High Performance Programmable Parsers

Common Parser Language,kParser, kParser+XDP

Tom Herbert - SiPanda
Pratyush Khan - SiPanda
Aravind Buduri - Ventana

Netdev 0x16

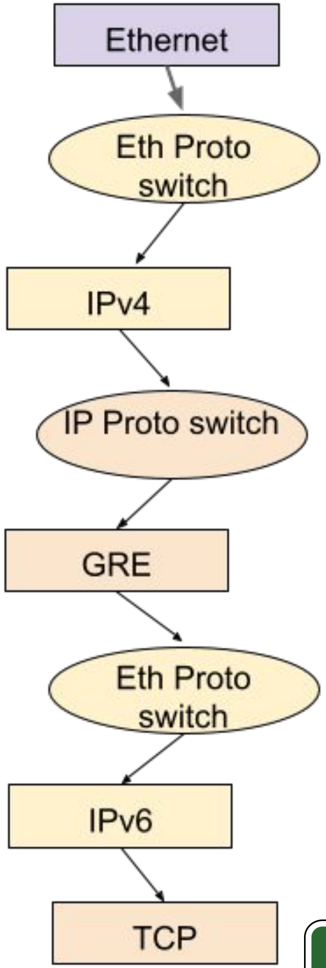
Agenda

- Protocol parsing fundamentals
- Common Parser Language (.json)
- kParser and CLI
- XDP + kParser (XDP + flow dissector)
- Upstream efforts, and futures

