XDPeriments: Tinkering with DNS and XDP

...

Willem Toorop, Luuk Hendriks
NLnet Labs
SURF RoN 2020 - virtual

Motivation, Ideas & Plans

- Programmable networks are hot (see also: P4), and for good reasons!
- Flexibility in the data plane without sacrificing performance
- Specifically using XDP: easy way to perform some parts *in kernel* (heavy lifting) but still have traditional userspace software 'after' that.

XDP does not have to replace everything we do in userspace, it can *augment* it.

(e)BPF, XDP, DNS

(Extended) Berkeley Packet Filter:

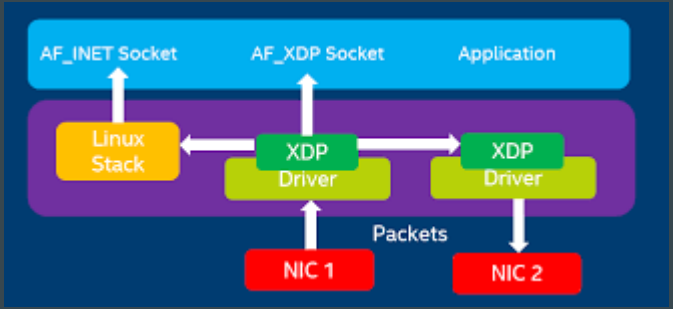
> Once the VM that handles your `tcpdump` filters, now a much more powerful concept with a slightly deceiving name: run verified code in kernel space without rebooting.

eXpress Data Path:

> Network driver hook to run BPF code. Executed before anything happens in the kernel networking stack.

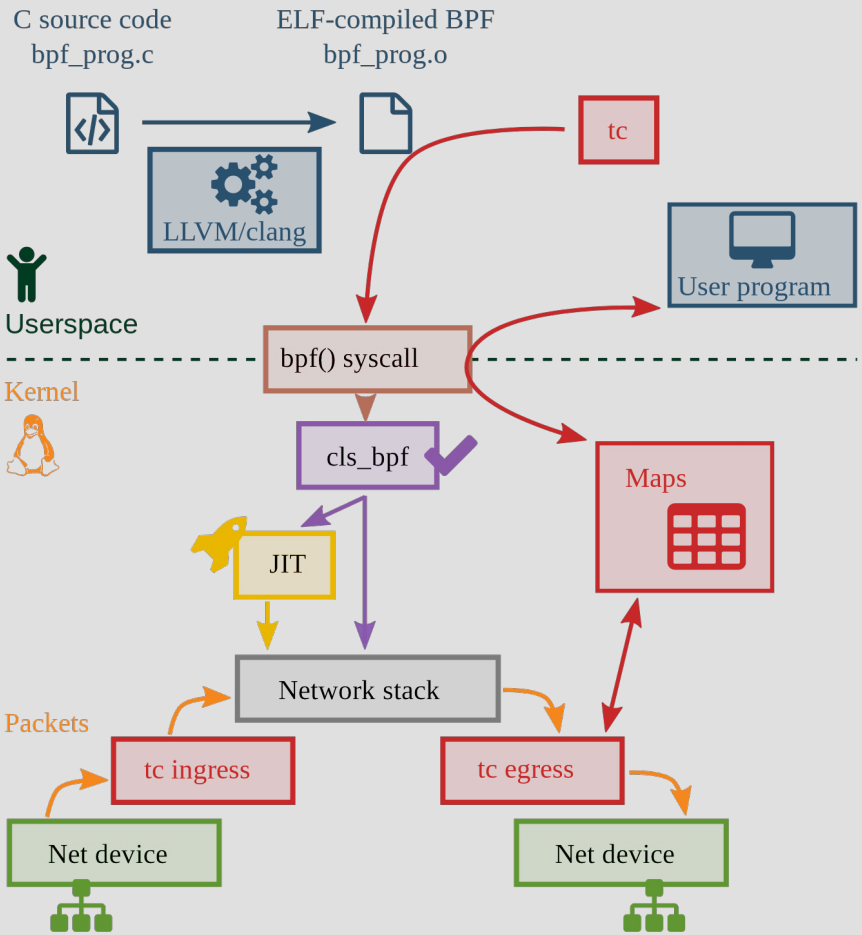
DNS:

> Just DNS.



XDP actions (return codes)

XDP_PASS: pass on to network stack
XDP_DROP: drop the packet
XDP_TX: send it out of the NIC
XDP_ABORTED: program error



https://qmonnet.github.io/whirl-offload/2017/02/11/implementing-openstate-with-ebpf/

DNS says no!

Goal: reply to incoming DNS queries with response code REFUSED

Requirements:

- parse IP, UDP, and DNS
- toggle the QR bit,
  set rcode to REFUSED
- update the checksum,
  send the packet out

DNS says no!

Goal: reply to incoming DNS queries with response code REFUSED

Requirements:

- parse IP, UDP, and DNS
- toggle the QR bit,
  set rcode to REFUSED
- update the checksum,
  send the packet out

```
110  #define PARSE_FUNC_DECLARATION(STRUCT)                    \
111  static __always_inline                                    \
112  struct STRUCT *parse_ ## STRUCT (struct cursor *c)        \
113  {                                                         \
114          struct STRUCT *ret = c->pos;                      \
115          if (c->pos + sizeof(struct STRUCT) > c->end)      \
116                  return 0;                                 \
117          c->pos += sizeof(struct STRUCT);                  \
118          return ret;                                       \
119  }
120
121  PARSE_FUNC_DECLARATION(ethhdr)
122  PARSE_FUNC_DECLARATION(vlanhdr)
123  PARSE_FUNC_DECLARATION(iphdr)
124  PARSE_FUNC_DECLARATION(ipv6hdr)
125  PARSE_FUNC_DECLARATION(udphdr)
126  PARSE_FUNC_DECLARATION(dnshdr)
```

DNS says no!

Goal: reply to incoming DNS querie

Requirements:

- parse IP, UDP, and DNS
- toggle the QR bit,
  set rcode to REFUSED
- update the checksum,
  send the packet out

```
26  int udp_dns_reply(struct cursor *c)
27  {
28      struct udphdr  *udp;
29      struct dnshdr  *dns;
30
31      if (!(udp = parse_udphdr(c))|| udp->dest != __bpf_htons(DNS_PORT)
32      ||  !(dns = parse_dnshdr(c)))
33          return -1;
34
35      uint16_t old_val = dns->flags.as_value;
36      dns->flags.as_bits_and_pieces.ad = 0;
37      dns->flags.as_bits_and_pieces.qr = 1;
38      dns->flags.as_bits_and_pieces.rcode = RCODE_REFUSED;
39      update_checksum(&udp->check, old_val, dns->flags.as_value);
40
41      udp->dest   = udp->source;
42      udp->source = __bpf_htons(DNS_PORT);
43
44      return 0;
45  }
```

DNS says no!

```
$ dig @167.172.45.230 will.you.answer.me. A +norec
```

DNS says no!

```
$ dig @167.172.45.230 will.you.answer.me. A +norec

; <<>> DiG 9.16.1-Ubuntu <<>> @167.172.45.230 will.you.answer.me. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: REFUSED, id: 55544
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1
```

DNS says no!

```
$ dig @167.172.45.230 will.you.answer.me. A +norec

; <<>> DiG 9.16.1-Ubuntu <<>> @167.172.45.230 will.you.answer.me. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: REFUSED, id: 55544
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: c5ebfcd9e4943c77 (echoed)
;; QUESTION SECTION:
;will.you.answer.me.        IN    A
```

DNS says no!

```
$ dig @167.172.45.230 will.you.answer.me. A +norec

; <<>> DiG 9.16.1-Ubuntu <<>> @167.172.45.230 will.you.answer.me. A
; (1 server found)
;; global options: +cmd
;; Got answer:
;; ->>HEADER<<- opcode: QUERY, status: REFUSED, id: 55544
;; flags: qr; QUERY: 1, ANSWER: 0, AUTHORITY: 0, ADDITIONAL: 1

;; OPT PSEUDOSECTION:
; EDNS: version: 0, flags:; udp: 4096
; COOKIE: c5ebfcd9e4943c77 (echoed)
;; QUESTION SECTION:
;will.you.answer.me.        IN    A
```

DNS says no!

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- update the checksum, send the packet out

DNS says no!

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- update the checksum, send the packet out

```
1   static __always_inline
2   uint8_t *parse_dname(struct cursor *c, uint8_t *pkt)
3   {
4           uint8_t *dname = c->pos;
5
```

```
31          return dname;
32  }
```

DNS says no!

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- update the checksum, send the packet out

```
1  static __always_inline
2  uint8_t *parse_dname(struct cursor *c, uint8_t *pkt)
3  {
4          uint8_t *dname = c->pos;
5
6          int  i;
7          for (i = 0; i < 128; i++) { /* Maximum 128 labels */
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30         }
31         return dname;
32 }
```

DNS says no!

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- update the checksum, send the packet out

```
1  static __always_inline
2  uint8_t *parse_dname(struct cursor *c, uint8_t *pkt)
3  {
4          uint8_t *dname = c->pos;
5
6          int  i;
7          for (i = 0; i < 128; i++) { /* Maximum 128 labels */
8                  uint8_t o;
9
10                 if (c->pos + 1 > c->end)
11                         return 0;
12
13                 o = *(uint8_t *)c->pos;
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28                 if (!o)
29                         break;
30         }
31         return dname;
32 }
```

DNS says no!

