ISOVALENT

BPF "static keys"
Wildcard map
XXH3 hash

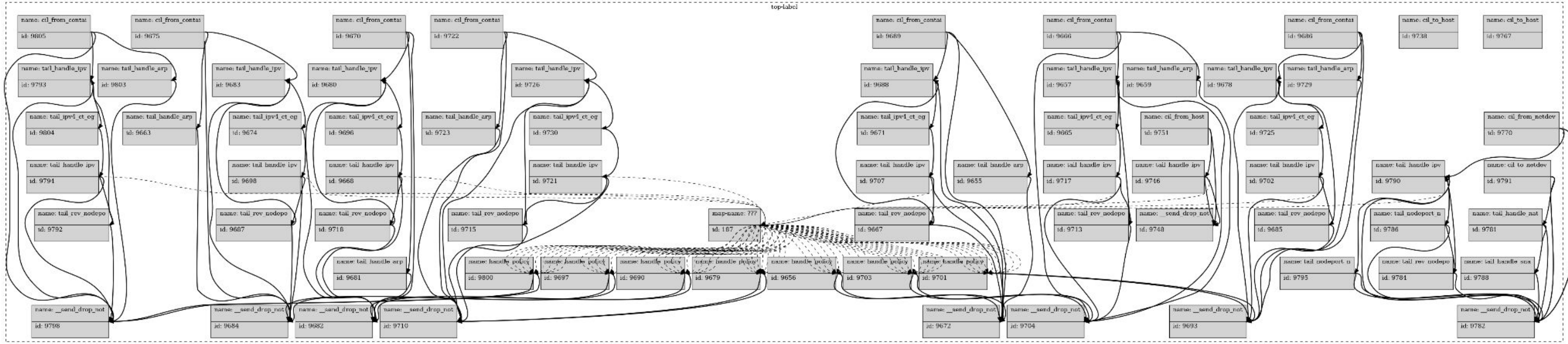
LSF/MM/BPF 2023

Anton Protopopov

Agenda

- BPF static key/branch
- Wildcard map: the current state
- XXH3 usage in BPF: stacktrace map and/or hashmap/bloom filter

Cilium: how to trace BPF tail calls?



Debugging with pwru

- pwru can be used to reply packet path inside the Linux kernel
- We want to also see what BPF datapath does in a similar manner
- Solution 0: don’t use tail calls, use BPF-to-BPF calls
- Solution 1: #ifdef, reload programs (works, but scary [not enough mem, complexity, etc…])
- Solution 2: insert an empty noinline function @ tailcall entry (adds ~1.7ns/tailcall, so, say, ~10ns/packet)
- Solution 3: patch tail calls to run fentry (I have a hack to do this, but looks like this is not visibly cheaper than just inserting a dummy noinline function)
- Solution 4: BPF “static keys” or alike: branch with zero overhead

BPF static keys

```
__section("kprobe/__x64_sys_getpgid")
int worker(void *ctx)
{
        if (bpf_static_branch(&debug_key)) {
                bpf_printk("hello from st. branch");
        }

        bpf_printk("whatever...");

        return 0;
}
```

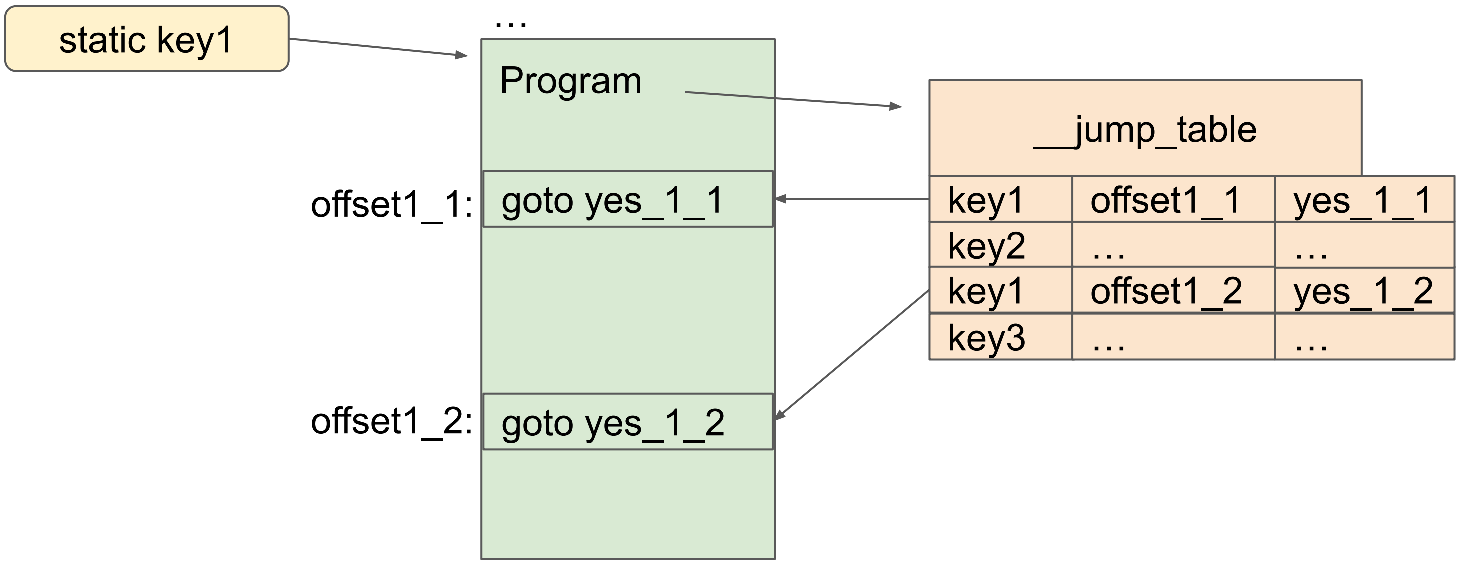
BPF static keys

```
struct {
        __uint(type, BPF_MAP_TYPE_ARRAY);
        __type(key, __u32);
        __type(value, __u32);
        __uint(map_flags, BPF_F_STATIC_BRANCH);
        __uint(max_entries, 1);
} debug_key __section(".maps");
```