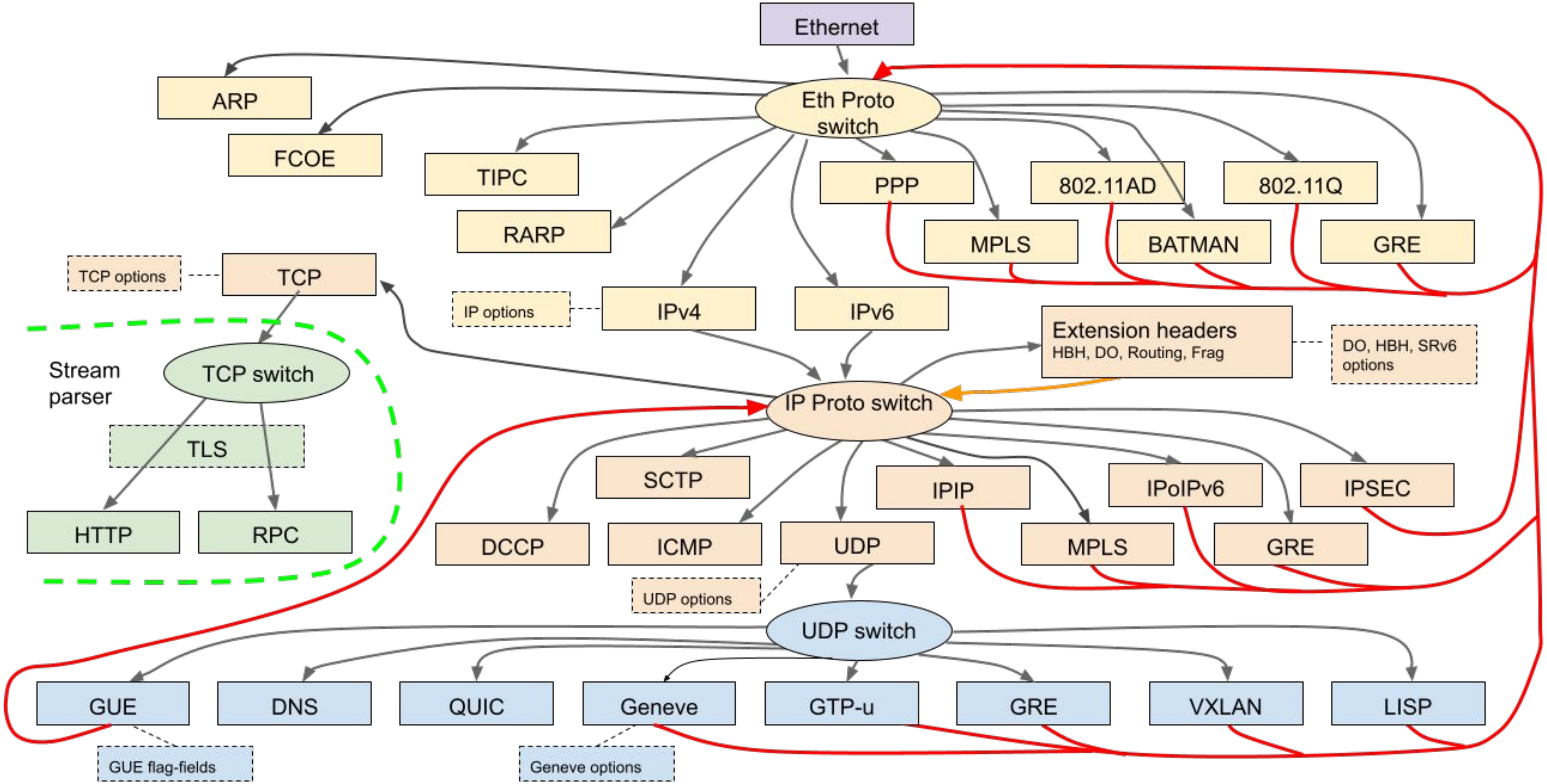
Parsing and parser fundamentals

About and why it’s important

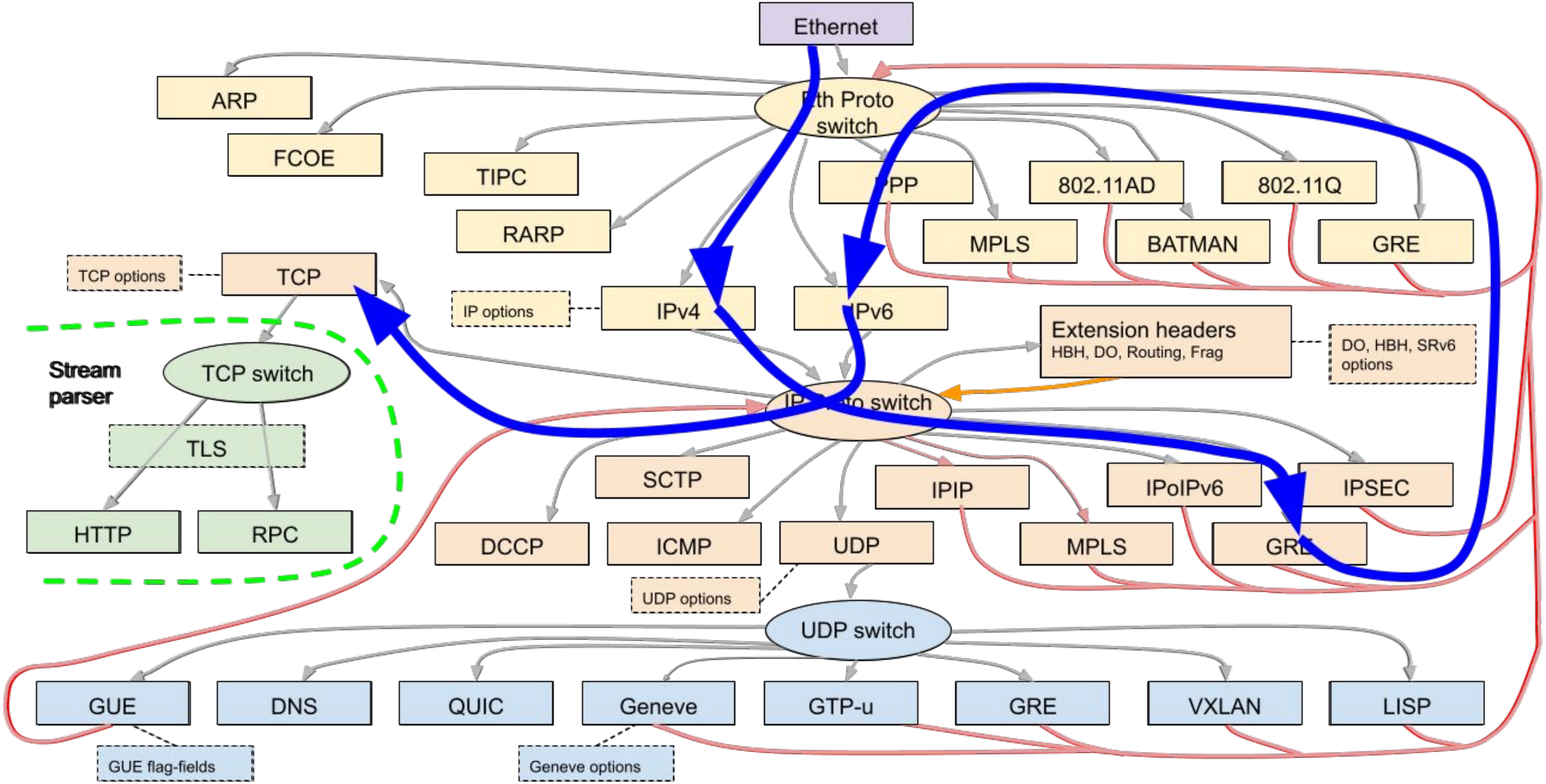
- **Protocol parsing** is to analyze a packet or PDU to identify it’s various protocol headers and layers per the rules of the protocol definitions
- The set of parsable protocols constitutes a parse graph. Processing one packet is a “**walk in the parse graph**”
- A **parser** parses the protocols for some supported parse graph. Implementation-wise it’s a type of Finite State Machine
- Protocol parsing is an inherently **serialized process**, it also one of the **most common functions** in a networking stack



Example parse graph



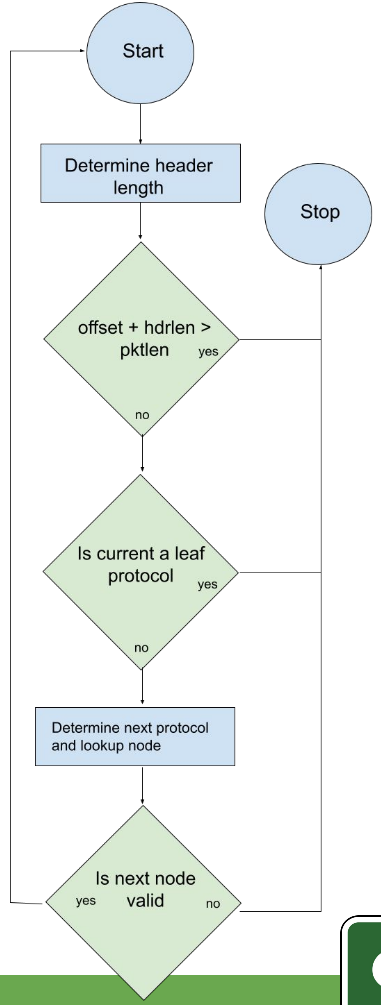
Example walk:Eth->IPv4->GRE->IPv6->TCP



Mechanics of a parser

- Parser maintains current header cursor with a byte length
- Transitions in the FSM are made based on looking up next protocol header in a table

- *What is the length of the current protocol header?*
  - Fixed length (e.g. IPv6)
  - Variable length from header field (e.g TCP)
  - Length from flag fields (e.g. GRE)
- *Does derived header length exceed bounds?*
- *What is the type of the next header?*
  - No next header for leaf protocols (e.g TCP)
  - Fixed value (e.g.IPIP)
  - Next protocol field (e.g. EtherType in Ethernet header)



Additional functions

- Metadata collection and extraction
  - Metadata: any information the parser reports about a packet
  - E.g. protocol field values, header offsets, timestamps, header lengths
  - The parser can places metadata into a structure for later consumption
- Per layer handlers
  - Deeper processing may invoked from the parser to process a protocol layer (e.g. a TCP function may be run to validate the TCP header)
  - The function has access to its protocol layer and all metadata reported to that point

Common Parser Language

Parser representation

Goal: *A common, flexible, and abstracted method to represent parsers*

Motivation:

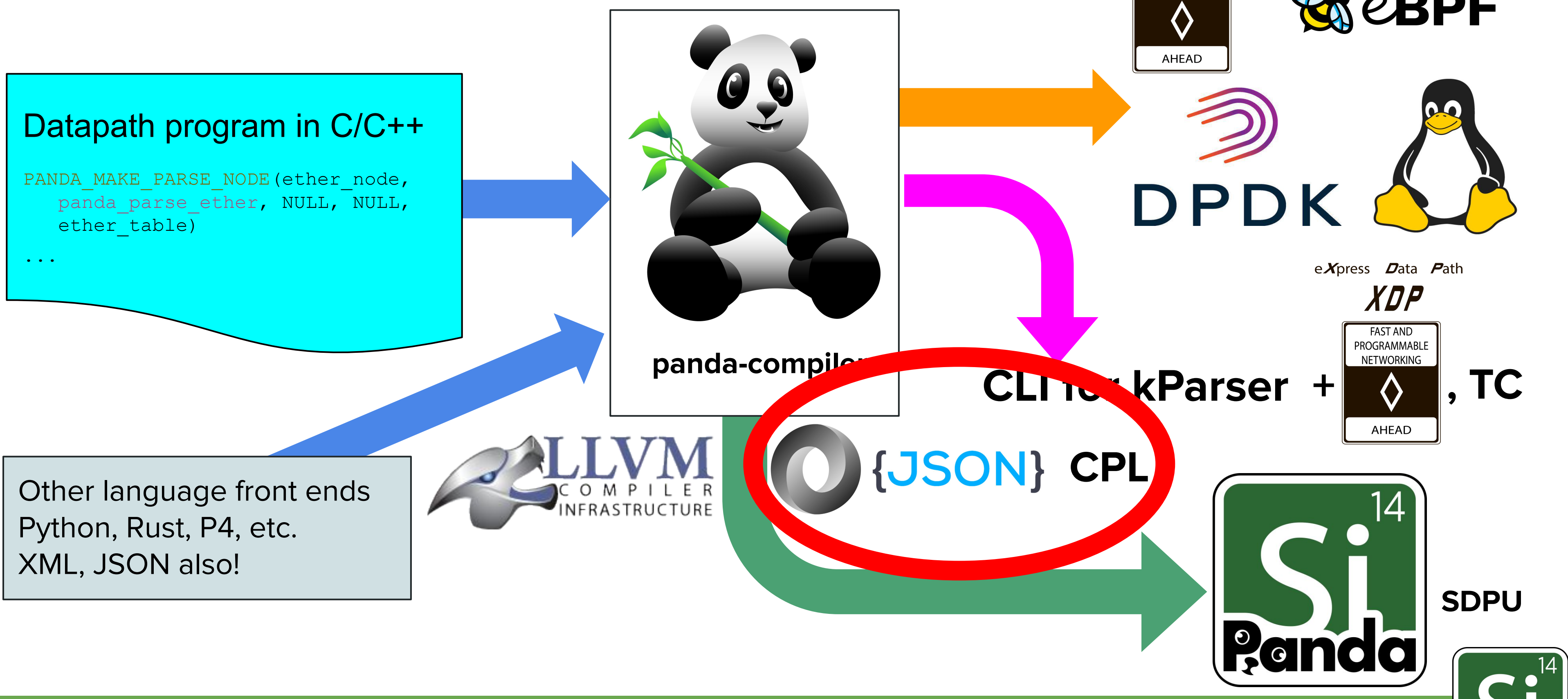
- Parsers are best represented in declarative, not imperative, representation
- Desire a common Intermediate Representation of parsers in compilers
- Help address a major problem in parser offload (more on that later)

Solution: Common Parser Language (CPL)

- Parser representation in declarative .json
- Objects: parsers, parse nodes, protocol tables, metadata rules, etc.
- Support to represent nearly all protocols and protocol constructs
- At the highest level, the .json *looks* like the parse graph I just showed
- .json is very easy to work with in Python, GUI's (e.g. kParser json2ip.py script)
- Machine readable to be a formal Intermediate Representation (IR)

CPL {JSON}

CPL in the parser ecosystem



Objects

- Parsers
  - Root parse node and attributes
- Parse nodes
  - Rules to determine header length and next header
  - Reference to next header table
  - Metadata to extract
- Protocol tables
  - Map protocol number to next parse node
- Metadata rules and lists
  - Definitions for metadata extraction

Parameterized functions

- Functions for parsing in CPL .json are represented as parameterized functions
- For example, parameterize function to compute variable header length is **F**(*field_offset*, *field_len*, *field_mask*, *endianswap*, *multiplier*, *add_length*)

hdr_bytes = (Load_bytes(**cur_hdr** + *len_field_offset*, *len_field_bytes*) & *field_mask*)
>> shfit_from_mask(*field_mask*)
hdr_bytes = *endianswap* ? swap_bytes(hdr_bytes, *len_field_bytes*) : hdr_bytes
hdr_bytes = (*multiplier* * hdr_bytes) + *add_length*

- So the function to compute IPv4 header length is **F**(0, 1, 0x0f, false, 4, 0)

Features in CPL (these map to kParser as well)

- Handler functions
- TLV parsing
- Flag-fields parsing
- Encapsulation
- Overlay nodes
- Counters
- Conditional expressions
- Limits