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- update the checksum, send the packet out

```
1  static __always_inline
2  uint8_t *parse_dname(struct cursor *c, uint8_t *pkt)
3  {
4          uint8_t *dname = c->pos;
5
6          int  i;
7          for (i = 0; i < 128; i++) { /* Maximum 128 labels */
8                  uint8_t o;
9
10                 if (c->pos + 1 > c->end)
11                         return 0;
12
13                 o = *(uint8_t *)c->pos;
14                 if ((o & 0xC0) == 0xC0) {
15                         /* Compression label, Only back references! */
16                         if ((o | 0x3F) >= (dname - pkt))
17                                 return 0;
18
19                         /* Compression label is the last label of a dname. */
20                         c->pos += 1;
21                         break;
22
23                 } else if (o & 0xC0)
24                         /* Unknown label type */
25                         return 0;
26
27                 c->pos += o + 1;
28                 if (!o)
29                         break;
30         }
31         return dname;
32 }
```

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- **update the checksum,** send the packet out

```
3       if (dns->arcount) {
4           struct dns_rr  *opt_rr;
5           uint8_t        *opt_owner = c->pos;
6
```

```
22              opt_rr->rdata_len = 0;
```

```
29              opt_rr->rdata_len = 0;
```

```
32      }
```

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- **update the checksum,** send the packet out

```
3       if (dns->arcount) {
4           struct dns_rr  *opt_rr;
5           uint8_t        *opt_owner = c->pos;
6
7           opt_owner = c->pos;
8           if (++c->pos > c->end || *opt_owner
9           || !(opt_rr = parse_dns_rr(c))
10          ||   opt_rr->type != __bpf_htons(RR_TYPE_OPT))
11              return -1;
12
13          if (opt_rr->rdata_len == 0)
14              ; /* pass */
15
16          else if (c->pos + 1 > c->end)
17              return -1;
```

```
22              opt_rr->rdata_len = 0;
```

```
29              opt_rr->rdata_len = 0;
```

```
32      }
```

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- **update the checksum,** send the packet out

```
3       if (dns->arcount) {
4           struct dns_rr  *opt_rr;
5           uint8_t        *opt_owner = c->pos;
6
7           opt_owner = c->pos;
8           if (++c->pos > c->end || *opt_owner
9           || !(opt_rr = parse_dns_rr(c))
10          ||   opt_rr->type != __bpf_htons(RR_TYPE_OPT))
11              return -1;
12
13          if (opt_rr->rdata_len == 0)
14              ; /* pass */
15
16          else if (c->pos + 1 > c->end)
17              return -1;
```

```
22              opt_rr->rdata_len = 0;
```

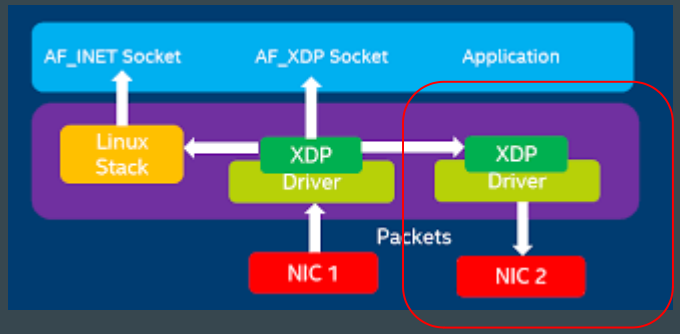
```
27          } else {
28              uint16_t old_val = opt_rr->rdata_len;
29              opt_rr->rdata_len = 0;
30              update_checksum(&udp->check, old_val, 0);
31          }
32      }
```

Requirements:

- find out position of OPT (i.e. parse qname, variable length)
- strip the cookie
- **update the checksum,** send the packet out

```
3       if (dns->arcount) {
4           struct dns_rr  *opt_rr;
5           uint8_t        *opt_owner = c->pos;
6
7           opt_owner = c->pos;
8           if (++c->pos > c->end || *opt_owner
9           || !(opt_rr = parse_dns_rr(c))
10          ||   opt_rr->type != __bpf_htons(RR_TYPE_OPT))
11              return -1;
12
13          if (opt_rr->rdata_len == 0)
14              ; /* pass */
15
16          else if (c->pos + 1 > c->end)
17              return -1;
18
19          else if (((void *)&opt_rr->rdata_len - (void *)dns) % 2 == 1) {
20              uint16_t old1 = *(uint16_t *)(c->pos - 3);
21              uint16_t old2 = *(uint16_t *)(c->pos - 1);
22              opt_rr->rdata_len = 0;
23              update_checksum(&udp->check, old1,
24                      *(uint16_t *)(c->pos - 3));
25              update_checksum(&udp->check, old2,
26                      *(uint16_t *)(c->pos - 1));
27          } else {
28              uint16_t old_val = opt_rr->rdata_len;
29              opt_rr->rdata_len = 0;
30              update_checksum(&udp->check, old_val, 0);
31          }
32      }
```

DNS says no!



XDP return code **XDP_TX**:
send out packet directly without going up to OS network stack

DNS says no! - lessons learned

A certain share of DNS queries can be answered (producing completely valid DNS responses) without touching userland in any way;

Even very simple functionality (DNS says no)
requires massaging the code for the verifier;

Even very simple DNS augmentations require in-depth knowledge of:
IP/transport layers: updating checksums, and,
DNS itself: userland software contains decades of knowledge and experience!

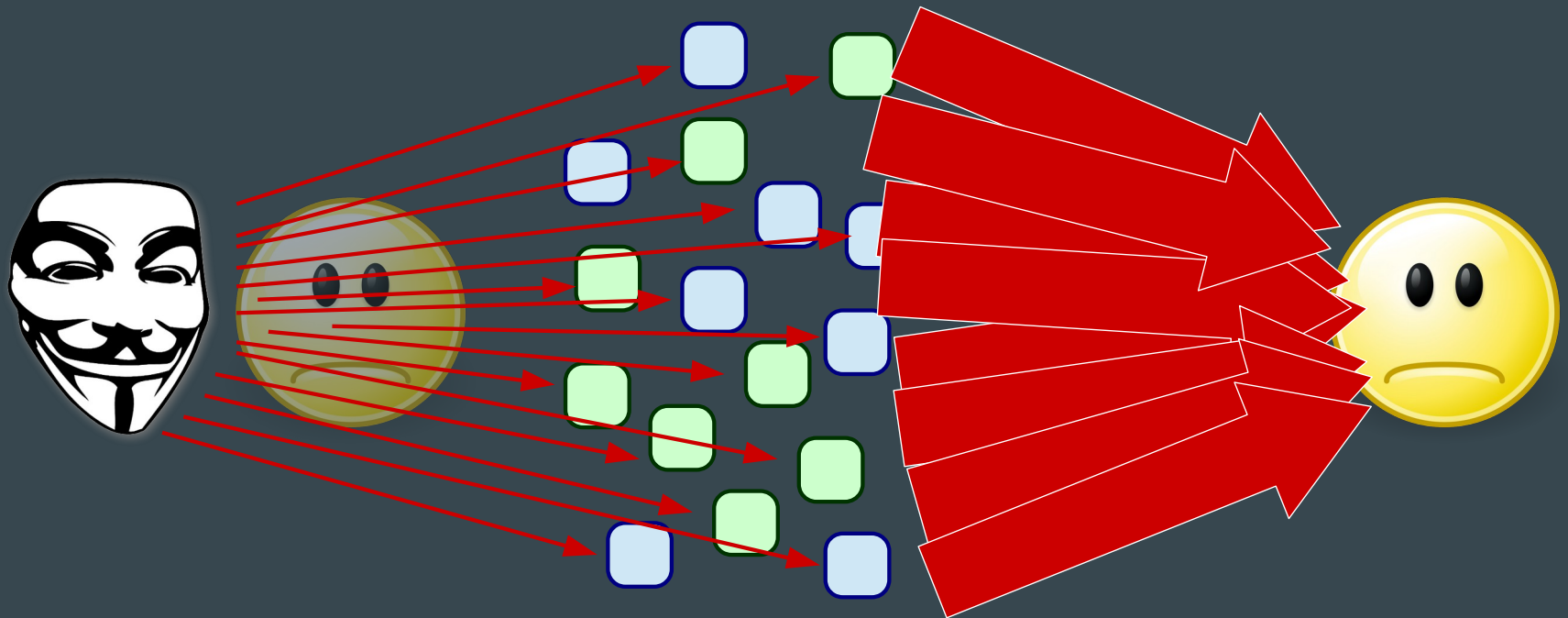
Response Rate Limiting

RP2 project Tom Carpay: https://www.nlnetlabs.nl/downloads/publications/DNS-augmentation-with-eBPF.pdf

