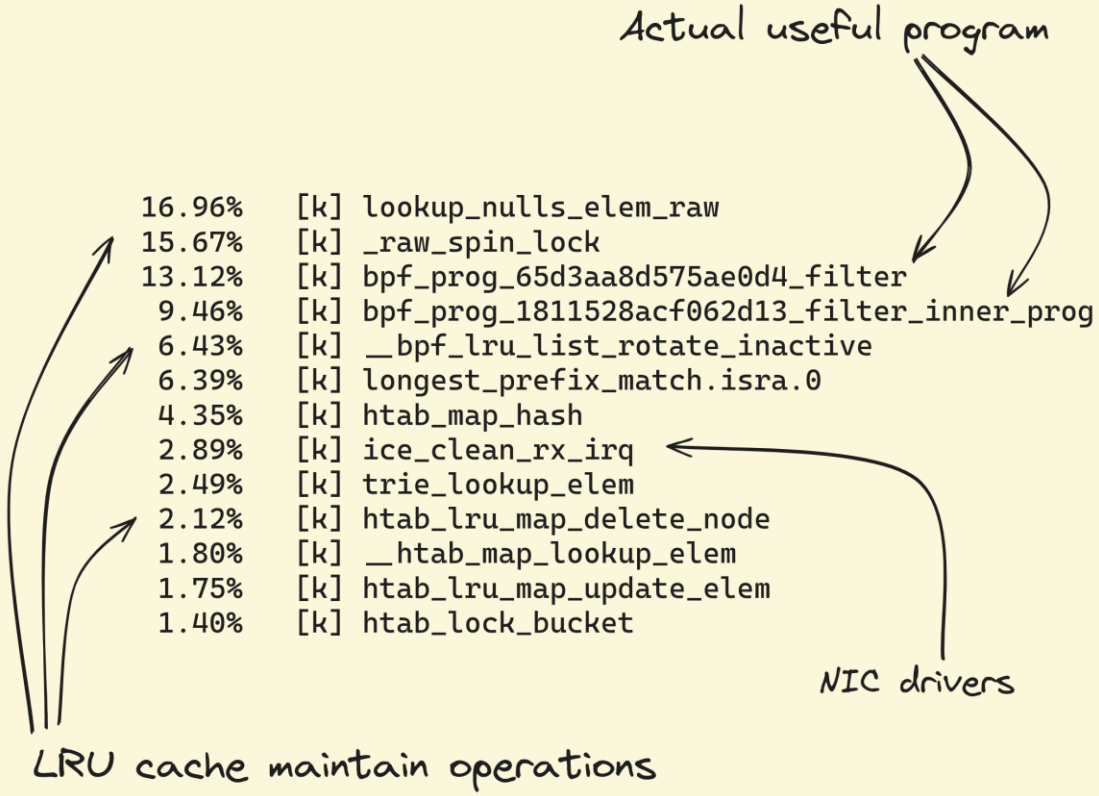
Control-plane DDoS:

- Userspace companion app overloads by events notifications via ring buffers or perf events

Data-plane operation jams:

- State allocation for statefull firewall
- ACL flow caching

NIC drivers



Rate limits are everywhere

- bpf_map_lookup_elem() to get rate limit configuration and state

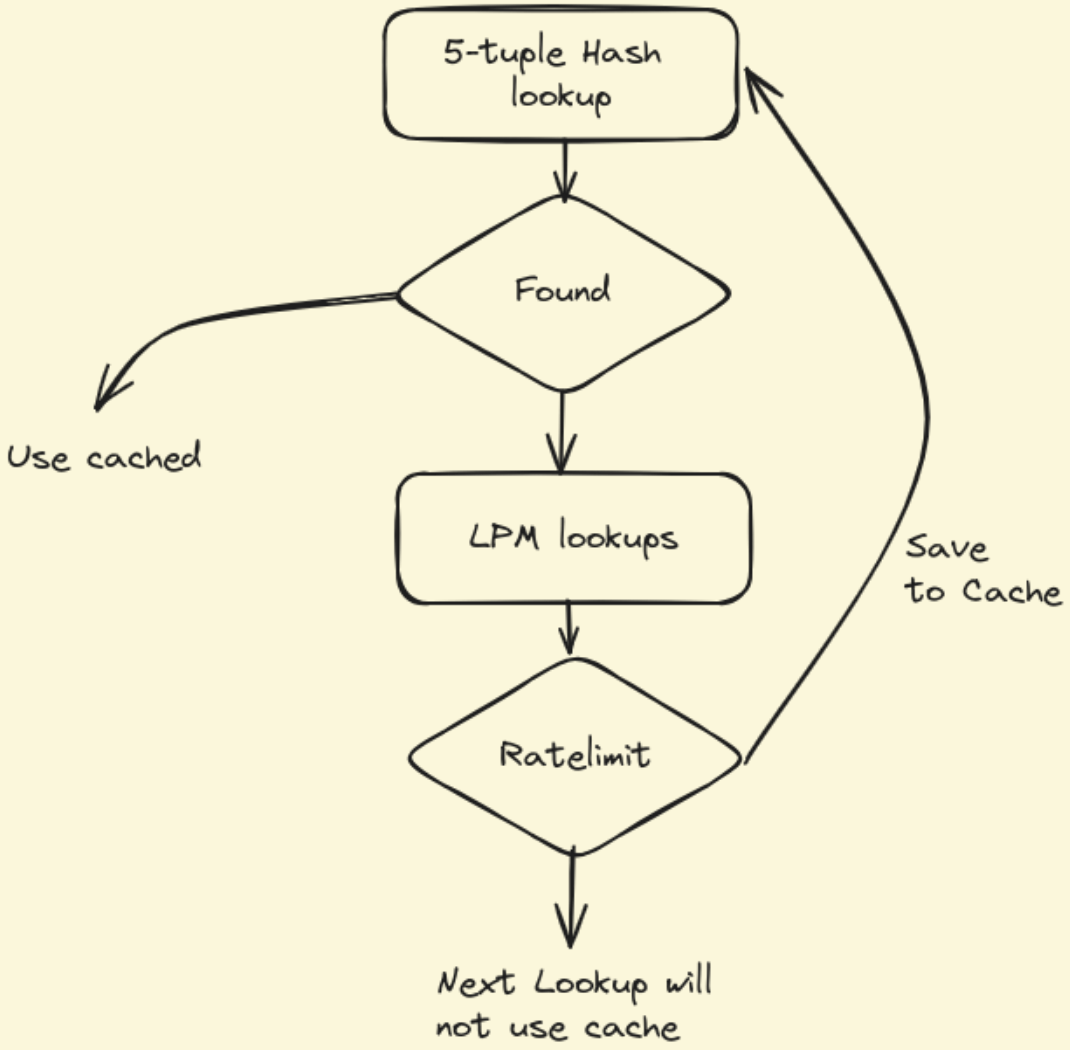
  ↓

- Checking for allowed credits/timestamps

  ↓

- Desired bpf_map_update_elem() operation

All bpf_map_update_elem() calls are end up wrapped into ratelimits



GCORE

Example: ACL flow cache

Naïve implementation of ACL flow cache: 2 LPM maps + 3 arrays.

LRU-hash for caching, populated from eBPF side. Caches are unique for each CPU and allocated on the closest NUMA node.

The rate limiter is implemented per-CPU.

100k flows No Caching	135.35 Mpps	28.36% 19.08% 13.66% 6.37% 5.80%	bpf_prog_filter bpf_prog_filter_acl longest_prefix_match.isra.0 htab_map_hash lookup_nulls_elem_raw
100K flows With Caching	226.74 Mpps	40.12% 10.77% 8.93% 5.86% 5.07%	bpf_prog_filter lookup_nulls_elem_raw htab_map_hash ice_clean_rx_irq memcmp
100K flows With Caching With Rate limit	223.64 Mpps	40.25% 10.92% 8.87% 5.75% 5.07%	bpf_prog_filter lookup_nulls_elem_raw htab_map_hash ice_clean_rx_irq memcmp
10M flows No Caching	128.02 Mpps	24.88% 19.80% 13.76% 12.10% 5.57%	bpf_prog_filter bpf_prog_filter_acl longest_prefix_match.isra.0 lookup_nulls_elem_raw htab_map_hash
10M flows With Caching	66.45 Mpps	16.96% 15.67% 13.12% 9.46% 6.43% 6.39% 4.35%	lookup_nulls_elem_raw _raw_spin_lock bpf_prog_filter bpf_prog_filter_acl __bpf_lru_list_rotate_inactive longest_prefix_match.isra.0 htab_map_hash
100K flows With Caching With Rate limit	110.85 Mpps	23.65% 22.52% 15.68% 11.12% 4.88%	lookup_nulls_elem_raw bpf_prog_filter bpf_prog_filter_acl longest_prefix_match.isra.0 htab_map_hash

GCORE

Proposal

- Not for performance improvements but an easier way to create safe applications. **DDoS-safety from the box.**
- Allowed only for maps with dynamically allocated elements.
- Independent per-CPU calculations. Provide high watermark against overloading, not a strict operation limit.
- Insert operations can follow or bypass the rate limit.
- A single if() overhead if not needed.