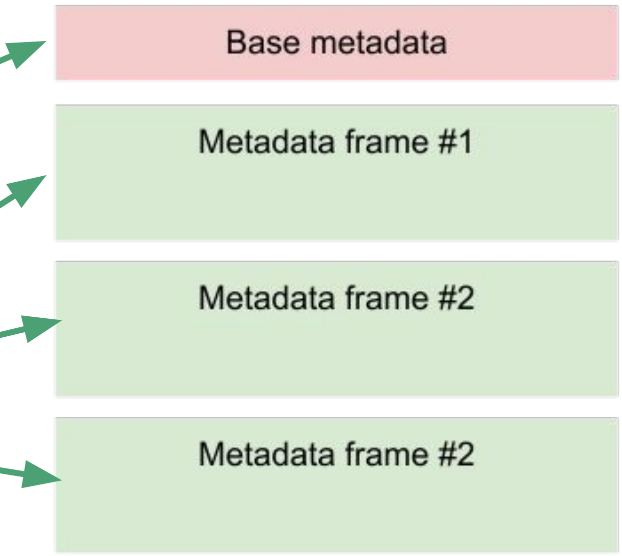
Metadata

- Metadata is written to a medata block defined by the user. Typically, this is overlaid by the data structure defined by the user
- Two sub-data structures in metadata
    - **Base metadata**: Metadata fields that are common to all protocol layers
    - **Metadata frames**: Array of structures containing fields for each encapsulation layer
- Metadata field is referred by (***isframe, md-off, length, encap-layer***)
    - *isframe*: boolean to indicate that the field is in a metadata frame
    - *md-off* : offset of the metadata field in the base metadata of the metadata frame
    - *length* is the metadata field size in bytes
    - *encap-layer* serves as the index into the metadata frames

Metadata example

C representation

```
struct base_metadata {
    unsigned short ip_offset;
    unsigned short tcp_offset;
};

struct metadata frame {
    in_addr src;
    in_addr dest;
    __u16 sport;
    __u16 dest;
};

struct metadata {
    struct base_metadata base;
    struct metadata_frame
        frames[5];
};
```



Accessing the source port in 2nd encap level (i.e. (**true**, 8, 2, 1)

In C code: `metadata->frames[1].sport`

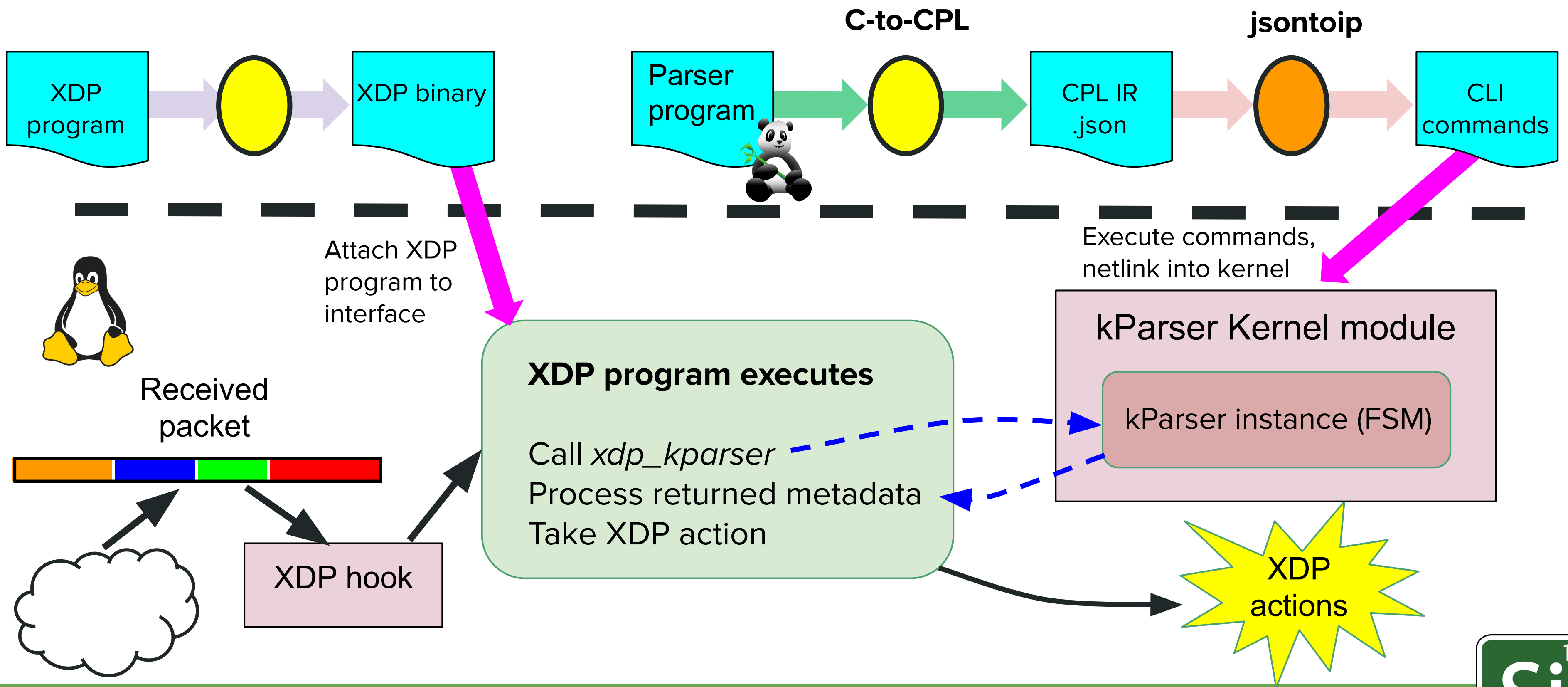
CPL .json and CLI in later slide

kParser+CLI

kParser

- In-kernel programmable parser
- Based on open source PANDA parser
- Programmed via “*ip parser*” commands with netlink backend
- CPL->CLI and CLI->CPL helpers
- Segue to parser offload

kParser (CLI+XDP)



kParser kernel API

- kParser can be called from various places in the kernel
    - We’ll show use in XDP with an XDP helper function
    - P4TC is using kParser also
- Four API In-kernel programmable parser

```
int kparser_parse(struct sk_buff *skb, const struct kparser_hkey *kparser_key,
                  void *_metadata, size_t metadata_len);

int __kparser_parse(const void *parser, void *_hdr, size_t parse_len, void *_metadata,
                    size_t metadata_len);

const void *kparser_get_parser(const struct kparser_hkey *kparser_key);

int kparser_put_parser(const void *parser);
```

Example: Five tuple parser with header offsets

- Parse Ethernet/IPv4 to UDP and TCP
- Report the offsets of the IPv4 and TCP or UDP header
- Extract IPv4 addresses and UDP or TCP port numbers
- Call the parser from XDP and xdp_print returned values