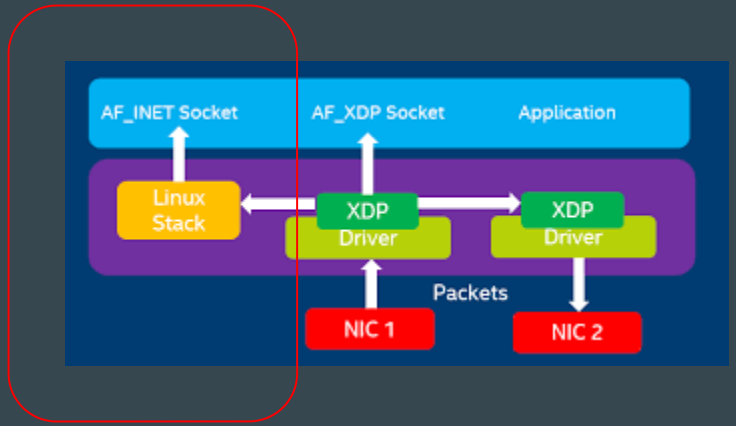
Take away:
XDP enables performant but flexible security mechanisms like RRL, and with that, fulfil an often heard operator desire.
Maps allow stateful processing and on-the-fly configuration

Response Rate Limiting 101

- When Queries per Second > X (from certain source IP or Prefix)
- Then Return truncated (or drop)

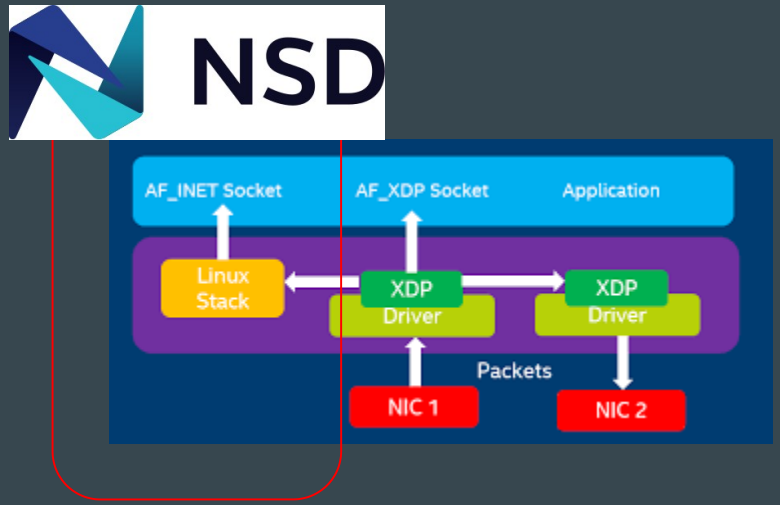


Response Rate Limiting



XDP return code **XDP_PASS** or **XDP_TX**

Response Rate Limiting



XDP return code **XDP_PASS** or **XDP_TX**

Response Rate Limiting

Requirements:

- **define 'state', i.e. BPF *maps***
- update state and act accordingly, on every incoming DNS query

Response Rate Limiting

```
1  struct bucket {
2          uint64_t start_time;
3          uint64_t n_packets;
4  };
5
```

Requirements:

- **define 'state', i.e. BPF *maps***
- update state and act accordingly, on every incoming DNS query

Response Rate Limiting

Requirements:

- **define 'state', i.e. BPF *maps***
- update state and act accordingly, on every incoming DNS query

```
1  struct bucket {
2          uint64_t start_time;
3          uint64_t n_packets;
4  };
5
6  struct bpf_map_def SEC("maps") state_map = {
7          .type = BPF_MAP_TYPE_PERCPU_HASH,
8          .key_size = sizeof(uint32_t),
9          .value_size = sizeof(struct bucket),
10         .max_entries = 1000000
11 };
12
```

Response Rate Limiting

Requirements:

- **define 'state', i.e. BPF *maps***
- update state and act accordingly, on every incoming DNS query

```
1  struct bucket {
2          uint64_t start_time;
3          uint64_t n_packets;
4  };
5
6  struct bpf_map_def SEC("maps") state_map = {
7          .type = BPF_MAP_TYPE_PERCPU_HASH,
8          .key_size = sizeof(uint32_t),
9          .value_size = sizeof(struct bucket),
10         .max_entries = 1000000
11 };
12
13 struct bpf_map_def SEC("maps") state_map_v6 = {
14         .type = BPF_MAP_TYPE_PERCPU_HASH,
15         .key_size = sizeof(struct in6_addr),
16         .value_size = sizeof(struct bucket),
17         .max_entries = 1000000
18 };
```

On the state of BPF Maps

```
  6 enum bpf_map_type {
  5     BPF_MAP_TYPE_UNSPEC,
  4     BPF_MAP_TYPE_HASH,
  3     BPF_MAP_TYPE_ARRAY,
  2     BPF_MAP_TYPE_PROG_ARRAY,
  1     BPF_MAP_TYPE_PERF_EVENT_ARRAY,
118     BPF_MAP_TYPE_PERCPU_HASH,
  1     BPF_MAP_TYPE_PERCPU_ARRAY,
  2     BPF_MAP_TYPE_STACK_TRACE,
  3     BPF_MAP_TYPE_CGROUP_ARRAY,
  4     BPF_MAP_TYPE_LRU_HASH,
  5     BPF_MAP_TYPE_LRU_PERCPU_HASH,
  6     BPF_MAP_TYPE_LPM_TRIE,
  7     BPF_MAP_TYPE_ARRAY_OF_MAPS,
  8     BPF_MAP_TYPE_HASH_OF_MAPS,
  9     BPF_MAP_TYPE_DEVMAP,
 10     BPF_MAP_TYPE_SOCKMAP,
 11     BPF_MAP_TYPE_CPUMAP,
 12     BPF_MAP_TYPE_XSKMAP,
 13     BPF_MAP_TYPE_SOCKHASH,
 14     BPF_MAP_TYPE_CGROUP_STORAGE,
 15     BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
 16     BPF_MAP_TYPE_PERCPU_CGROUP_STORAGE,
 17     BPF_MAP_TYPE_QUEUE,
 18     BPF_MAP_TYPE_STACK,
 19     BPF_MAP_TYPE_SK_STORAGE,
 20     BPF_MAP_TYPE_DEVMAP_HASH,
 21 };
```

/usr/include/linux/bpf.h

Datastructures *specific* to BPF, require specific functions to read/write at runtime, e.g.:

bpf_map_lookup_elem()
bpf_map_update_elem()
bpf_map_delete_elem()

NB: Hardware offloading might not support all of these map types

Response Rate Limiting

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

Response Rate Limit

```
2  int udp_dns_reply_v4(struct cursor *c, uint32_t key)
3  {
4          struct udphdr  *udp;
5          struct dnshdr  *dns;
6
7          // check that we have a DNS packet
8          if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
9          ||  !(dns = parse_dnshdr(c)))
10                 return 1;
```

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
27 }
```

Response Rate Limit

```
2   int udp_dns_reply_v4(struct cursor *c, uint32_t key)
3   {
4           struct udphdr  *udp;
5           struct dnshdr  *dns;
6
7           // check that we have a DNS packet
8           if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
9           ||  !(dns = parse_dnshdr(c)))
10                  return 1;
```

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
27  }
```

Response Rate Limit

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
2   int udp_dns_reply_v4(struct cursor *c, uint32_t key)
3   {
4           struct udphdr  *udp;
5           struct dnshdr  *dns;
6
7           // check that we have a DNS packet
8           if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
9           ||  !(dns = parse_dnshdr(c)))
10                  return 1;
11
12          // get the rrl bucket from the map by IPv4 address
13          struct bucket *b = bpf_map_lookup_elem(&state_map, &key);
14
15          // did we see this IPv4 address before?
16          if (b)
17                  return do_rate_limit(udp, dns, b);
18
27  }
```

src ip

Response Rate Limit

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```c
int udp_dns_reply_v4(struct cursor *c, uint32_t key)
{
        struct udphdr  *udp;
        struct dnshdr  *dns;

        // check that we have a DNS packet
        if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
        ||  !(dns = parse_dnshdr(c)))
                return 1;

        // get the rrl bucket from the map by IPv4 address
        struct bucket *b = bpf_map_lookup_elem(&state_map, &key);

        // did we see this IPv4 address before?
        if (b)
                return do_rate_limit(udp, dns, b);
```

src ip

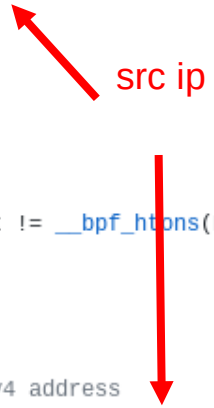
```c
}
```

Response Rate Limit

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
2  int udp_dns_reply_v4(struct cursor *c, uint32_t key)
3  {
4          struct udphdr  *udp;
5          struct dnshdr  *dns;
6
7          // check that we have a DNS packet
8          if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
9          ||  !(dns = parse_dnshdr(c)))
10                 return 1;
11
12         // get the rrl bucket from the map by IPv4 address
13         struct bucket *b = bpf_map_lookup_elem(&state_map, &key);
14
15         // did we see this IPv4 address before?
16         if (b)
17                 return do_rate_limit(udp, dns, b);
18
19         // create new starting bucket for this IPv4 address
20         struct bucket new_bucket;
21         new_bucket.start_time = bpf_ktime_get_ns();
22         new_bucket.n_packets = 0;
23
24         // store the bucket and pass the packet
25         bpf_map_update_elem(&state_map, &key, &new_bucket, BPF_ANY);
26         return 1;
27 }
```



Response Rate Limiting

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

Response Rate Li