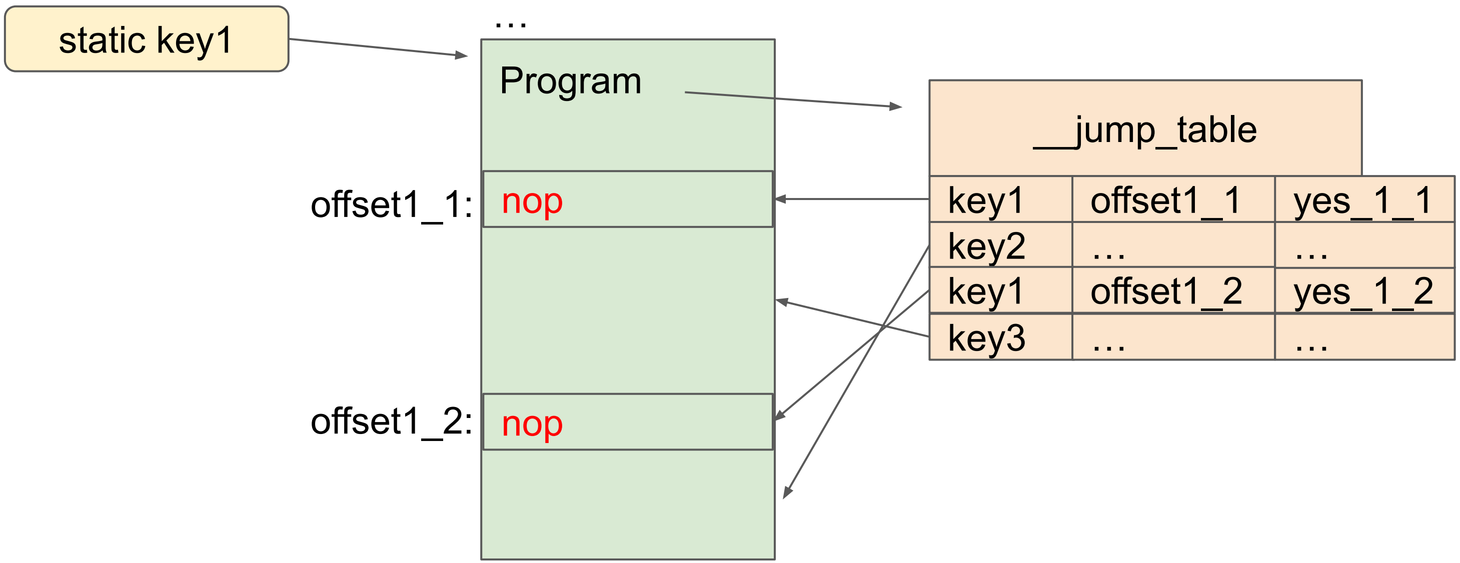
BPF static keys

```
static __always_inline bool bpf_static_branch(void *map)
{
        asm goto("1:"
                "goto %l[l_yes]\n\t"
                ".pushsection __jump_table,  \"aw\" \n\t"
                ".balign 8\n\t"
                ".long 1b - . \n\t"
                ".long %l[l_yes] - . \n\t"
                ".quad %c0 - .\n\t"
                ".popsection \n\t"
                :: "i" (map)
                :: l_yes);
        return false;
l_yes:
        return __lookup_and_deref_static_branch(map);
}
```

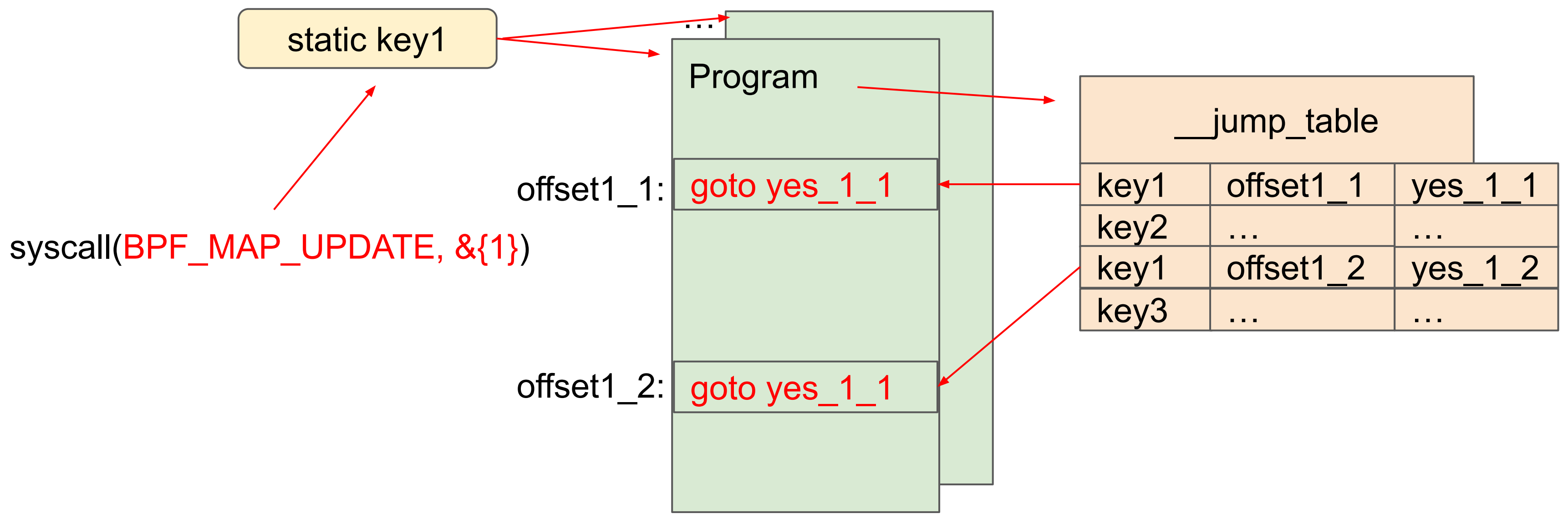
BPF static keys: what verifier thinks on load



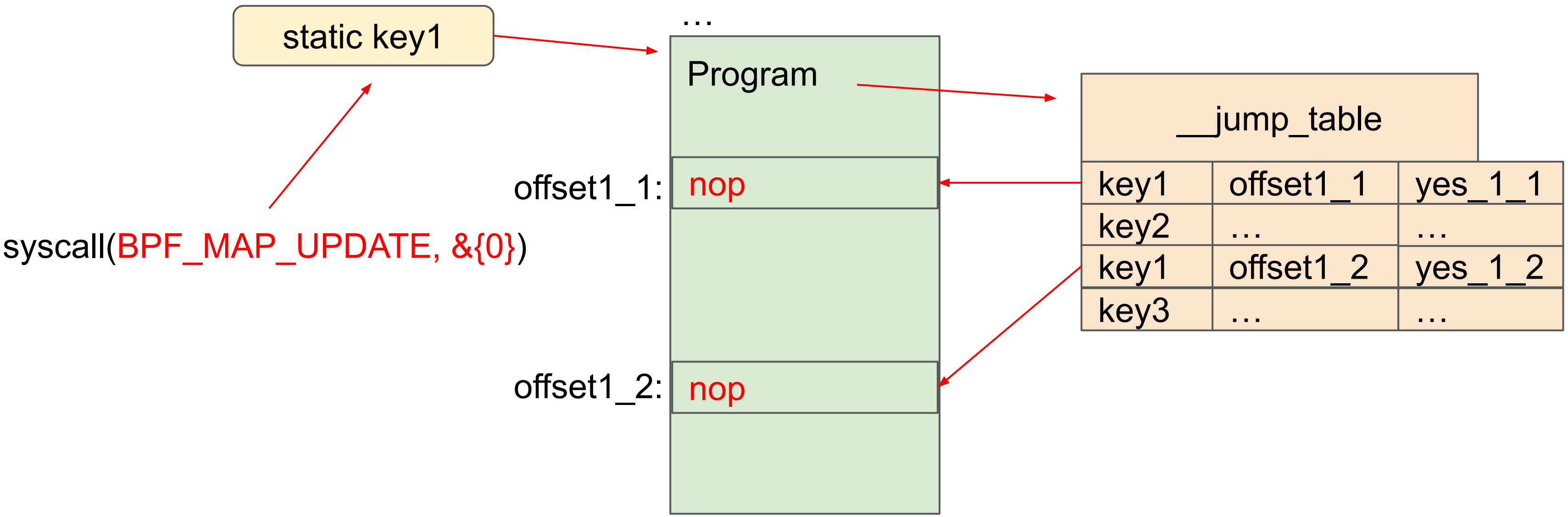
BPF static keys: after verified we patch goto->nop