PANDA-C code for tuple_parser

```
#include "panda/parser.h"
```

```
struct metadata {
      unsigned short ip_offset;
      unsigned short l4_offset;
      inaddr_t addrs[2];
      inaddr_t ports[2];
};
```

```
static void extract_ipv4(const void *_hdr, size_t hdr_len,
    size_t hdr_offset, void *_metadata,  void *_frame,
    const struct panda_ctrl_data *ctrl)
{
    ((struct metadata *)_metadata)->ip_offset = hdr_offset;
    memcpy(((struct metadata *)_metadata)->addrs,
           (__u8 *)&_hdr[12], 8);
}
```

```
static void extract_ports(const void *_hdr, size_t hdr_len,
    size_t hdr_offset, void *_metadata, void *_frame,
    const struct panda_ctrl_data *ctrl)
{
    ((struct metadata_base *)_metadata)->l4_offset =
        hdr_offset;
    memcpy(((struct metadata *)_metadata)->ports, _hdr, 4);

}
```

```
PANDA_MAKE_PARSE_NODE(ether_node, NULL, NULL, NULL, ether_table);
PANDA_MAKE_PARSE_NODE(ipv4_node, NULL, extract_ipv4, NULL,
                      ip_table);
PANDA_MAKE_LEAF_PARSE_NODE(ports_node, NULL, extract_ports,
                           NULL);

PANDA_MAKE_PROTO_TABLE(ether_table, { __cpu_to_be16(ETH_P_IP),
                       &ipv4_node });
PANDA_MAKE_PROTO_TABLE(ip_table, { IPPROTO_TCP, &ports_node },
    { IPPROTO_UDP, &ports_node } );

PANDA_PARSER(tuple_parser, "Tuple parser", &ether_node);
```

CPL .json for tuple_parser


```
{
  "parsers": [
    {
      "name": "tuple_parser",
      "root-node": "ether_node"
    }
  ],
  "parse-nodes": [
    {
      "name": "ether_node",
      "min-hdr-length": 14,
      "next-proto": {
        "field-off": 12,
        "table": "ether_table",
        "field-len": 2
      }
    },
```

```
{
    "name": "ipv4_node",
    "min-hdr-length": 20,
    "next-proto": {
      "field-off": 9,
      "table": "ip_table",
      "field-len": 1
      "field-off": 0,
      "mask": "0xf",
      "field-len": 1,
      "multiplier": 4 },
    "metadata": {
      "ents": [
        {
          "hdr-src-off": 12,
          "length": 8,
          "md-off": 4, },
        {
          "md-off": 0,
          "name": "ip_offset",
          "type": "offset"
  }}}],
```

```
{
  "name": "ports_node",
  "min-hdr-length": 4,
  "metadata": {
   "ents": [
      {
        "hdr-src-off": 0,
        "length": 4,
        "md-off": 12,
      },
        c-off": 0,
        length": 1,
        "md-off": 2,
        "type": "offset"
],
```

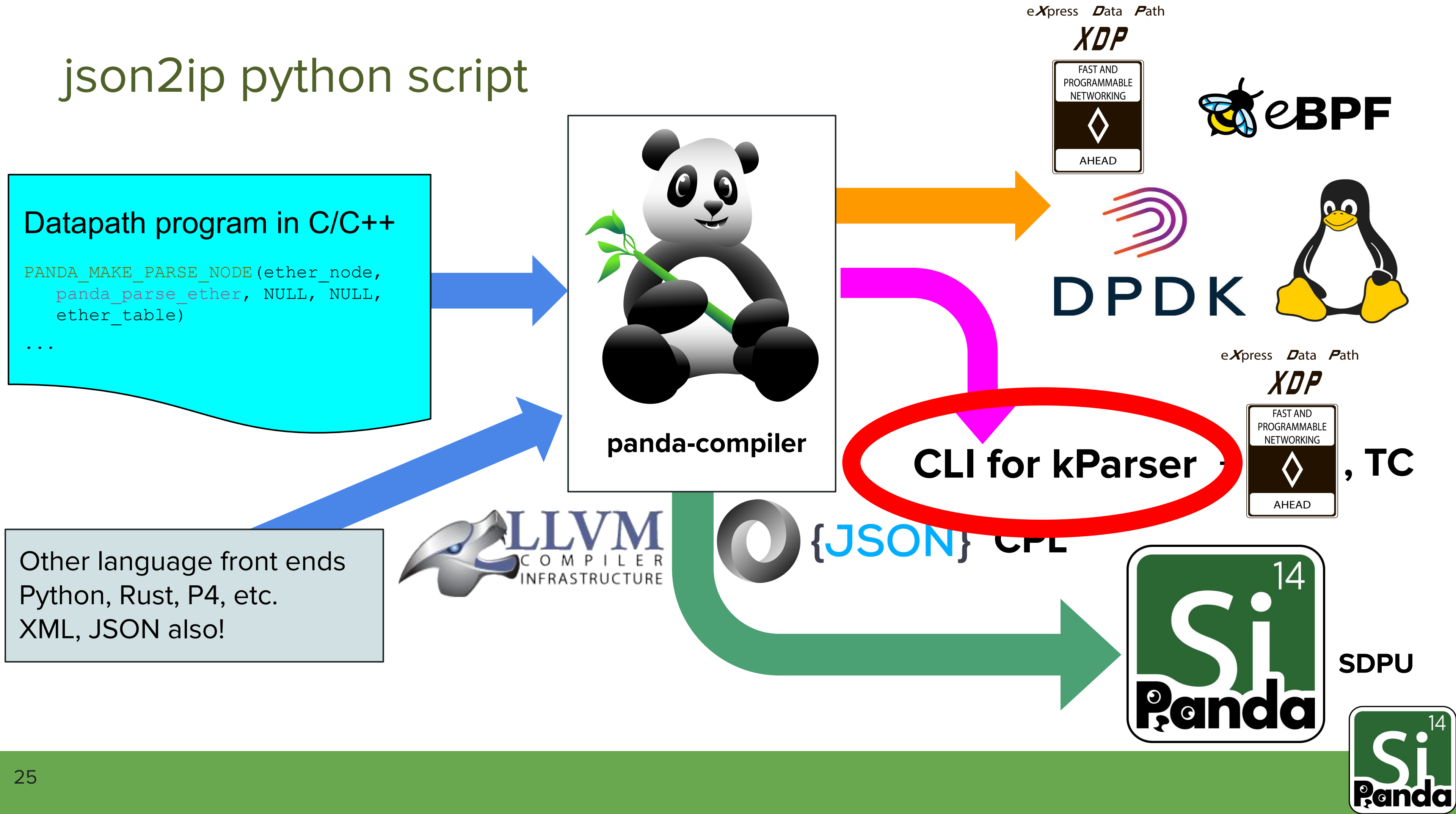
```
"proto-tables": [
  {
    "name": "ether_table",
    "ents": [
      {
        "name": "ipv4_node",
        "key": "0x800",
        "node": "ipv4_node"
      } ] },
  {
    "name": "ip_table",
    "ents": [
      {
        "name": "ports_node",
        "key": "0x6",
        "node": "ports_node"
      },
      {
        "name": "ports_node",
        "key": "0x11",
        "node": "ports_node"
      } ] } ]
}
```