```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
```

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

Response Rate Li

```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
```

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

Response Rate Li

```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
4          // increment number of packets
5          b->n_packets++;
6
7          // get the current and elapsed time
8          uint64_t now = bpf_ktime_get_ns();
9          uint64_t elapsed = now - b->start_time;
10
```

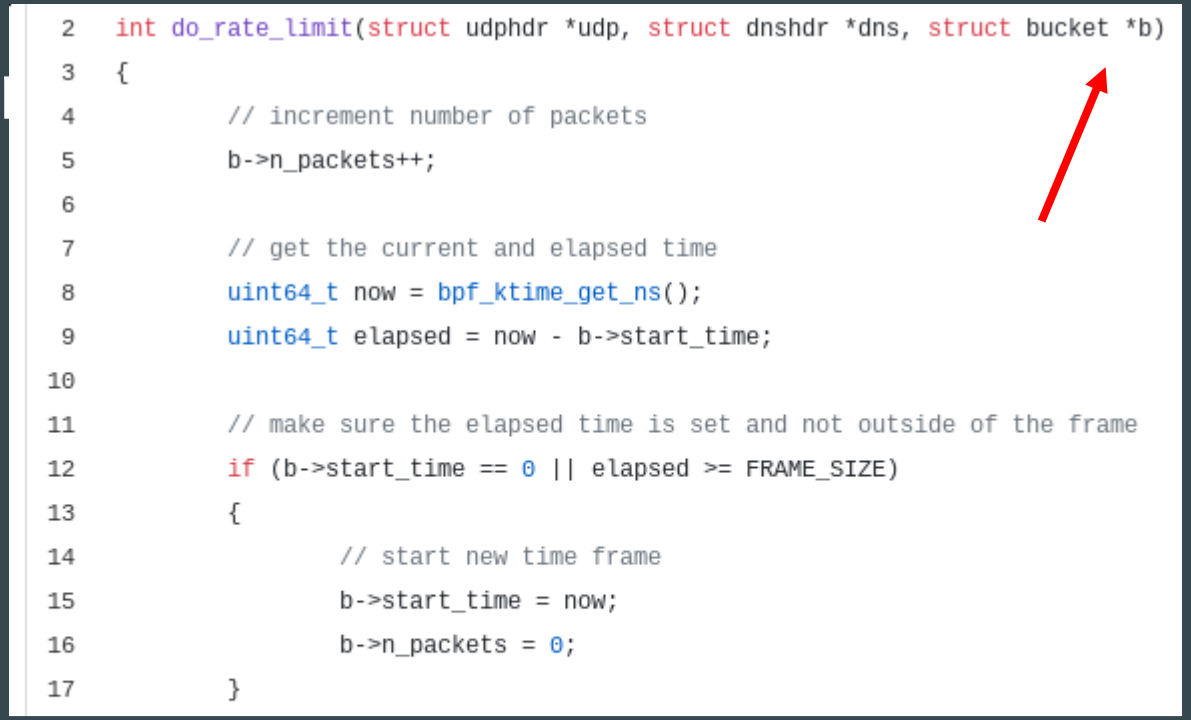
Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

Response Rate Li

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**



```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
4          // increment number of packets
5          b->n_packets++;
6
7          // get the current and elapsed time
8          uint64_t now = bpf_ktime_get_ns();
9          uint64_t elapsed = now - b->start_time;
10
11         // make sure the elapsed time is set and not outside of the frame
12         if (b->start_time == 0 || elapsed >= FRAME_SIZE)
13         {
14                 // start new time frame
15                 b->start_time = now;
16                 b->n_packets = 0;
17         }
```

Response Rate Li

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
4          // increment number of packets
5          b->n_packets++;
6
7          // get the current and elapsed time
8          uint64_t now = bpf_ktime_get_ns();
9          uint64_t elapsed = now - b->start_time;
10
11         // make sure the elapsed time is set and not outside of the frame
12         if (b->start_time == 0 || elapsed >= FRAME_SIZE)
13         {
14                 // start new time frame
15                 b->start_time = now;
16                 b->n_packets = 0;
17         }
```

multiple modifications per packet

Response Rate Li

Requirements:

- define 'state', i.e. BPF *maps*
- **update state and act accordingly, on every incoming DNS query**

```
2  int do_rate_limit(struct udphdr *udp, struct dnshdr *dns, struct bucket *b)
3  {
4          // increment number of packets
5          b->n_packets++;
6
7          // get the current and elapsed time
8          uint64_t now = bpf_ktime_get_ns();
9          uint64_t elapsed = now - b->start_time;
10
11         // make sure the elapsed time is set and not outside of the frame
12         if (b->start_time == 0 || elapsed >= FRAME_SIZE)
13         {
14                 // start new time frame
15                 b->start_time = now;
16                 b->n_packets = 0;
17         }
18
19         // less QPS than the threshold? Then pass.
20         if (b->n_packets < THRESHOLD)
21                 return 1;
22 (...truncated ...)
```

multiple modifications per packet

RRL VIP

Operator request:

> *"RRL, but not for $very_important_prefix"*

RRL VIP

Operator request:

"RRL, but not for $very_important_prefix"

```
struct bpf_map_def SEC("maps") exclude_v4_prefixes = {
        .type = BPF_MAP_TYPE_LPM_TRIE,
        .key_size = sizeof(struct bpf_lpm_trie_key) + sizeof(uint32_t),
        .value_size = sizeof(uint64_t),
        .max_entries = 10000
};

struct bpf_map_def SEC("maps") exclude_v6_prefixes = {
        .type = BPF_MAP_TYPE_LPM_TRIE,
        .key_size = sizeof(struct bpf_lpm_trie_key) + 8, // first 64 bits
        .value_size = sizeof(uint64_t),
        .max_entries = 10000
};
```

RRL VIP

Operator request:

"RRL, but not for $very_important_prefix"

```
1  struct bpf_map_def SEC("maps") exclude_v4_prefixes = {
2          .type = BPF_MAP_TYPE_LPM_TRIE,
3          .key_size = sizeof(struct bpf_lpm_trie_key) + sizeof(uint32_t),
4          .value_size = sizeof(uint64_t),
5          .max_entries = 10000
6  };
7
8  struct bpf_map_def SEC("maps") exclude_v6_prefixes = {
9          .type = BPF_MAP_TYPE_LPM_TRIE,
10         .key_size = sizeof(struct bpf_lpm_trie_key) + 8, // first 64 bits
11         .value_size = sizeof(uint64_t),
12         .max_entries = 10000
13 };
```

NB:
not a 'PERCPU' datastructure!
Easier for reads, harder for writes in this use case.

RRL VIP: userspace flexibility, kernel performance

RRL VIP: userspace flexibility, kernel performance

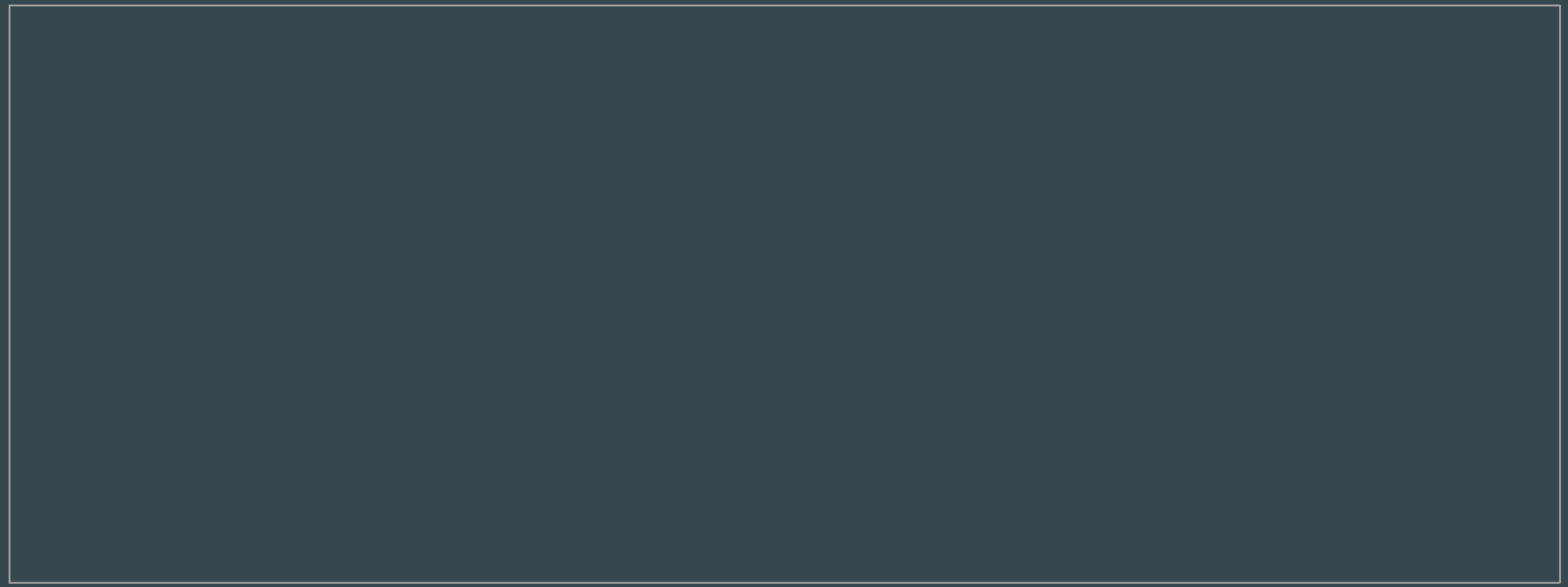
```
# mount -t bpf /sys/fs/bpf
# bpftool map create /sys/fs/bpf/rrl_exclude_v4_prefixes flags 1 \
      name exclude_v4_prefixes type lpm_trie key 8 value 8 entries 10000
# bpftool prog load xdp_rrl_VIP_list.o /sys/fs/bpf/rrl_VIP_list type xdp \
      map name exclude_v4_prefixes pinned /sys/fs/bpf/rrl_exclude_v4_prefixes
# ip link set dev eth0 xdpgeneric pinned /sys/fs/bpf/rrl_VIP_list
```