Create map:

```
struct bpf_map_create_opts opts = { 0 };
opts.sz = sizeof(opts);
opts.insert_rlim.limit = VALUE;
opts.insert_rlim.timeframe = FRAME;
bpf_map_create(BPF_MAP_TYPE_LRU_HASH, map_name,
    sizeof(key), sizeof(value), MAX_ENTRIES, &opts);
```

OR

```
struct map_name_ratelimit {
__uint(limit, VALUE);
__uint(timeframe, FRAME);
}

struct {
__uint(type, BPF_MAP_TYPE_LRU_HASH);
__type(key, struct key);
__type(value, struct value);
__uint(max_entries, MAX_ENTRIES);
__type(insert_rlim, struct map_name_ratelimit);
} map_name SEC(".maps");
```

Insert at runtime:

```
bpf_map_update_elem(&map_name, &key, &value, BPF_ANY);
bpf_map_update_elem(&map_name, &key, &value, BPF_ANY |
BPF_NO_RATELIMIT);
```

GCORE

02

TTL maps

Objects with limited lifetime

Very native to data plane applications:

- Flow caches
- Stateful firewall tables
- Temporal block lists
- Cached responses

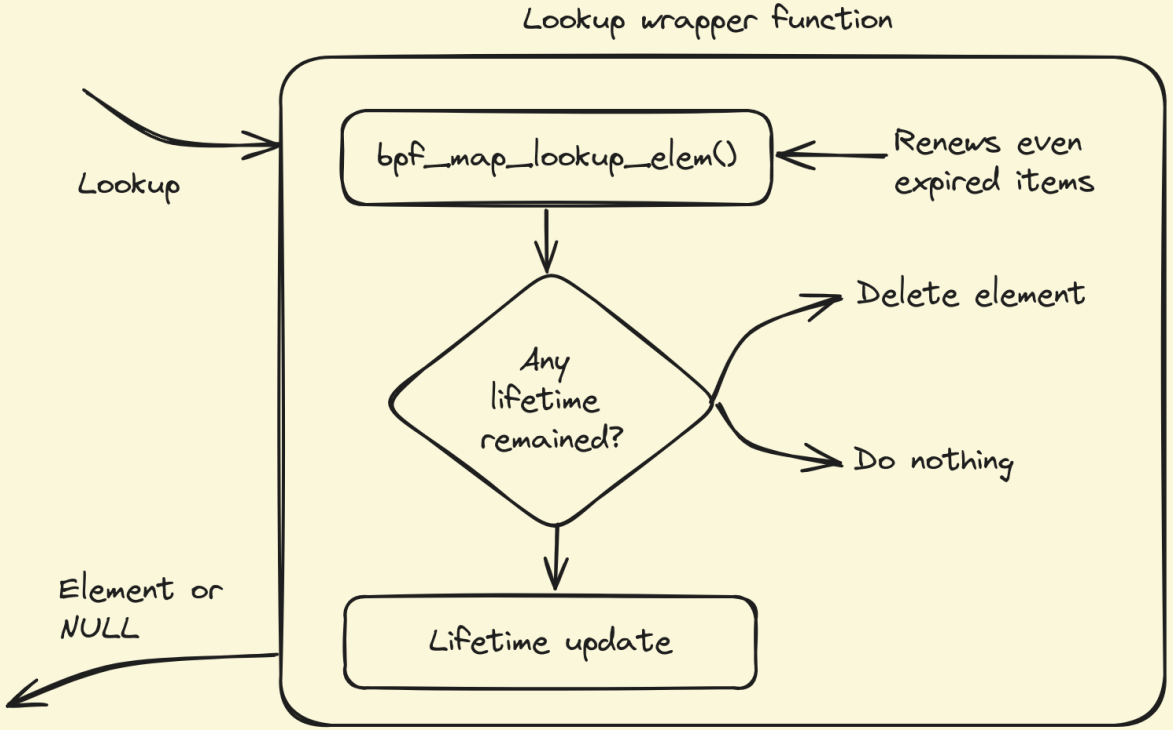
Access patterns:

- Renew on lookup
- Renew on update

GCORE

TTL over LRU

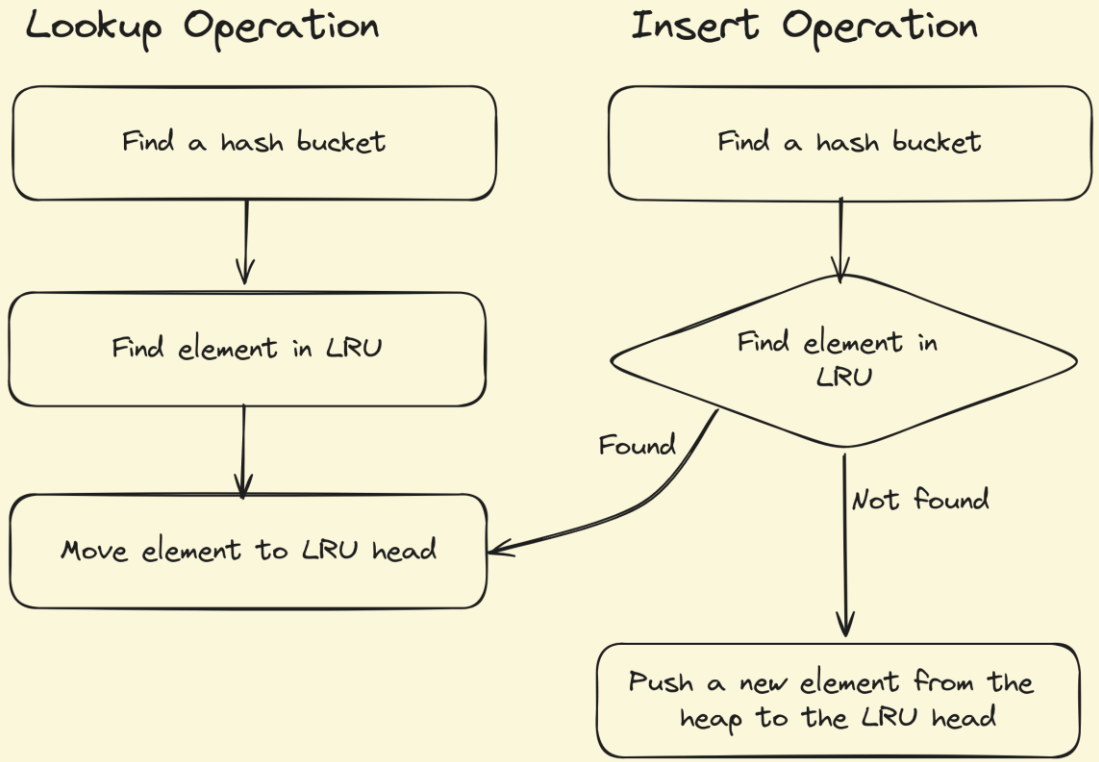
- Each entry contains remaining lifetime. Atomic operations on updating that lifetime are not required, as the time is always pushed forward, some data races are allowed.
- Wrappers over bpf_map_lookup_elem() inside eBPF code check and renew the lifetime on lookup. They return NULL to the caller if the lifetime is expired.
- The kernel is not aware of the lifetime, it can remove entry at any time or keep it forever. Lifetime depends only on the insertion rate. An “expired” element can be accessed frequently and saved from the garbage collector.