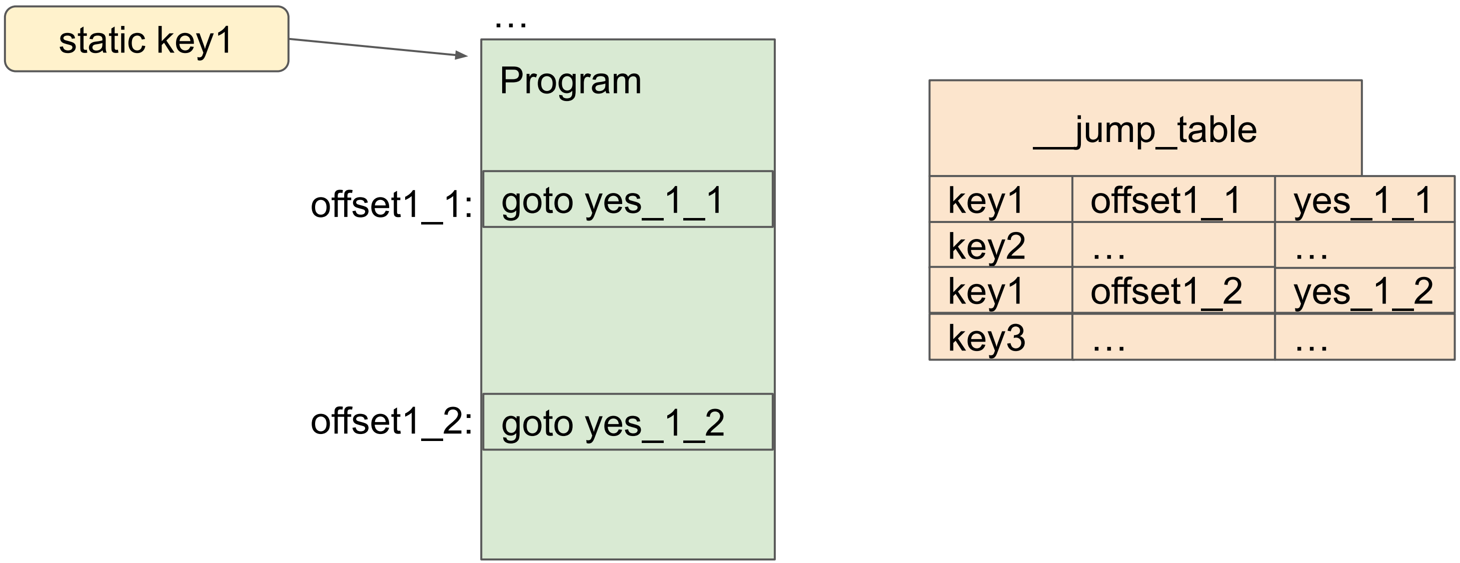
BPF static keys: if map value != 0, then we patch



BPF static keys: map value = 0, then back to nop



How to pass __jump_table to kernel?



Wildcard map: use cases

- We want to classify input with bpf_map_lookup_elem() for such cases:
- 4/5-tuple, say [192.68.0.0/24, *, *, 22] (cilium packet recorder)
- Cilium firewall: [security_id, dport, protocol, direction]
- LPM: [192.68.0.0/24] (e.g., for geoip mapping)
- …
- (one map per use case)

Wildcard map: two types of keys

- For bpf_map_lookup_elem() we support two types of keys: rule and match
- (For other operations only rule is supported)
- Example: (ID, port range):

Key type rule:

type=rule priority=X	ID	port_min	port_max

Key type match:

type=match priority=0	ID	port	0

Wildcard map: types of rules

- Every type is of size N=1,2,...,16 bytes

BPF_WILDCARD_RULE_PREFIX:

prefix (N bytes)	prefix_len (u32)

prefix (N bytes)

BPF_WILDCARD_RULE_RANGE:

min (N bytes)	max (N bytes)

value (N bytes)

BPF_WILDCARD_RULE_MATCH/BPF_WILDCARD_RULE_WILDCARD_MATCH:

value (N bytes)

value (N bytes)

Wildcard map: API

- To create map we need to specify a list of rule formats

```
BPF_WILDCARD_DESC_2(
        policy,
        BPF_WILDCARD_RULE_PREFIX, __u32, id,
        BPF_WILDCARD_RULE_RANGE, __u16, dport
);
struct {
        __uint(type, BPF_MAP_TYPE_WILDCARD);
        __type(key, struct policy_key);
        __type(value, __u64);
        __uint(max_entries, MAX_ENTRIES);
        __uint(map_flags, BPF_F_NO_PREALLOC);
} NAME SEC(".maps")
```

Wildcard map: API

- The description is passed to the kernel as part of key BTF:

```
struct policy_key {
        struct policy_desc desc[0];
        __u32 type;
        __u32 priority;
        union {
                struct {
                        __u32 id;
                        __u16 dport_min;
                        __u16 dport_max;
                } __attribute__((packed)) rule;
                struct {
                        __u32 id;
                        __u16 dport;
                } __attribute__((packed));
        };
} __attribute__((packed));
```