RRL VIP: userspace flexibility, kernel performance

```
# mount -t bpf /sys/fs/bpf
# bpftool map create /sys/fs/bpf/rrl_exclude_v4_prefixes flags 1 \
       name exclude_v4_prefixes type lpm_trie key 8 value 8 entries 10000
# bpftool prog load xdp_rrl_VIP_list.o /sys/fs/bpf/rrl_VIP_list type xdp \
       map name exclude_v4_prefixes pinned /sys/fs/bpf/rrl_exclude_v4_prefixes
# ip link set dev eth0 xdpgeneric pinned /sys/fs/bpf/rrl_VIP_list


# bpftool map update pinned /sys/fs/bpf/rrl_exclude_v4_prefixes \
       key 24 0 0 0  80 114 156 0 \
       value 0 0 0 0 0 0 0 0
```

RRL VIP: userspace flexibility, kernel performance



RRL VIP: userspace flexibility, kernel performance

```
$ dig -4 foo.nl @bpf.nlnetlabs.net
```

RRL VIP: userspace flexibility, kernel performance

```
$ dig -4 foo.nl @bpf.nlnetlabs.net
```

```
# bpftool map dump pinned /sys/fs/bpf/rrl_exclude_v4_prefixes
key: 18 00 00 00 50 72 9c 62  value: 01 00 00 00 00 00 00 00
Found 1 element
```

Response Rate Limiting - lessons learned

We can leverage XDP to *augment* DNS services:
handle the packet in XDP, or,
decide to punt it upwards to a userspace nameserver

Maps enable keeping state,
not only for e.g. statistics and rates calculations,
but moreover for configuration from userspace at runtime

When choosing a BPF map type, consider concurrency (PERCPU or not)
and possible performance hits

DNS Cookies

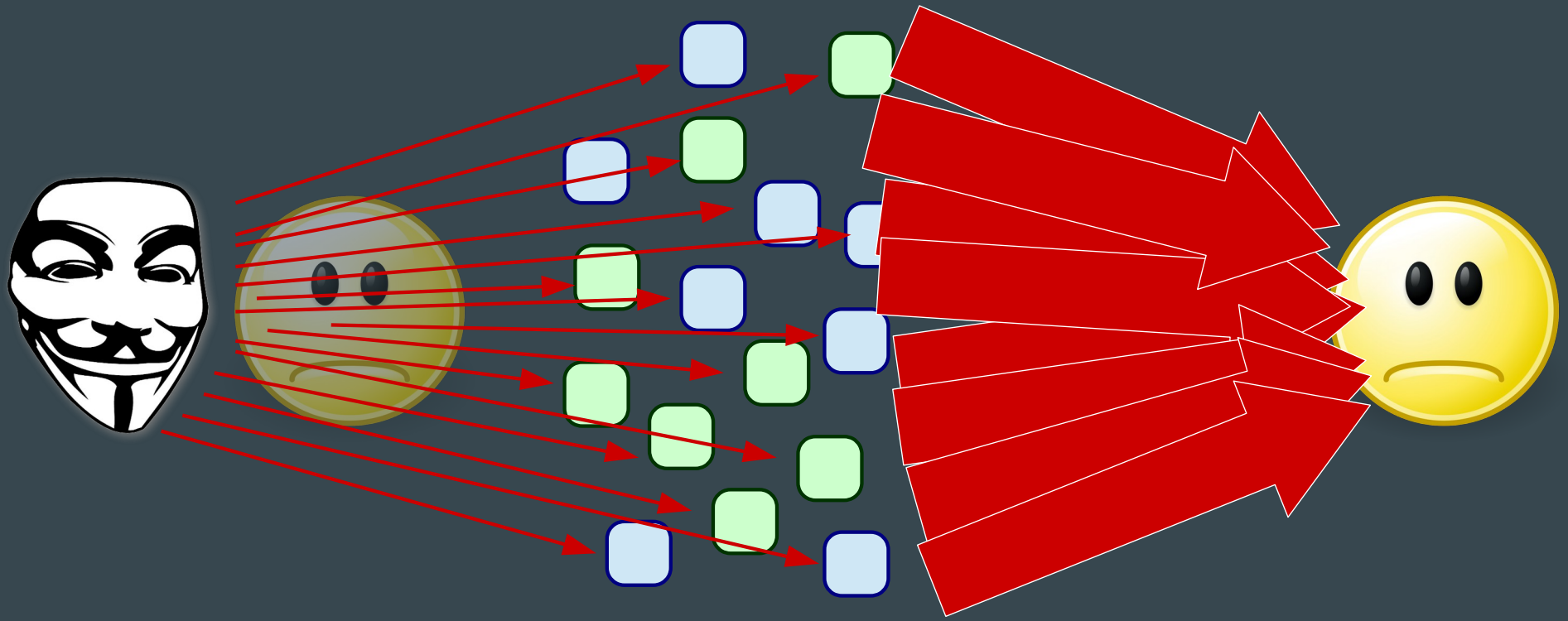
ongoing work

Cookies 101

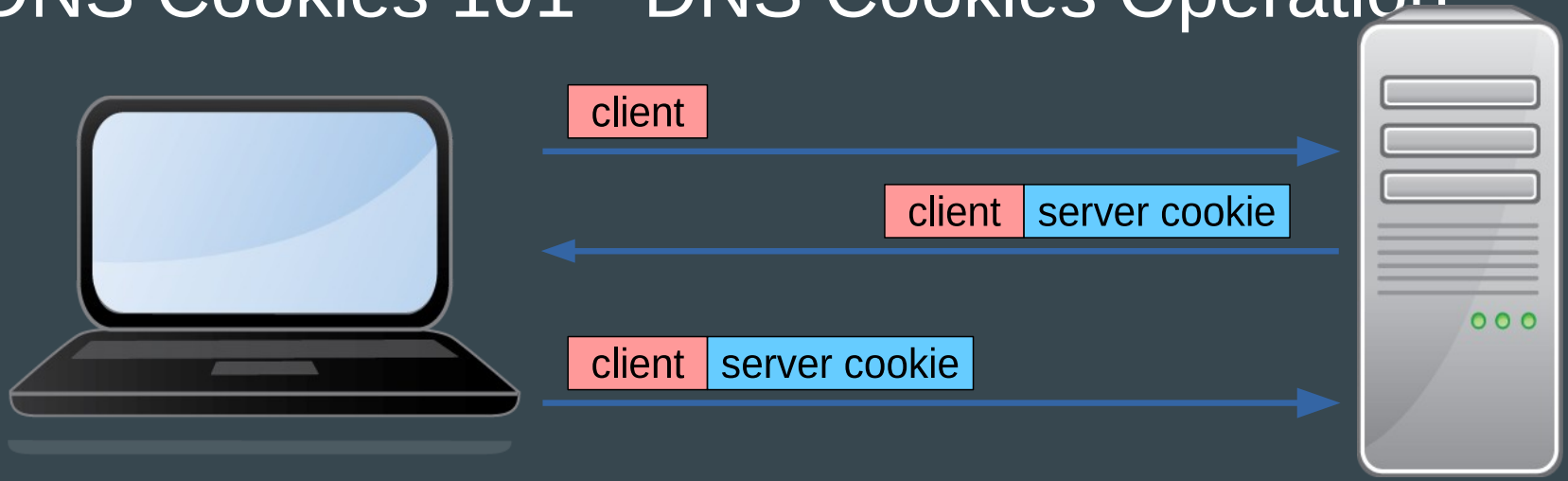
+ Using existing crypto code: siphash
+ Combining/composing XDP programs
  with bpf_tail_call()
+ egress eBPF on the TC layer
+ relate incoming to outgoing packets

takeaway: XDP can be put to work ad hoc
to mitigate DDoS attacks on existing deployments
that had not even considered XDP in the first place

DNS Cookies 101 - Why DNS Cookies?