GCORE

Limitations of LRU maps

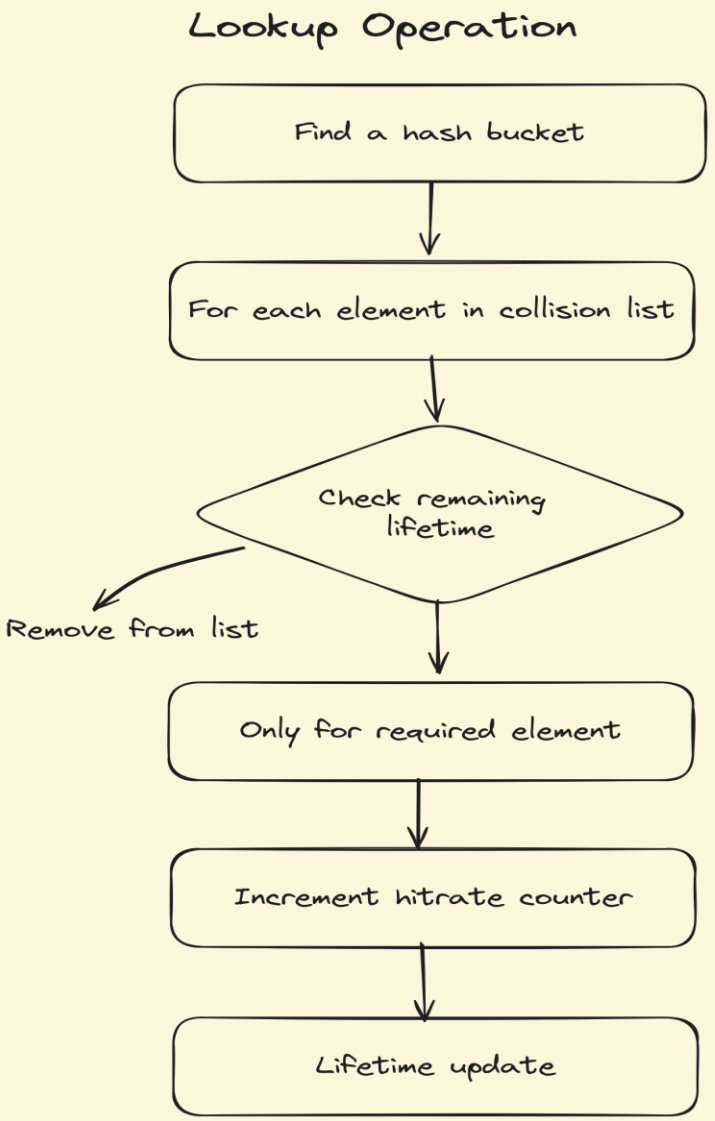
- Main usage: self-supporting hash table that doesn't require a garbage collector.
- Space-based eviction. Reliable and fast.
- No guarantees how long an entry will be kept in the map.
- Shared index, so LRU maintain operations may slow down other flows.
- BPF_F_NO_COMMON_LRU speed ups self-maintaining, but space effectiveness breaks when only some CPUs work with that map.
- Garbage collection from the userspace is not reliable, as the next key may be invalidated at any time.



GCORE

TTL hash in the kernel

- A shared search index between all CPUs.
- The value type must inherit base value class, which exposes remaining lifetime and hit rate to the caller. Both the hit rate and the lifetime may be not 100% accurate to reduce atomic pressure.
- The renew strategy is defined when the map is created.
- Shadow evictions: on the lookup, outdated entries might be removed from the map.
- Explicit garbage collection can be done via (batch) lookup operations from the userspace.
- When the map is full, all new insertions will be failed. Exposed lifetime and hit rate allow smart evictions from the userspace.



GCORE

Proposal

- New map types: TTL_HASH and PERCPU_TTL_HASH
- New map options: lifetime updates
- Lifetime stores coarse monotonic time since boot
- Runtime updates for 'lifetime' configuration is mandatory - but not for the update strategy

```
Struct bpf_ttl_hash_entry_header {
  __u64 lifetime;
  __u64 hitrate;
  __u8 data[0];
}
```

Create map:

```
struct bpf_map_create_opts opts = { 0 };
opts.sz = sizeof(opts);
opts.entry_ttl.lifetime = VALUE;
opts.entry_ttl.update_on_lookup = true;
bpf_map_create(BPF_MAP_TYPE_TTL_HASH, map_name,
    sizeof(key), sizeof(value), MAX_ENTRIES, &opts);
```

OR

```
struct map_name_entry_ttl {
__uint(lifetime, VALUE);
__uint(update_on_lookup, true);
}

struct {
__uint(type, BPF_MAP_TYPE_LRU_HASH);
__type(key, struct key);
__type(value, struct value);
__uint(max_entries, MAX_ENTRIES);
__type(entry_ttl, struct map_name_entry_ttl);
} map_name SEC(".maps");
```

Visibility of hitrate

- Smart garbage collection
- Drop oldest entries first
- Or drop entries with low hitrate - slowloris attacks
- Or drop entries with hig hitrate – spammers

03

Runtime option updates

Dynamic nature of map options

- Map reuse on eBPF code update
- Reaction to traffic overloads

Proposal

- bpf_obj_get_info_by_fd() gives read access to all the map options already
- Only a few options can reasonably get updates at the runtime
- Setsockopt(2)-like approach is nice to have for rate limit and lifetime updates
- Problems: BTF information is not dynamic

```
bpf_map_option_set(int map_fd, int option,
                   size_t size, void *option)
```

```
Example:
struct bpf_map_opt_insert_rlim lim =
        {.limit = VALUE, .timeframe = FRAME};
bpf_map_setopt(fd, BPF_MAP_OPT_INSERT_RLIM, &lim,
sizeof(lim));
```

Thank you!

gcore.com