A BPF Map for Online Packet Classification

Anton Protopopov,
Isovalent
Outline

- Online packet classification: algorithms
- New map: `BPF_MAP_TYPE_WILDCARD`
- Pictures, Numbers
Packet Classification

● A classifier: rules + packets
● A Rule: one or more of
  ○ bitmask/prefix, e.g., 127.0.0.0/8
  ○ Range, e.g., 1-1024
● Fast lookups
Online Packet Classification

- Fast lookups
- And also fast updates
Use cases

- SDNs, Firewalls, ACLs, Routing, etc.
- Cilium: XDP prefilter
- Cilium: L4LB packet recorder
- Cilium: Network Policies
Cilium: XDP prefilter

- Allows to filter packets at the earliest time
- Limited functionality:
  - only lookups by source IP (hash)
  - or by source CIDR (LPM)
Cilium: L4LB packet recorder

- “Wildcard” 4-tuples, but
- Updates may require recompilation
- Doesn’t scale well (linear by #masks)
- Doesn’t support port ranges
- Added BPF complexity!
- See [LPC2021 talk](#) by Daniel & Martynas
Cilium: network policies

● K8S 1.25: Promoted `endPort` in Network Policy to Stable

● We can implement ranges using LPM, but:
  ○ No real ranges (only prefixes)
  ○ No support for both src & dst ranges
  ○ LPM is slower than hash
New Map Design

- Supports different wildcard rules and any number of fields
- Complexity: just a map lookup/update
- Fast lookups
- [reasonably] fast updates
Existing Algorithms

- **Brute Force**: actually works when there are just a few rules
- **Hash-based**: TSS (Tuple Space Search) => TM (Tuple Merge)
- **Tree-based**: Partition Sort

TupleMerge: Fast Software Packet Processing for Online Packet Classification
A Sorted Partitioning Approach to High-speed and Fast-update OpenFlow Classification
Existing Algorithms

* The picture is copied from "TupleMerge: Fast Software Packet Processing for Online Packet Classification"
Tuple Space Search

Say, we have rules of form IP/prefix

E.g., we have the following set of rules:

- 172.16.0.0/16
- 172.17.0.0/16
- 8.0.0.0/8
- 10.1.1.0/24
- 10.2.2.0/24
Tuple Space Search

- We can combine then as follows:
  - T(16): 172.16.0.0, 172.17.0.0
  - T(8): 8.0.0.0
  - T(24): 10.1.1.0, 10.2.2.0
Tuple Space Search

- Packet arrives from 10.2.2.2
  - T(16): 172.16.0.0, 172.17.0.0
  - T(8): 8.0.0.0
  - T(24): 10.1.1.0, 10.2.2.0
Tuple Space Search

- Table-1 lookup: 10.2.2.2 & ffff0000
  - T(16): 172.16.0.0, 172.17.0.0 10.2.0.0
  - T(8): 8.0.0.0
  - T(24): 10.1.1.0, 10.2.2.0
Tuple Space Search

- Table-2 lookup: 10.2.2.2 & ff000000
  - T(16): 172.16.0.0, 172.17.0.0
  - T(8): 8.0.0.0 10.0.0.0
  - T(24): 10.1.1.0, 10.2.2.0
Tuple Space Search

- Table-3 lookup: 10.2.2.2 & fffffff00
  - T(16): 172.16.0.0, 172.17.0.0
  - T(8): 8.0.0.0
  - T(24): 10.1.1.0, 10.2.2.0
Problem with Tuple Space Search

- Each successful lookup requires to search N/2 tables => N/2 hash table lookups
- Each unsuccessful lookup requires to search all N tables
Problems with Tuple Space Search, IPv4

# of tables for TSS for two-field rule: source and destination CIDR
100, ..., 25600 random rules, prefixes are chosen from [8,24], TSS caps at 289=17^2
Problems with Tuple Space Search, IPv6