Map description

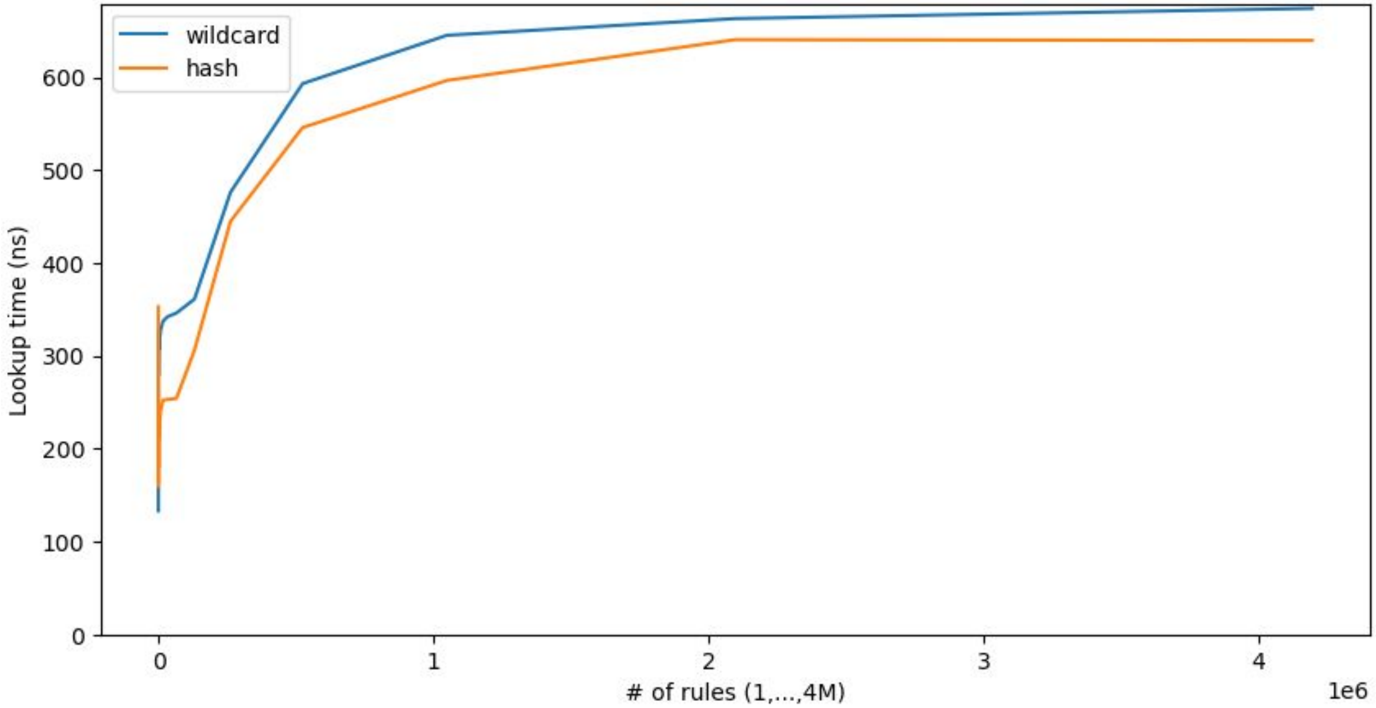
Rule exact format

Match exact format

Wildcard map: cilium firewall

- Motivation: k8s requires netpolicies with port ranges
- Current implementation:
  - Given a **[security_identity, dport, protocol, direction]** tuple do [up to] 6 lookups:
    - (security_identity, dport, protocol, direction)
    - (*, dport, protocol, direction)
    - (security_identity, *, protocol, direction)
    - (*, *, protocol, direction)
    - (security_identity, *, *, direction)
    - (*, *, *, direction)
- With wildcard map:
  - Just do one lookup **(security_identity, dport, protocol, direction)**
  - This actually is automatically translated to a similar algorithm to the above, but is configured dynamically by installing different types of rules. Say, if we only have a wildcard entry, then this is 6 times faster. Plus port ranges are supported.

Cilium firewall (random input, 1…1M rules)



Wildcard map: cilium packet recorder

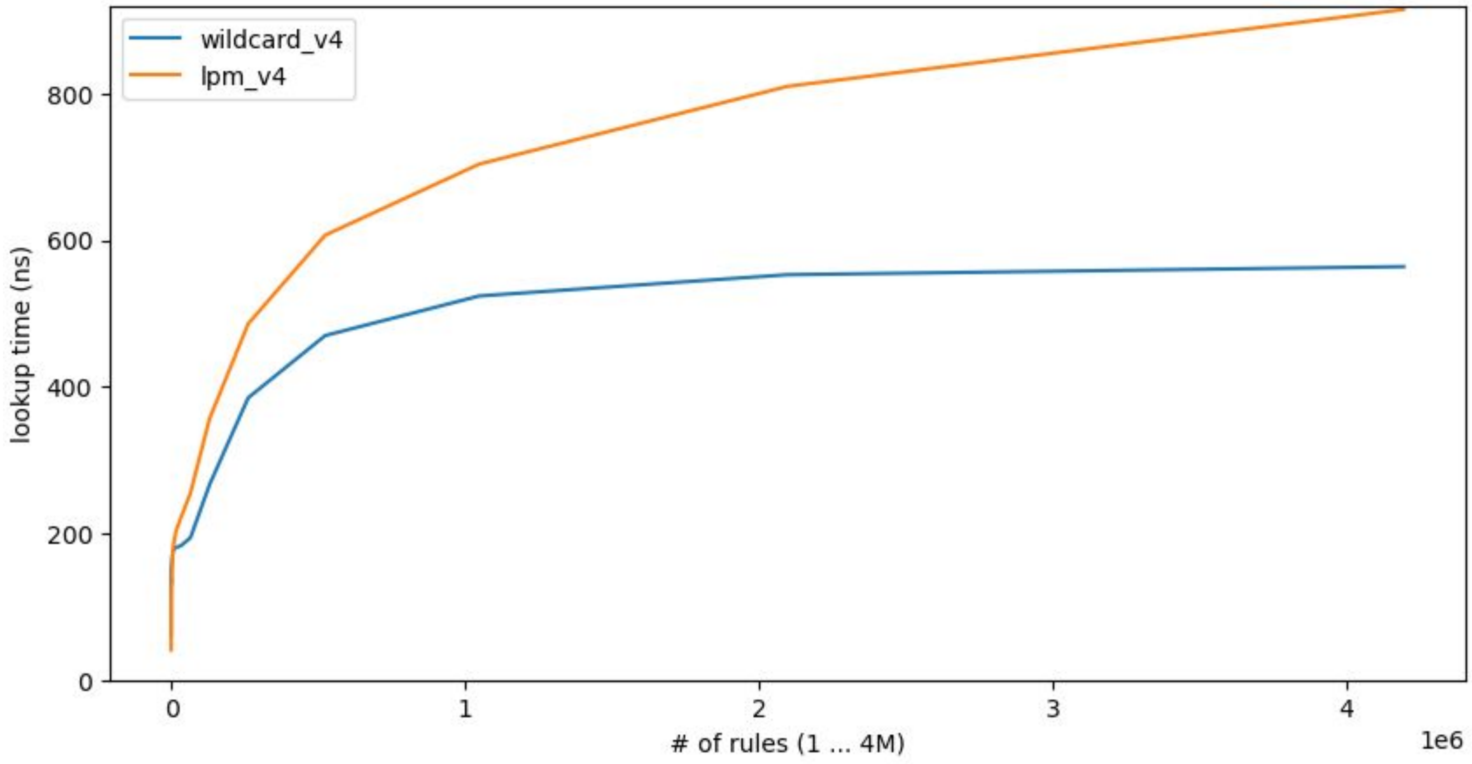
- We want to be able to filter packets by ⅘-tuple:
- [167.138.128.0/17, 10.0.0.0/24, *, 22]
- [*, 10.0.0.0/24, *, 1-1024]