Parser

Parse nodes

Protocol tables

json2ip python script



CLI commands for tuple_parser

```
ip parser create md-rule name md.iphdr_offset type offset md-off 0
ip parser create md-rule name md.ipaddrs src-hdr-off 12 length 8 md-off 4
ip parser create md-rule name md.l4_hdr.offset type offset md-off 2
ip parser create md-rule name md.ports src-hdr-off 0 length 4 md-off 12

ip parser create node name node.ether hdr.minlen 14 nxt.offset 12 nxt.length 2 \
    nxt.table-ent 0x800:node.ipv4

ip parser create node name node.ports hdr.minlen 4 md-rule md.l4.hdr_offset md-rule md.ports

ip pa          name node.ipv4 hdr.minlen 20 hdr.len-field-off 0 hdr.len-field-len 1 \
    h          k 0x0f hdr.len-field-multiplier 4       nxt.field-off nxt.field-len 1    \
    nxt.table-ent 6:node.ports nxt.table-ent 17:node.ports md-rule md.iphdr_offset \
    md-rule md.ipaddrs

ip parser create parser name tuple_parser rootnode node.ether
```

Metadata rules

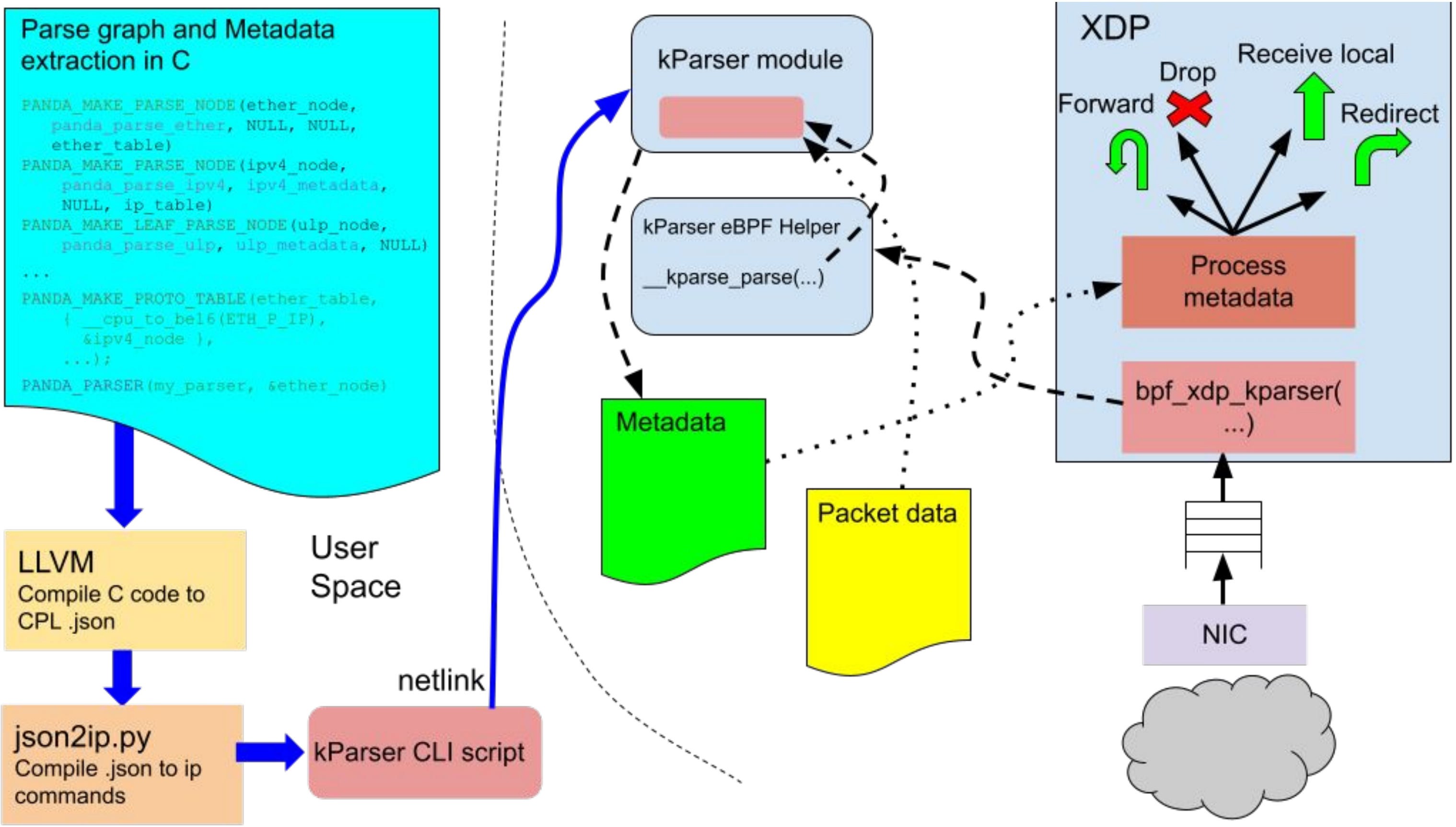
Parser

Parse nodes

kParser in an XDP helper

(and Flow Dissector helper)