# of tables for TSS for two-field rule: source and destination CIDR
100, ..., 102400 random rules, prefixes are chosen from [32,96], TSS caps at 4225=65^2
Tuple Merge to save the day

- **Idea 1:** group rules not by exact prefix match, but if \texttt{rule->prefix} \texttt{\geq} \texttt{table->prefix}

  - Example: table T(16) fits
    - 192.168.0.0/16 and 192.168.128.0/17 and 10.1.1.0/24 etc.
    - Doesn’t match, say, 127.0.0.1/8

* See the whitepaper for more details: [https://nonsns.github.io/paper/rossi19ton.pdf](https://nonsns.github.io/paper/rossi19ton.pdf)
Tuple Merge to save the day, IPv4

# of tables for two-field rule: source and destination CIDR, TSS (orange), TM (blue)
100, ..., 25600 random rules, prefixes are chosen from [8,24], TSS caps at 289=17^2
Tuple Merge to save the day, IPv6

# of tables for two-field rule: source and destination CIDR, TSS (orange), TM (blue)
100, ..., 102400 random rules, prefixes are chosen from [32,96], TSS caps at 4225=65^2
Tuple Merge to save the day

- **Idea 2:** create new tables based on trimmed masks
- **Example:**
  - Got new rule 192.168.0.0/16
  - Create new table /14 (= 16 - 16/8)
  - Next rule 172.17.0.0/15 still fits

* See the whitepaper for more details: [https://nonsns.github.io/paper/rossi19ton.pdf](https://nonsns.github.io/paper/rossi19ton.pdf)
Tuple Merge to save the day, IPv4

# of tables for TM with untrimmed tables (orange), TM (blue)
100, ..., 25600 random rules, prefixes are chosen from [8,24]
Tuple Merge to save the day, IPv6

# of tables for TM with untrimmed tables (orange), TM (blue)

100, ..., 102400 random rules, prefixes are chosen from [32,96]
Tuple Merge to save the day

- **Idea 3** (not implemented for the map): if we have table \((x, y)\) where \(y \ll x\), then convert it to \((x, 0)\), i.e., ignore \(y\)

- **Example:**
  - Rule: 192.168.0.0/16, 192.168.2.0/28
  - New table \((14, 0)\)

* See the whitepaper for more details: [https://nonsns.github.io/paper/rossi19ton.pdf](https://nonsns.github.io/paper/rossi19ton.pdf)
Some Problems with Tuple Merge

- We can’t cap the number of tables
- The number of tables depends on the order in which rules appear, e.g.:
  - 10.0.0.0/8, 192.168.0.0/16 => $T(7)$
  - 192.168.0.0/16, 10.0.0.0/8 => $T(14), T(7)$
Static TM to save the day

- We can preallocate tables based on prior knowledge of rules structure
- E.g., for IPv6 4-tuples:
  - \((32, 32, 0, 0), (32, 0, 0, 0), (0, 32, 0, 0)\)
- We will need only 1 hash computation\(^*\), and we still have 64 bits of randomness

\(^*\) if no fields are ignored; e.g., a table \((32, 0, 0, 0)\) will be used if there are rules of form \((ip/prefix, *, src-port-range, dst-port-range)\)
Problems with Static TM day

- Cast in Stone. Say, we’ve created a map with one table: (32,32,0,0), then we can’t ignore fields or add shorter
- Error-prone and bug-report-prone (users for sure will shoot themselves in the foot with this interface)
So, which algorithm to use?
So, which algorithm to use?

- Brute force:
  
  BPF_WILDCARD_F_ALGORITHM_BF
So, which algorithm to use?

- Brute force: 
  BPF_WILDCARD_F_ALGORITHM_BF
- Tuple Merge: 
  BPF_WILDCARD_F_ALGORITHM_TM
So, which algorithm to use?