- The map usage is straightforward

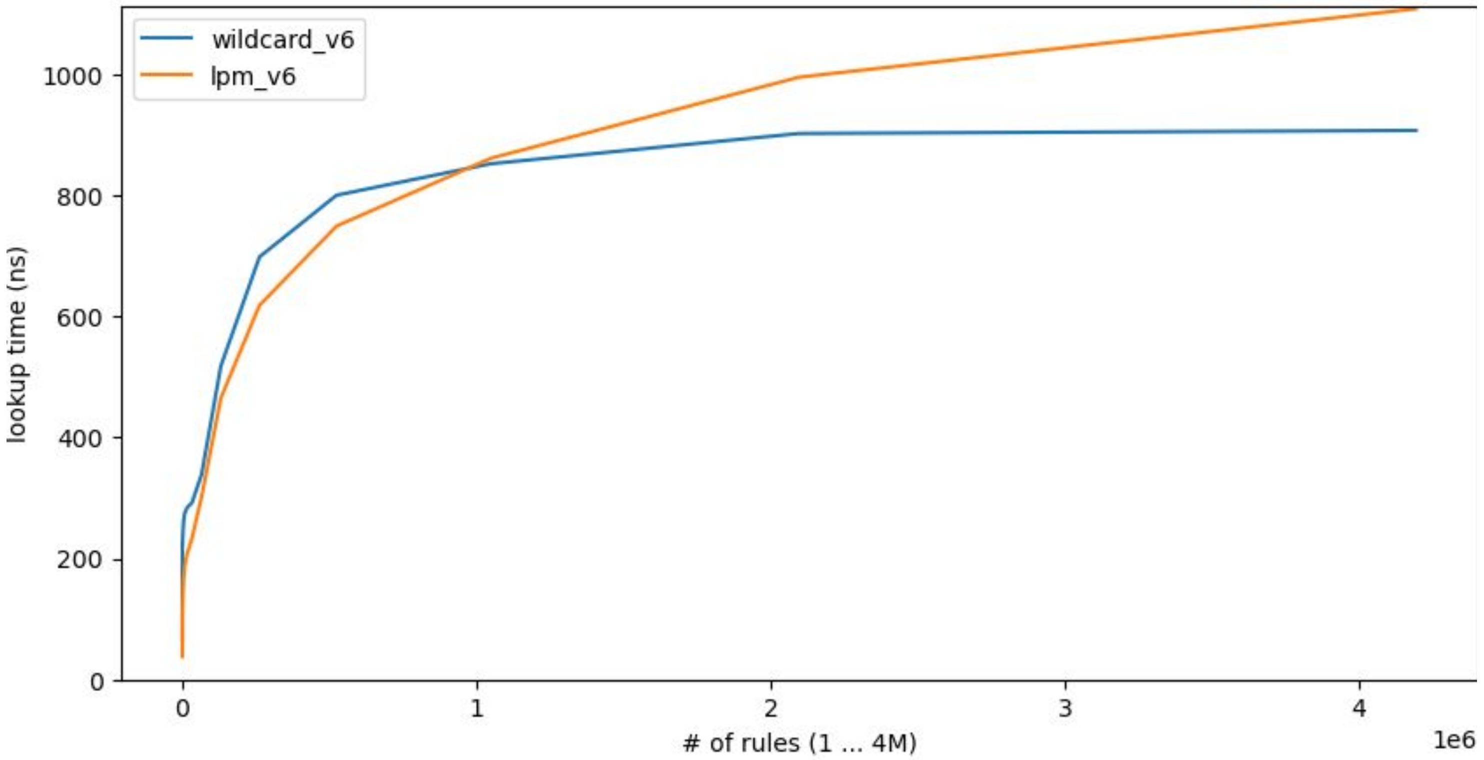
Wildcard map: geoIP

- Wildcard map can mimic LPM trie:
  - LPM{ prefix, addr }
  - Wildcard{ **.priority = 32- prefix**, addr, prefix }
- Works faster than LPM (if rules are prepared properly):
  - IPv4: 3.7M entries, LPM: **1189ns/packet,** wildcard: **709ns/packet**
  - IPv6: 1.3M entries, LPM: **1090ns/packet,** wildcard: **1550ns/packet**

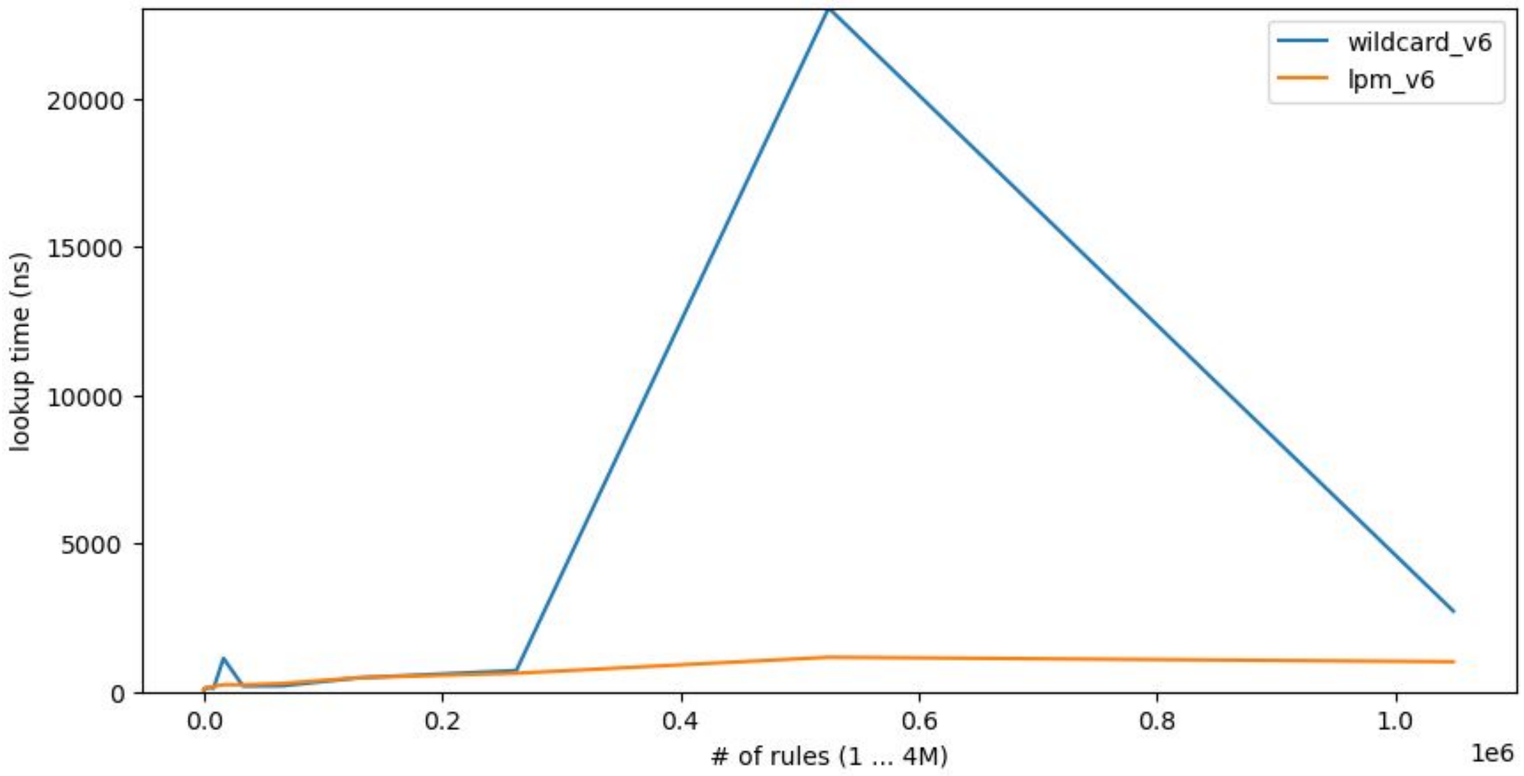
Wildcard map: random input, v4, “offline”