DNS Cookies 101 - DNS Cookies Operation



- Valid Server Cookie? Large answers
- Valid Server Cookie? RRL disabled

DNS Co

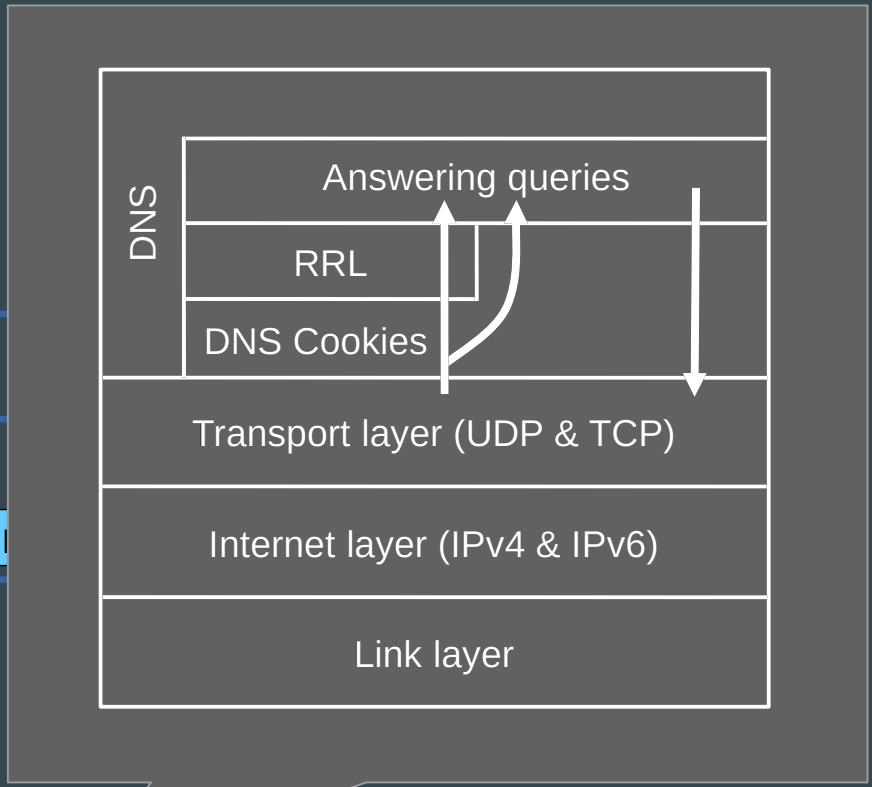
DNS Cookies protect
against SAD DNS!

- Valid S ver Cookie? Large answers
- Valid Server Cookie? RRL disabled

DNS Cookies 101 - DNS

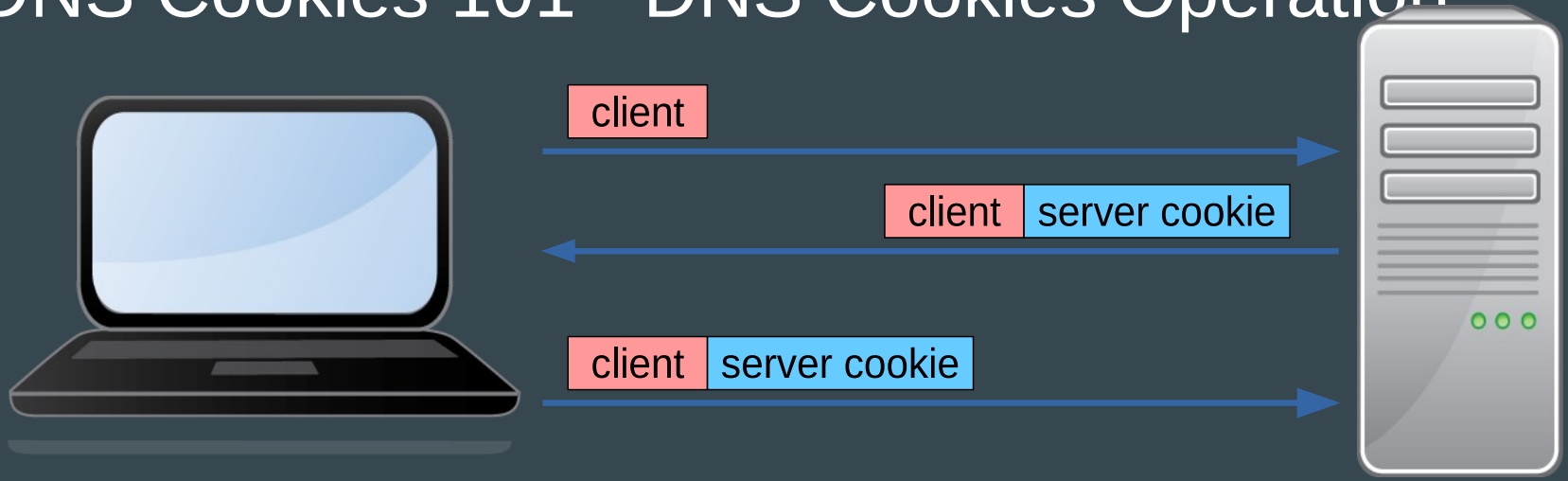
client

client server



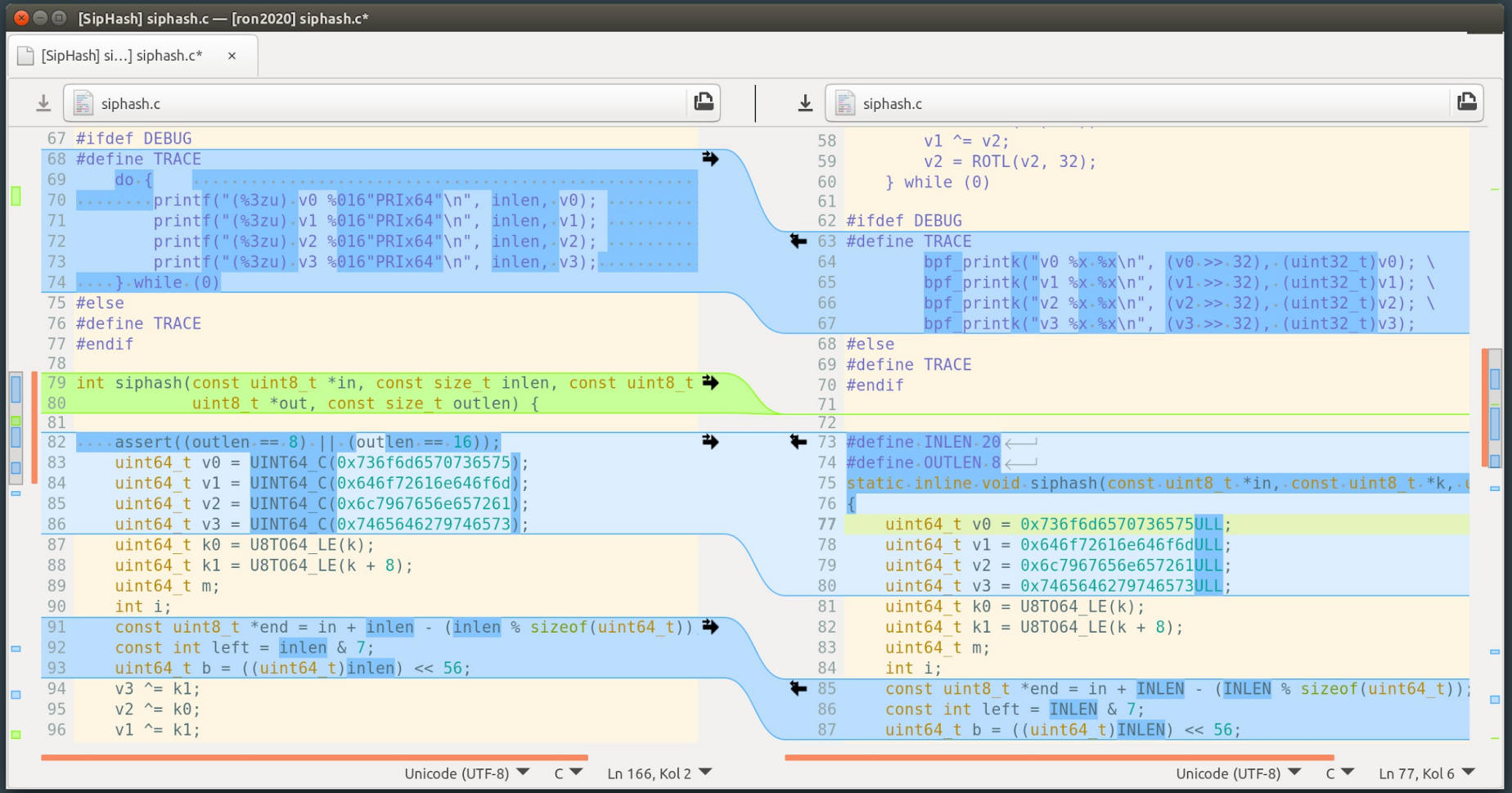
- Valid Server Cookie? Larg answers
- Valid Server Cookie? RRL disabled