- Brute force:
  \texttt{BPF\_WILDCARD\_F\_ALGORITHM\_BF}

- Tuple Merge:
  \texttt{BPF\_WILDCARD\_F\_ALGORITHM\_TM}

- Static Tuple Merge:
  \texttt{BPF\_WILDCARD\_F\_ALGORITHM\_TM} | \texttt{BPF\_WILDCARD\_F\_TM\_STATIC\_POOL}
So, which algorithm to use?

- Just choose the default algorithm
- Maybe provide some flags
Example: 4-tuple, wildcard

Four Fields:

- Source IP/Prefix: (10.3.4.0/24)
- Destination IP/Prefix: (192.168.0.0/16)
- Source Port Range: (*)
- Destination Port Range: (1-1024)
Allocate a map

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);

struct {
    __uint(type, BPF_MAP_TYPE_WILDCARD);
    __type(key, struct capture4_wcard_key);
    __type(value, __u64);
    __uint(max_entries, 100000);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(map_extra, BPF_WILDCARD_F_ALGORITHM_TM);
    __type(wildcard_desc, struct capture4_wcard_desc);
} filter_v4_tm_dynamic __section(".maps");
Allocate a map

```c
BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);

struct {
    __uint(type, BPF_MAP_TYPE_WILDCARD);
    __type(key, struct capture4_wcard_key);
    __type(value, __u64);
    __uint(max_entries, 100000);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(map_extra, BPF_WILDCARD_F_ALGORITHM_TM);
    __type(wildcard_desc, struct capture4_wcard_desc);
} filter_v4_tm_dynamic __section(".maps");
```
Insert rules (userspace)

```c
struct capture4_wcard_key rule = {
    .type = BPF_WILDCARD_KEY_RULE,
    .rule = {
        .saddr = pton("10.3.4.0"),
        .saddr_prefix = 24,
        .daddr = pton("192.168.0.0"),
        .daddr_prefix = 16,
        .sport_min = 0,
        .sport_max = 0xffff,
        .dport_min = 1,
        .dport_max = 1024,
    },
};

bpf_map_update_elem(fd, &rule, &val, 0);
```
Match packets (kernel space)

```c
struct capture4_wcard_key key = {};

// ... set up struct iphdr *ip4 and L4 *l4 ...

key.type = BPF_WILDCARD_KEY_ELEM;
key.saddr = ip4->saddr;
key.daddr = ip4->daddr;
memcpy(&key.sport, l4, 4); /* copy both ports */

bpf_map_lookup_elem(&map, &key);
```
Define the rule structure

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);
Define the rule structure

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);
Define the rule structure

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);
Define the rule structure

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);
Define the rule structure

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);
Allocate a map

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);

struct {
    __uint(type, BPF_MAP_TYPE_WILDCARD);
    __type(key, struct capture4_wcard_key);
    __type(value, __u64);
    __uint(max_entries, 100000);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(map_extra, BPF_WILDCARD_F_ALGORITHM_TM);
    __type(wildcard_desc, struct capture4_wcard_desc);
} filter_v4_tm_dynamic __section(".maps");
Map allocation

```c
struct capture4_wcard_key {
    __u32 type;
    union {
        struct { /* rule */ };
        struct { /* packet */ };
    };
};

type is {BPF_WILDCARD_KEY_RULE,BPF_WILDCARD_KEY_ELEM}
```
Map allocation

```c
struct capture4_wcard_key {
    __u32 type;
    union {
        struct { /* rule */ };  
        struct { /* packet */ };  
    };
};

type is {BPF_WILDCARD_KEY_RULE,BPF_WILDCARD_KEY_ELEM}
```
Map allocation

```c
struct capture4_wcard_key {
    __u32 type;
    union {
        struct { /* rule */ };
        struct { /* packet */ };
    };
}

type is {BPF_WILDCARD_KEY_RULE, BPF_WILDCARD_KEY_ELEM}
```
Map allocation