Wildcard map: random input, v6, “offline”



Wildcard map: random input



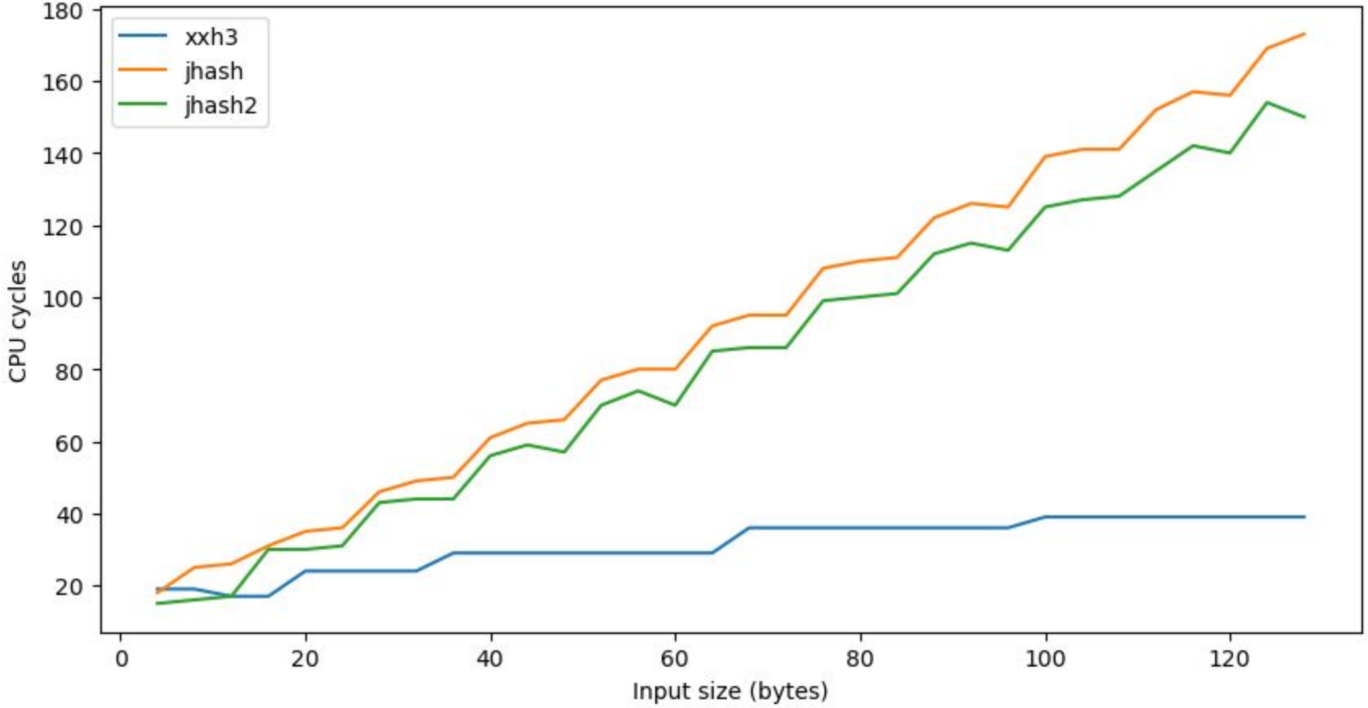
Wildcard map: WIP…

- So wildcard map still can degrade for different inputs
- There are ways to fix this, however, this is WIP
- Maybe will have to switch to another backend algorithm after all, as TupleMerge turned out to be really hard to ride