DNS Cookies 101 - DNS Cookies Operation



Server Cookie

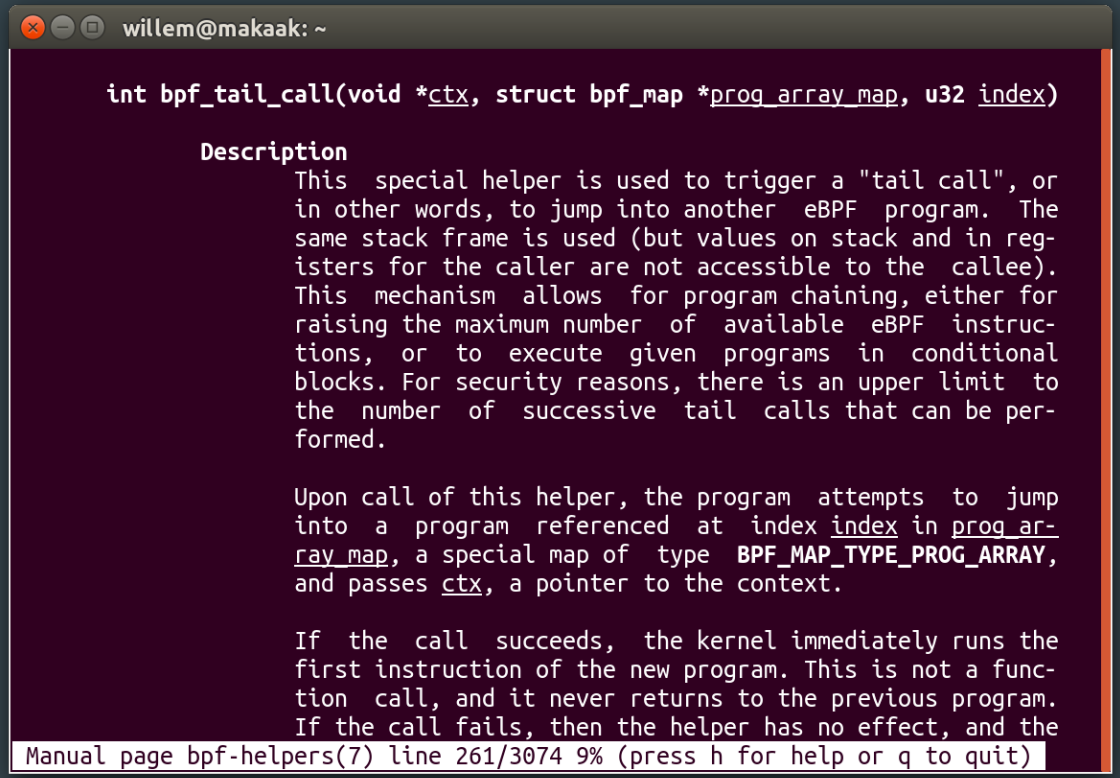
Hash = SipHash2.4(Client Cookie
| Version
| Reserved
| Timestamp
| Client-IP
, Server Secret)


[SipHash] siphash.c — [ron2020] siphash.c*
[SipHash] si…] siphash.c*
siphash.c
siphash.c
#ifdef DEBUG
#define TRACE
do {
printf("(%3zu) v0 %016"PRIx64"\n", inlen, v0);
printf("(%3zu) v1 %016"PRIx64"\n", inlen, v1);
printf("(%3zu) v2 %016"PRIx64"\n", inlen, v2);
printf("(%3zu) v3 %016"PRIx64"\n", inlen, v3);
} while (0)
#else
#define TRACE
#endif
int siphash(const uint8_t *in, const size_t inlen, const uint8_t
uint8_t *out, const size_t outlen) {
assert((outlen == 8) || (outlen == 16));
uint64_t v0 = UINT64_C(0x736f6d6570736575);
uint64_t v1 = UINT64_C(0x646f72616e646f6d);
uint64_t v2 = UINT64_C(0x6c7967656e657261);
uint64_t v3 = UINT64_C(0x7465646279746573);
uint64_t k0 = U8TO64_LE(k);
uint64_t k1 = U8TO64_LE(k + 8);
uint64_t m;
int i;
const uint8_t *end = in + inlen - (inlen % sizeof(uint64_t));
const int left = inlen & 7;
uint64_t b = ((uint64_t)inlen) << 56;
v3 ^= k1;
v2 ^= k0;
v1 ^= k1;
v1 ^= v2;
v2 = ROTL(v2, 32);
} while (0)
#ifdef DEBUG
#define TRACE
bpf_printk("v0 %x %x\n", (v0 >> 32), (uint32_t)v0); \
bpf_printk("v1 %x %x\n", (v1 >> 32), (uint32_t)v1); \
bpf_printk("v2 %x %x\n", (v2 >> 32), (uint32_t)v2); \
bpf_printk("v3 %x %x\n", (v3 >> 32), (uint32_t)v3);
#else
#define TRACE
#endif
#define INLEN 20
#define OUTLEN 8
static inline void siphash(const uint8_t *in, const uint8_t *k,
{
uint64_t v0 = 0x736f6d6570736575ULL;
uint64_t v1 = 0x646f72616e646f6dULL;
uint64_t v2 = 0x6c7967656e657261ULL;
uint64_t v3 = 0x7465646279746573ULL;
uint64_t k0 = U8TO64_LE(k);
uint64_t k1 = U8TO64_LE(k + 8);
uint64_t m;
int i;
const uint8_t *end = in + INLEN - (INLEN % sizeof(uint64_t));
const int left = INLEN & 7;
uint64_t b = ((uint64_t)INLEN) << 56;
Unicode (UTF-8)
C
Ln 166, Kol 2
Unicode (UTF-8)
C
Ln 77, Kol 6

DNS Cookies - Divide and ~~Conquer~~ Overcome

- Variable length qname + Variable length options == too much!
- Verifier needs hard limits
- Split up in pieces with: bpf_tail_call()

- Also for composing eBPF/XDP programs



DNS Cookies - Pass info with meta data

- bpf_tail_call() is like goto

```
272: #define DO_RATE_LIMIT 0
273: #define HANDLE_IPv6   1
274: #define HANDLE_IPv4   2
275:
276: struct meta_data {
277:         uint16_t eth_proto;
278:         uint16_t ip_pos;
279:         uint16_t opt_pos;
280:         uint16_t unused;
281: };
282:
```

```
387: SEC("xdp-dns-cookies")
388: int xdp_dns_cookies(struct xdp_md *ctx)
389: {
390:         struct meta_data *md = (void *)(long)ctx->data_meta;
391:         struct cursor     c;
392:         struct ethhdr    *eth;
393:         struct ipv6hdr   *ipv6;
394:         struct iphdr     *ipv4;
395:         enum xdp_action   action = XDP_PASS;
396:
397:         if (bpf_xdp_adjust_meta(ctx, -(int)sizeof(struct meta_data)))
398:                 return XDP_PASS;
399:
400:         cursor_init(&c, ctx);
401:         md = (void *)(long)ctx->data_meta;
402:         if ((void *)(md + 1) > c.pos)
403:                 return XDP_PASS;
404:
405:         if (!(eth = parse_eth(&c, &md->eth_proto)))
406:                 return XDP_PASS;
407:         md->ip_pos = c.pos - (void *)eth;
408:
409:         if (md->eth_proto == __bpf_htons(ETH_P_IPV6)) {
410:                 if (!(ipv6 = parse_ipv6hdr(&c))
411:                 ||    ipv6->nexthdr != IPPROTO_UDP
412:                 ||   (action = parse_udp_dns(&c, ctx)) != XDP_TX)
413:                         return action;
414:
415:                 md->opt_pos = c.pos - (void *)(ipv6 + 1);
416:                 bpf_tail_call(ctx, &jmp_table, HANDLE_IPv6);
417:
418:         } else if (md->eth_proto == __bpf_htons(ETH_P_IP)) {
419:                 if (!(ipv4 = parse_iphdr(&c))
```

DNS Cooki
Pass info
meta dat

- bpf_tail_call() is like goto

```
272: #define DO_RATE_LIMI
273: #define HANDLE_IPv6
274: #define HANDLE_IPv4
275:
276: struct meta_data {
277:         uint16_t eth_proto;
278:         uint16_t ip_pos;
279:         uint16_t opt_pos;
280:         uint16_t unused;
281: };
282:
```

```
357: static inline
358: enum xdp_action parse_udp_dns(struct cursor *c, struct xdp_md *ctx)
359: {
360:         struct udphdr *udp;
361:         struct dnshdr *dns;
362:
363:         if (!(udp = parse_udphdr(c)) || udp->dest != __bpf_htons(DNS_PORT)
364:         ||  !(dns = parse_dnshdr(c)))
365:                 return XDP_PASS;
366:
367:         if (dns->qr
368:         ||  dns->qdcount != __bpf_htons(1)
369:         ||  dns->ancount || dns->nscount
370:         ||  dns->arcount >  __bpf_htons(1)
371:         ||  !parse_dname(c, (void *)dns)
372:         ||  !parse_dns_qrr(c))
373:                 return XDP_ABORTED;
374:
375:         if (dns->arcount == 0) {
376:                 bpf_tail_call(ctx, &jmp_table, DO_RATE_LIMIT);
377:                 return XDP_PASS;
378:         }
379:         if (c->pos + 1 > c->end
380:         ||  *(uint8_t *)c->pos != 0)
381:                 return XDP_ABORTED;
382:         c->pos += 1;
383:
384:         return XDP_TX;
385: }
```

```
387: SEC("xdp-dns-cookies")
```

```
418:         } else if (md->eth_proto == __bpf_htons(ETH_P_IP)) {
419:                 if (!(ipv4 = parse_iphdr(&c))
```