kParser in XDP helper



tuple_parser XDP program

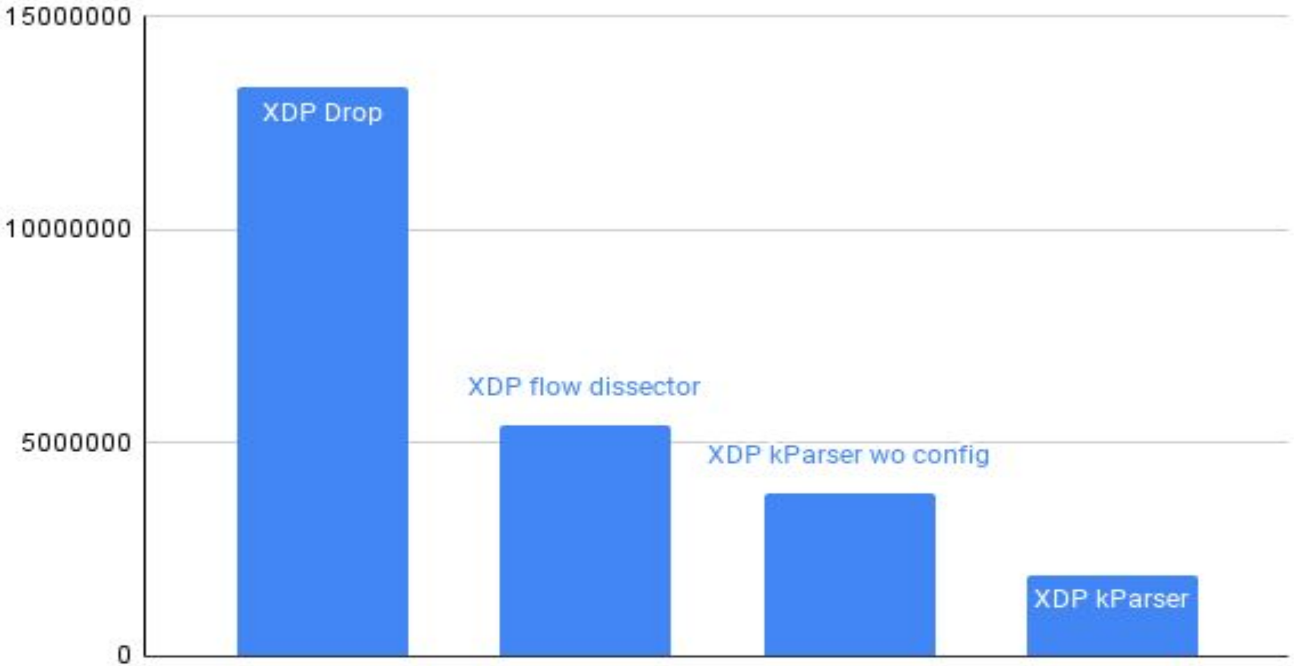
```
SEC("prog")
int xdp_parser_prog(struct xdp_md *ctx)
{
    struct metadata metadata;
    struct kparser_hkey *parser_key;
    key_config((char *)arr);
    memset(&metadata, 0, sizeof(metadata));
    xdp_kparser(ctx, &metadata, sizeof(metadata), &parser_key, sizeof(parser_keY));
    xdp_update_ctx(&metadata,sizeof(metdata));
    bpf_printk("---- kParser packet\n”);
    bpf_printk(“IPv4 offset: %u\n”, metadata.ip_offset);
    bpf_printk(“L4 offset: %u\n”, metadata.l4_offset);
    bpf_printk(“IP address: src %c, dst %x\n”, metadata.addrs[0], metadata.addrs[1]);
    bpf_printk(“IP address: src %c, dst %x\n”, metadata.ports[0], metadata.ports[1]);

    return XDP_PASS;
}
```

Flow Dissector XDP helper

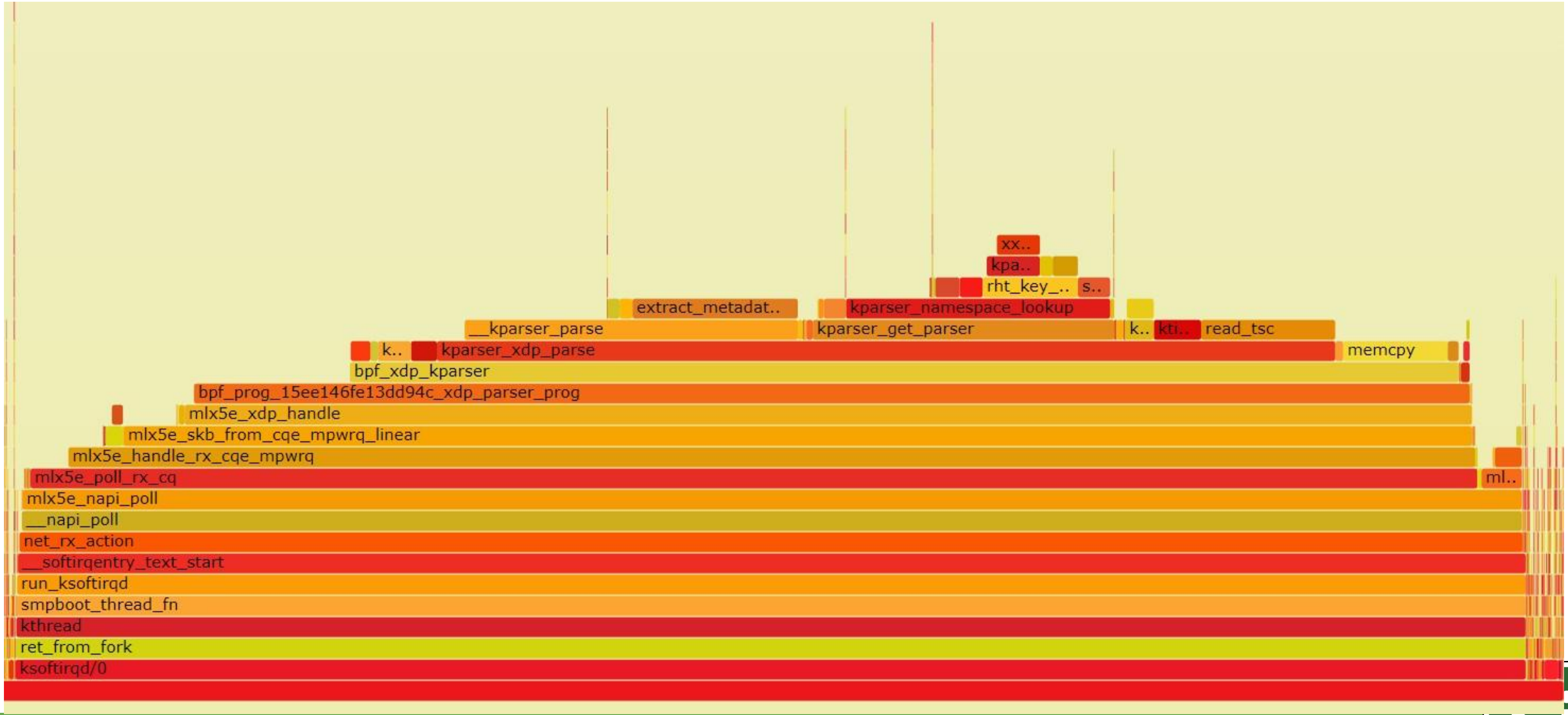
Similar to the kParser helper function, we added an helper function to invoke flow dissector

```
static int xdp_flow_dissector(struct xdp_buff *xdp, u32 flowd_sel, void *buf, u32 len);
```