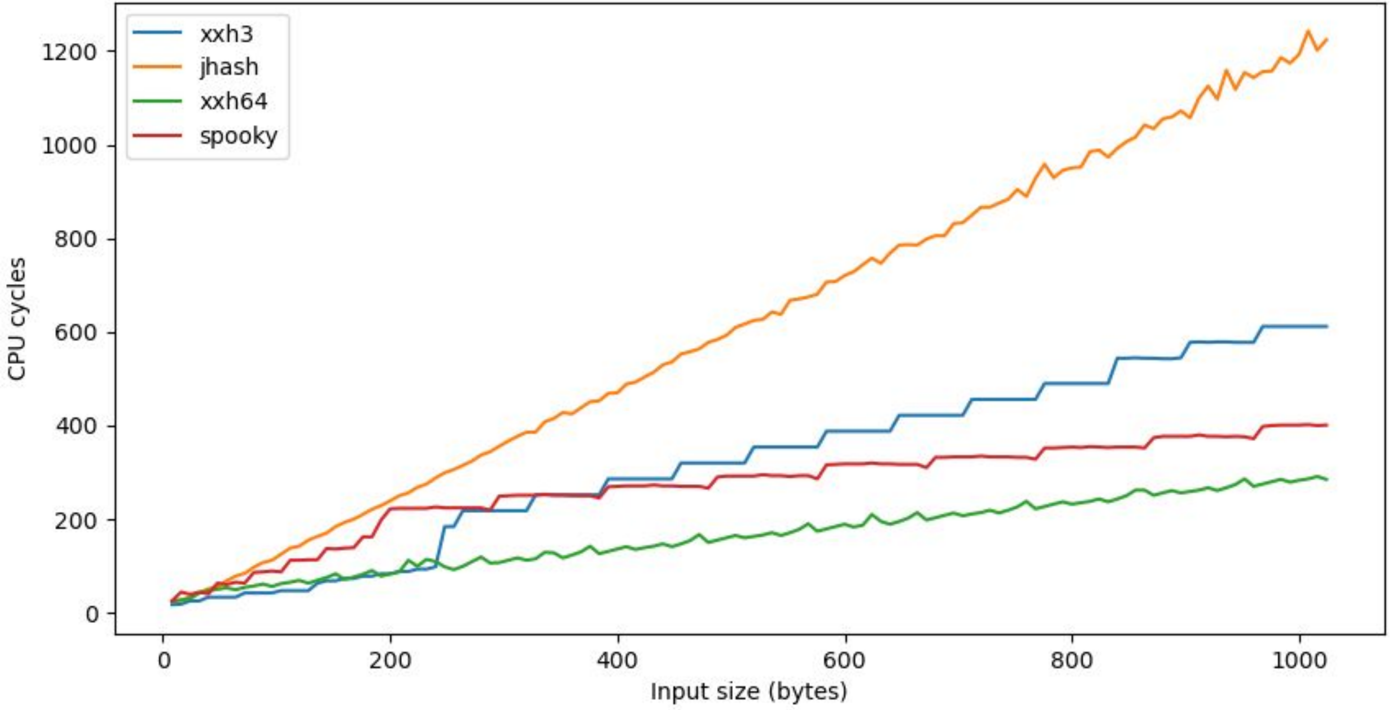
A better hash function for maps

- I’ve run some benchmarks on using different hash function for BPF maps, see a corresponding talk at fosdem 2023 with more details and for links
- The resume: for hashmap and bloom filter jhash2 is the best for small keys
- For keys >= 32 (or so, may differ for non-x86) xxh3 works way better
- Keys 16-32 are somewhere in between (may degrade for particular key sizes on some architectures for big and/or full maps)

A better hash function for maps

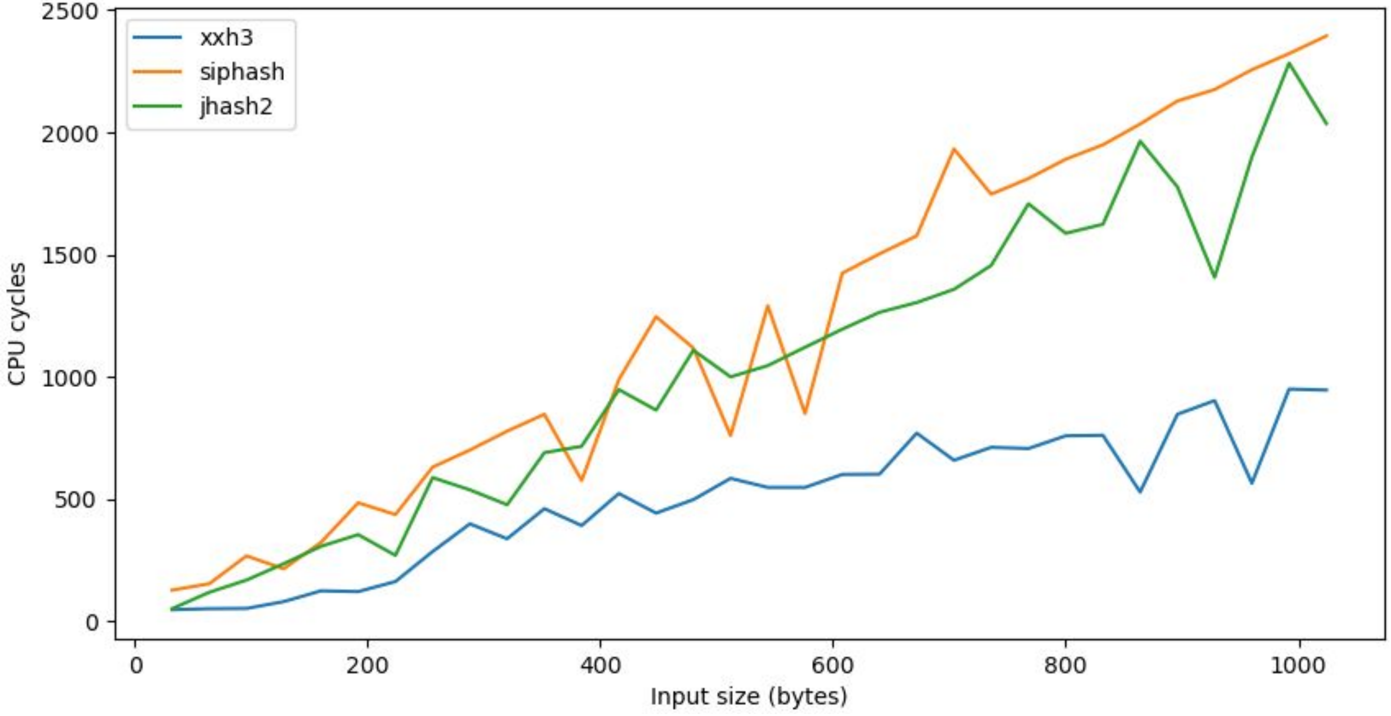


A better hash function for maps

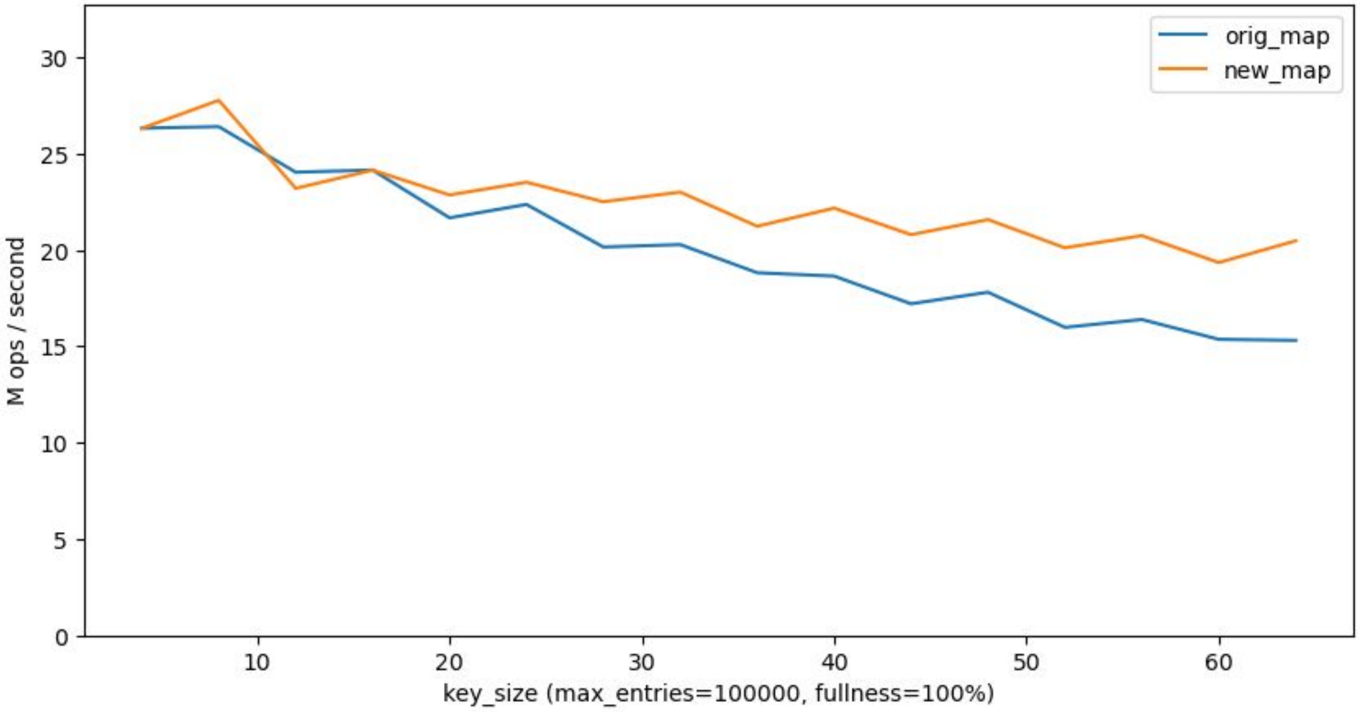


* Spooky will actually outperform xxh64 at about 8K, too far to be interesting for BPF maps