Rule:

```c
struct {
    __u32 saddr;
    __u32 saddr_prefix;
    __u32 daddr;
    __u32 daddr_prefix;
    __u16 sport;
    __u16 sport_min;
    __u16 sport_max;
    __u16 dport_min;
    __u16 dport_max;
};
```

Packet:

```c
struct {
    __u32 saddr;
    __u32 daddr;
    __u16 sport;
    __u16 dport;
};
```
Map allocation

Rule:

```c
struct {
    __u32 saddr;
    __u32 saddr_prefix;
    __u32 daddr;
    __u32 daddr_prefix;
    __u16 sport;
    __u16 sport_min;
    __u16 sport_max;
    __u16 dport;
    __u16 dport_min;
    __u16 dport_max;
};
```

Packet:

```c
struct {
    __u32 saddr;
    __u32 daddr;
    __u16 sport;
    __u16 dport;
};
```
Map allocation

Rule:

```c
struct {
    __u32 saddr;
    __u32 saddr_prefix;
    __u32 daddr;
    __u32 daddr_prefix;
    __u16 sport;
    __u16 sport_min;
    __u16 sport_max;
    __u16 dport_min;
    __u16 dport_max;
};
```

Packet:

```c
struct {
    __u32 saddr;
    __u32 daddr;
    __u16 sport;
    __u16 dport;
};
```
Map allocation

Rule:

```c
struct {
    __u32  saddr;
    __u32  saddr_prefix;
    __u32  daddr;
    __u32  daddr_prefix;
    __u16  sport;
    __u16  sport_min;
    __u16  sport_max;
    __u16  dport;
    __u16  dport_min;
    __u16  dport_max;
};
```

Packet:

```c
struct {
    __u32  saddr;
    __u32  daddr;
    __u16  sport;
    __u16  dport;
};
```
Map allocation

Rule:

```c
struct {
    __u32 saddr;
    __u32 saddr_prefix;
    __u32 daddr;
    __u32 daddr_prefix;
    __u16 sport;
    __u16 sport_min;
    __u16 sport_max;
    __u16 dport;
    __u16 dport_min;
    __u16 dport_max;
};
```

Packet:

```c
struct {
    __u32 saddr;
    __u32 daddr;
    __u16 sport;
    __u16 dport;
};
```
Map allocation, continued

- Users now know that this is a 4-tuple wildcard map
- But kernel will only see
  
  ```c
  void *key
  map->key_size
  ```
Map allocation, continued

Tell kernel about the map structure:

```c
struct wildcard_desc {
    __u32 n_rules;
    struct wildcard_rule_desc rule_desc[];
};

struct wildcard_rule_desc {
    __u32 type; /* WILDCARD_RULE_{PREFIX,RANGE,MATCH} */
    __u32 size; /* the size of the field in bytes */
    ...  
};
```
Map allocation, continued

Pass the following in bpf_attr:

```c
struct wildcard_desc desc = {
    .n_rules = 4,
    .rule_desc = {
        { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
        { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
        { .type = BPF_WILDCARD_RULE_RANGE, .size = 2, },
        { .type = BPF_WILDCARD_RULE_RANGE, .size = 2, },
    }
};
```
Map allocation: here comes BTF

- We can’t just specify a `struct wildcard_desc` in a BTF map definition

- So, another structure is parsed by libbpf and converted to `wildcard_desc`
struct capture4_wcard_desc {
    __uint(n_rules, 4);
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } saddr;
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } daddr;
    struct {
        __uint(type, BPF_WILDCARD_RULE_RANGE);
        __uint(size, sizeof(__u16));
    } sport;
    struct {
        __uint(type, BPF_WILDCARD_RULE_RANGE);
        __uint(size, sizeof(__u16));
    } dport;
};
struct capture4_wcard_desc {
    __uint(n_rules, 4);
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } saddr;
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } daddr;
    ...