Performance Comparison



Flame graph for kParser



Upstream status and futures

Upstream

- Kernel patches net-next as RFC
- Will post patches for iproute2 shortly
- iproute2 patches includes
  - CPL .json schema
  - json2ip.py– convert .json to CLI commands
  - ip2json.py– convert ip command .json output to CPL .json

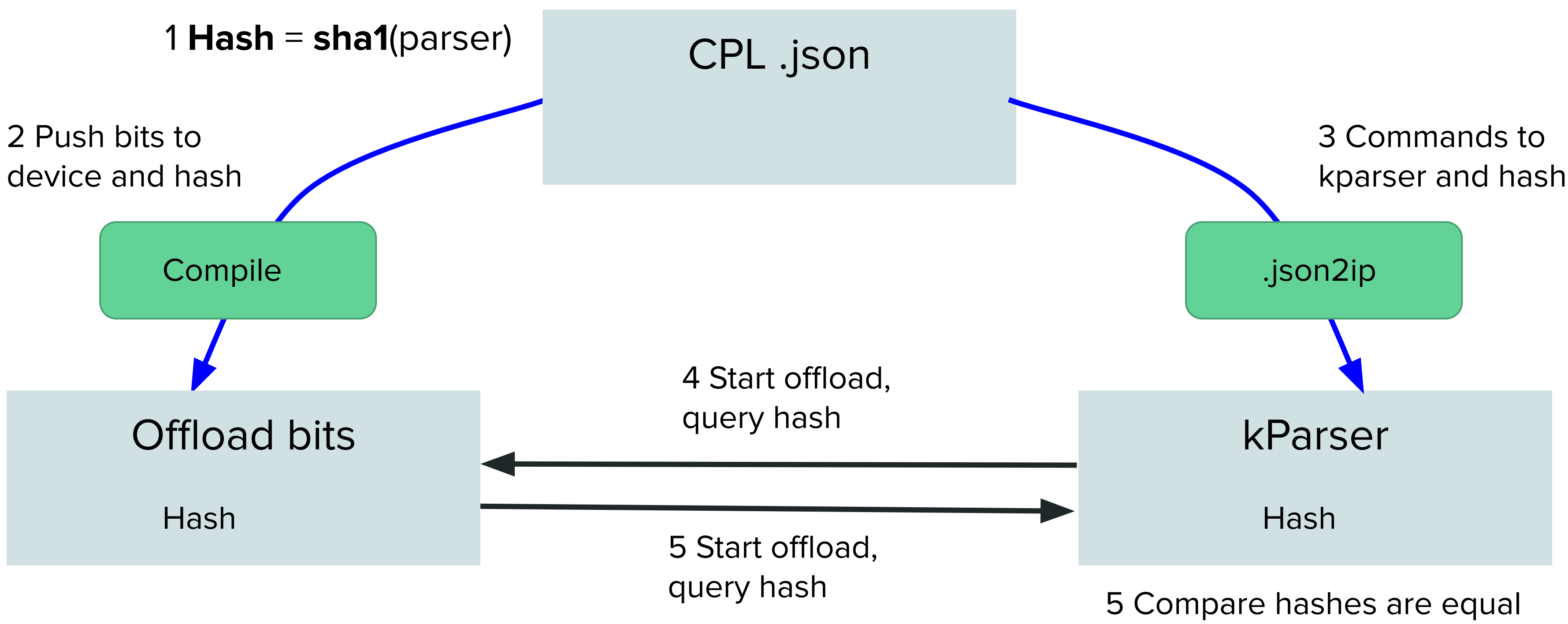
Future work

- Handlers in kParser. Thinking to implement them as eBPF snippets as callbacks in the kParser handling, “handler”: “<function-name>” in CPL
- Canned protocols definitions

```
ip parser create node name node.ipv4 $IPV4_PROTO_PARSER \
    nxt.table-ent 17:ports md-rule md.iphdr_offset md-rule md.ipaddrs
```

- Hardware offload of parsers
  - Fundamental problem of offloads: how does the stack *know* that the offload devices supports the *exact* same functionality and semantics (why their called XDP *hints* :-))
  - Consider a parser that returns the offset of the innermost UDP in a packet. How does the stack know if the device returned the offset of the innermost header, or the encapsulating header?

Solution: Offload with hash compare



Thank you!

Q/A