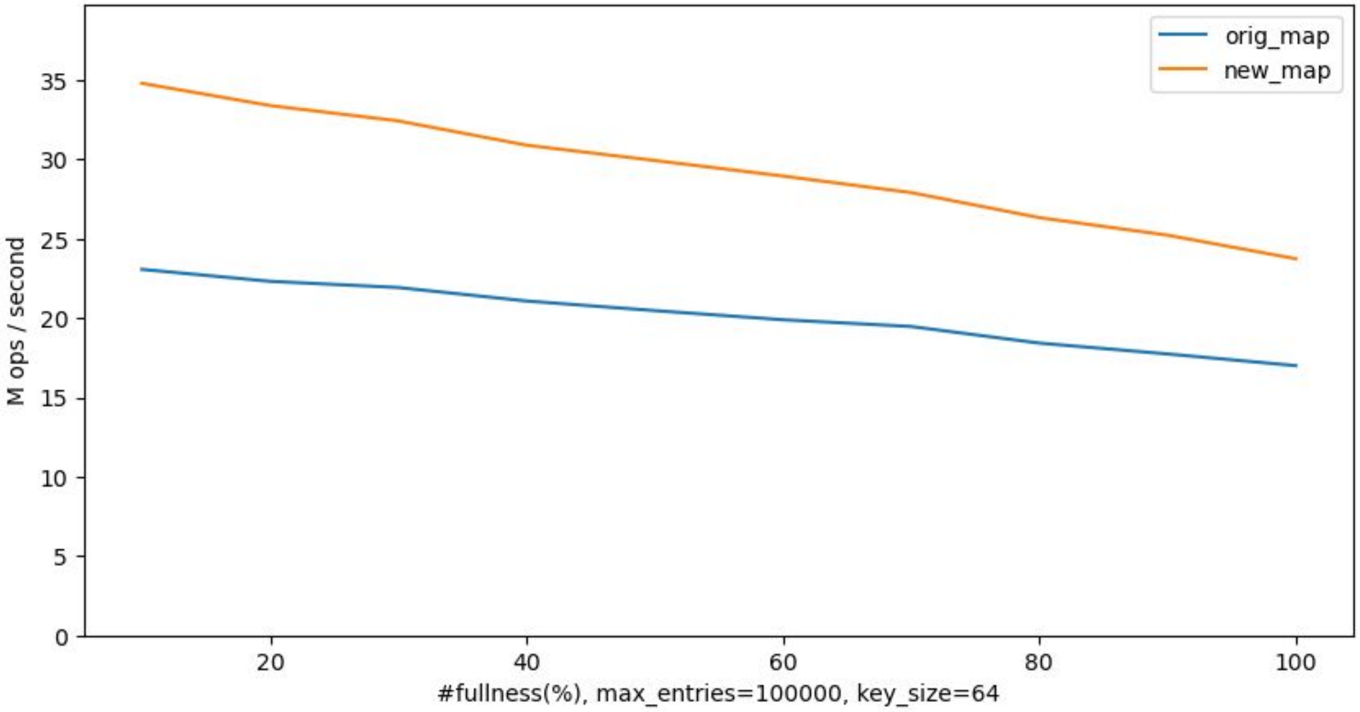
Siphash is actually comparable to jhash



Hashmap (max_entries=100K, 100% full, Ryzen 9)



Hashmap: 100K, key_size=64



Better hash for hashmap/bloom

```
static inline u32
htab_map_hash(const void *key, u32 key_len, u32 hashrnd)
{
        if (likely(key_len % 4 == 0 && key_len < 32))
                return jhash2(key, key_len / 4, hashrnd);
        return xxh3(key, key_len, hashrnd);
}
```

- What is left before posting a patch is to run benchmarks for different architectures to find if 32 is actually ok [and how much it affects keys <32]
- The actual patch may differ due to the fact that xxh3 degrades at key_len>240

xxh3 for stacktrace

- Stacktrace map doesn’t work [much] faster with a faster hash function (due to get_perf_callchain()
- However, it also doesn’t run slower, as stacktrace keys are 8 x stack depth long and this is typically > 32 where xxh3 is faster than jhash2
- What’s is better, we are interested in collisions even more than in speed optimizations

Experiment

- Take stacktrace and replace hash by either jhash2, xxh3, siphash
- Create three maps at the same time:
```
sudo bpftrace -e 'profile:hz:257 { @[kstack] = count(); }' &
sudo bpftrace -e 'profile:hz:257 { @[kstack] = count(); }' &
sudo bpftrace -e 'profile:hz:257 { @[kstack] = count(); }' &
```
- Run some noise, like
```
while true; do
    stress-ng --all 16 --timeout 2s
    sleep 20
done
```
- Kill all: `sudo pkill -9 bpftrace`
- `repeat`

Results: jhash is ok (but slower)

- About 5 minutes results (bpftrace creates a map of 2^17 entries, so 5 minutes was time to populate about 50K buckets with my load (~2.5M events = 32 CPU*257*300 sec))
- All hashes give about **1%** of collisions for half-full map (22K / 2.7M events)

	jhash2	siphash	xxh3
% collisions	0.85	0.88	0.86
% collisions	0.84	0.84	0.85
% collisions	0.80	0.82	0.81
% collisions (mean)	**0.83**	**0.84**	**0.84**

Results: jhash is ok (but slower)

- About 25 minutes results (bpftrace creates a map of 2^17 entries, 25 minutes was time to populate about 95K buckets (80%) with my load (~12M events = 32 CPU*257*1400 sec))
- All hashes give about **1.3%** of collisions for 80%-full map (146K/11.5M events)

	jhash2	siphash	xxh3
% collisions	1.263	1.281	1.281
% collisions	1.262	1.266	1.277
% collisions (mean)	1.26	1.27	1.28

Results: jhash is ok (but slower)

- About 24h results, 100% buckets full
- All hashes give about **1.7%** of collisions (700M events/12M collisions)
- (I probably didn't have enough random events to generate more collisions)

	jhash2	siphash	xxh3
% collisions	**1.69**	**1.69**	**1.69**