    struct wildcard_desc desc = {
        .n_rules = 4,
        .rule_desc = {
            { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
            { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
            ...
    }
}
struct capture4_wcard_desc {
    __uint(n_rules, 4);
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } saddr;
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } daddr;
    ...

    struct wildcard_desc desc = {
        .n_rules = 4,
        .rule_desc = {
            .type = BPF_WILDCARD_RULE_PREFIX, .size = 4,
            ...
        }
    };
}
Map allocation, continued

```c
struct capture4_wcard_desc {
    __uint(n_rules, 4);
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } saddr;
    struct {
        __uint(type, BPF_WILDCARD_RULE_PREFIX);
        __uint(size, sizeof(__u32));
    } daddr;
    ...

    struct wildcard_desc desc = {
        .n_rules = 4,
        .rule_desc = {
            { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
            { .type = BPF_WILDCARD_RULE_PREFIX, .size = 4, },
            ...
        }
    }
```
Allocating maps is actually quite simple

BPF_WILDCARD_DESC_4(
    capture4_wcard,
    BPF_WILDCARD_RULE_PREFIX, __u32, saddr,
    BPF_WILDCARD_RULE_PREFIX, __u32, daddr,
    BPF_WILDCARD_RULE_RANGE, __u16, sport,
    BPF_WILDCARD_RULE_RANGE, __u16, dport
);

struct {
    __uint(type, BPF_MAP_TYPE_WILDCARD);
    __type(key, struct capture4_wcard_key);
    __type(value, __u64);
    __uint(max_entries, 100000);
    __uint(map_flags, BPF_F_NO_PREALLOC);
    __uint(map_extra, BPF_WILDCARD_F_ALGORITHM_TM);
    __type(wildcard_desc, struct capture4_wcard_desc);
} filter_v4_tm_dynamic __section(".maps");
Kernel Changes Required

- **bpf_attr**: new fields, code to copy them:
  - `void *map_extra_data`
  - `u32 map_extra_data_size`
- Patch libbpf to parse new BTF definitions and pass data all the way to `bpf(2)`
Numbers: IPv4 4-tuple, TM, Static TM

successful lookups, in nanoseconds, 100, ..., 102400 random rules, prefixes are chosen from [8,24]
dyn: generic Tuple Merge; 4: Static TM with tables=[(16,16),(16,8),(8,16),(8,8)]; 2: Static TM with tables=[(16,16),(8,8)]
Numbers: IPv4 4-tuple, TM, Static TM

unsuccessful lookups, in nanoseconds, 100, …, 102400 random rules, prefixes are chosen from [8,24]
dyn: generic Tuple Merge; 4: Static TM with tables=\([(16,16),(16,8),(8,16),(8,8)]\); 2: Static TM with tables=\([(16,16),(8,8)]\)
Numbers: IPv6 4-tuple, TM, Static TM

successful lookups, in nanoseconds, 100, ..., 102400 random rules, prefixes are chosen from [8,24]
dyn: generic Tuple Merge; static: Static TM with tables=[(32,32)]
Numbers: IPv6 4-tuple, TM, Static TM

unsuccessful lookups, in nanoseconds, 100, ..., 102400 random rules, prefixes are chosen from [8,24] dyn: generic Tuple Merge; static: Static TM with tables=[(32,32)]
Successful lookups in nanoseconds, tm: Tuple Merge, trie: LPM trie
100,..., 64000 random IPv6 CIDRs, prefixes are random from /32 to /96
Numbers: TM, LPM

updates in nanoseconds, tm: Tuple Merge, trie: LPM trie
100,..., 64000 random IPv6 CIDRs, prefixes are random from /32 to /96
Existing Algorithms

* The picture is copied from "TupleMerge: Fast Software Packet Processing for Online Packet Classification"
Existing Algorithms, updated

* The picture is copied from “TupleMerge: Fast Software Packet Processing for Online Packet Classification”
Thanks! Questions, feedback, use cases?

- POC Code: [RFC patch]
- Please share your use cases!