DNS Cookies - Pass info with meta data

- bpf_tail_call() is like goto

```
272: #define DO_RATE_LIMIT 0
273: #define HANDLE_IPv6   1
274: #define HANDLE_IPv4   2
275:
276: struct meta_data {
277:         uint16_t eth_proto;
278:         uint16_t ip_pos;
279:         uint16_t opt_pos;
280:         uint16_t unused;
281: };
282:
```

```
307: SEC("xdp-handle-ipv4")
308: int xdp_handle_ipv4(struct xdp_md *ctx)
309: {
310:         struct cursor     c;
311:         struct meta_data *md = (void *)(long)ctx->data_meta;
312:         struct iphdr     *ipv4;
313:         struct dns_rr    *opt_rr;
314:         uint16_t          rdata_len;
315:         uint8_t           i;
316:
317:         cursor_init(&c, ctx);
318:         if ((void *)(md + 1) > c.pos || md->ip_pos > 24)
319:                 return XDP_ABORTED;
320:         c.pos += md->ip_pos;
321:
322:         if (!(ipv4 = parse_iphdr(&c)) || md->opt_pos > 4096)
323:                 return XDP_ABORTED;
324:         c.pos += md->opt_pos;
325:
326:         if (!(opt_rr = parse_dns_rr(&c))
327:         ||    opt_rr->type != __bpf_htons(RR_TYPE_OPT))
328:                 return XDP_ABORTED;
329:
330:         rdata_len = __bpf_ntohs(opt_rr->rdata_len);
331:         for (i = 0; i < 4 && rdata_len >= 28; i++) {
332:                 struct option *opt;
333:                 uint16_t       opt_len;
334:
335:                 if (!(opt = parse_option(&c)))
336:                         return XDP_ABORTED;
337:
338:                 rdata_len -= 4;
339:                 opt_len = __bpf_ntohs(opt->len);
```

DNS Cookies - Pass info with meta data

- bpf_tail_call() is like goto

```
272: #define DO_RATE_LIMIT 0
273: #define HANDLE_IPv6   1
274: #define HANDLE_IPv4   2
275:
276: struct meta_data {
277:         uint16_t eth_proto;
278:         uint16_t ip_pos;
279:         uint16_t opt_pos;
280:         uint16_t unused;
281: };
282:
```

```
324:         c.pos += md->opt_pos;
325:
326:         if (!(opt_rr = parse_dns_rr(&c))
327:         ||    opt_rr->type != __bpf_htons(RR_TYPE_OPT))
328:                 return XDP_ABORTED;
329:
330:         rdata_len = __bpf_ntohs(opt_rr->rdata_len);
331:         for (i = 0; i < 4 && rdata_len >= 28; i++) {
332:                 struct option *opt;
333:                 uint16_t       opt_len;
334:
335:                 if (!(opt = parse_option(&c)))
336:                         return XDP_ABORTED;
337:
338:                 rdata_len -= 4;
339:                 opt_len = __bpf_ntohs(opt->len);
340:                 if (opt->code == __bpf_htons(OPT_CODE_COOKIE)) {
341:                         if (opt_len == 24 && c.pos + 24 <= c.end
342:                         &&  cookie_verify_4(&c, ipv4))
343:                                 return XDP_PASS;
344:                         break;
345:                 }
346:                 if (opt_len > 1500 || opt_len > rdata_len
347:                 ||  c.pos + opt_len > c.end)
348:                         return XDP_ABORTED;
349:
350:                 rdata_len -= opt_len;
351:                 c.pos += opt_len;
352:         }
353:         bpf_tail_call(ctx, &jmp_table, DO_RATE_LIMIT);
354:         return XDP_PASS;
355: }
```

DNS Cookies

- bpf_tail_call()
  Needs loader program

```
 5: #include <bpf.h>
 6: #include <libbpf.h>
 7:
 8: int main(int argc, char **argv)
 9: {
10:         struct bpf_program *prog;
11:         struct bpf_object  *obj = NULL;
12:         int fd = -1, jmp_table_fd = -1;
13:         uint32_t key = 0;
14:
15:         if (!(obj = bpf_object__open_file("xdp_dns_cookies_kern.o", NULL))
16:         ||  libbpf_get_error(obj))
17:                 fprintf(stderr, "ERROR: opening BPF object file failed\n");
18:
19:         else if (bpf_object__load(obj))
20:                 fprintf(stderr, "ERROR: loading BPF object file failed\n");
21:
22:         else if ((jmp_table_fd = bpf_object__find_map_fd_by_name(obj, "jmp_table")) < 0)
23:                 fprintf(stderr, "ERROR: finding jmp_table failed\n");
24:
25:         else bpf_object__for_each_program(prog, obj) {
26:                 const char *title = bpf_program__title(prog, false);
27:
28:                 fd = bpf_program__fd(prog);
29:                 printf("key: %d, title: %s, fd: %d -> ", key, title, fd);
30:                 int r = bpf_map_update_elem(jmp_table_fd, &key, &fd, BPF_ANY);
31:                 printf("%d\n", r);
32:                 key++;
33:         }
34:         if (fd < 0)
35:                 ; /* earlier error */
36:
37:         else if (bpf_set_link_xdp_fd(1, fd, 0))
38:                 fprintf(stderr, "ERROR: attaching xdp program to device\n");
39:         else
40:                 while (true)
41:                         sleep(60);
42:
43:         return EXIT_FAILURE;
44: }
```

DNS

- bpf_t
  Need

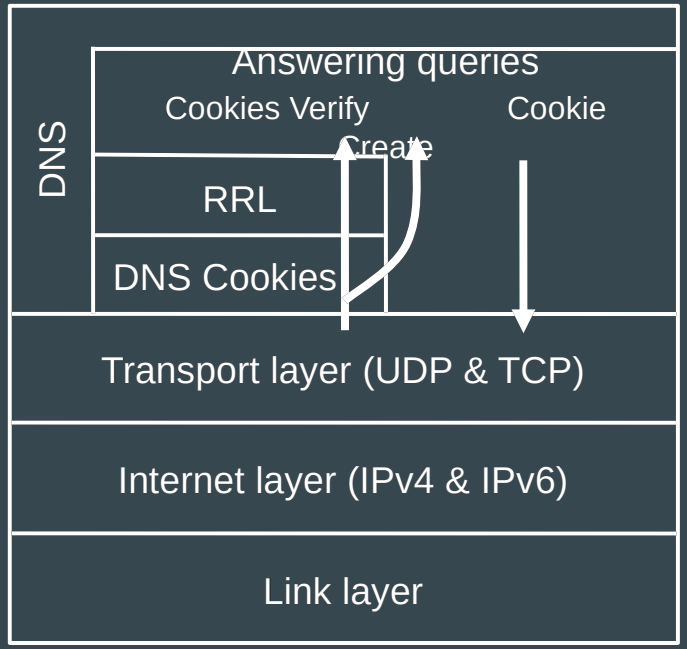
```
5: #include <bpf.h>
```

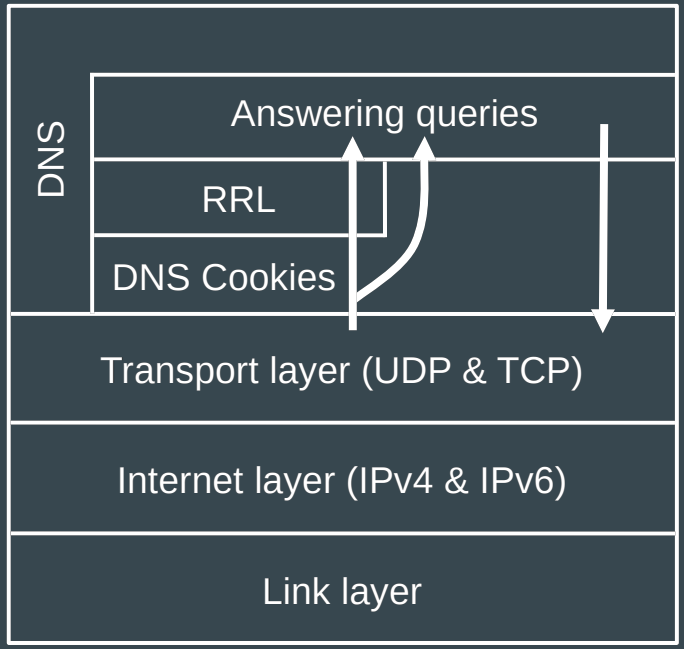
willem@makaak: ~/repos/XDPeriments/Cookies/Round1

```
willem@makaak:~/repos/XDPeriments/Cookies/Round1$ make
clang-9 -target bpf -O2 -Wall -Werror -I../../libbpf/src -c -o xdp_dns_cookies_k
ern.o xdp_dns_cookies_kern.c
clang-9 -static -O2 -Wall -Werror -I../../libbpf/src -o xdp_dns_cookies_user xdp
_dns_cookies_user.c -L../../libbpf/src -lbpf -lelf -lz
willem@makaak:~/repos/XDPeriments/Cookies/Round1$ sudo ./xdp_dns_cookies_user
key: 0, title: xdp-do-rate-limit, fd: 4 -> 0
key: 1, title: xdp-handle-ipv6, fd: 5 -> 0
key: 2, title: xdp-handle-ipv4, fd: 6 -> 0
key: 3, title: xdp-dns-cookies, fd: 7 -> 0
```

```
43:         return EXIT_FAILURE;
44: }
```

DNS Cookies - Just Verifying or Also Creating Cookies

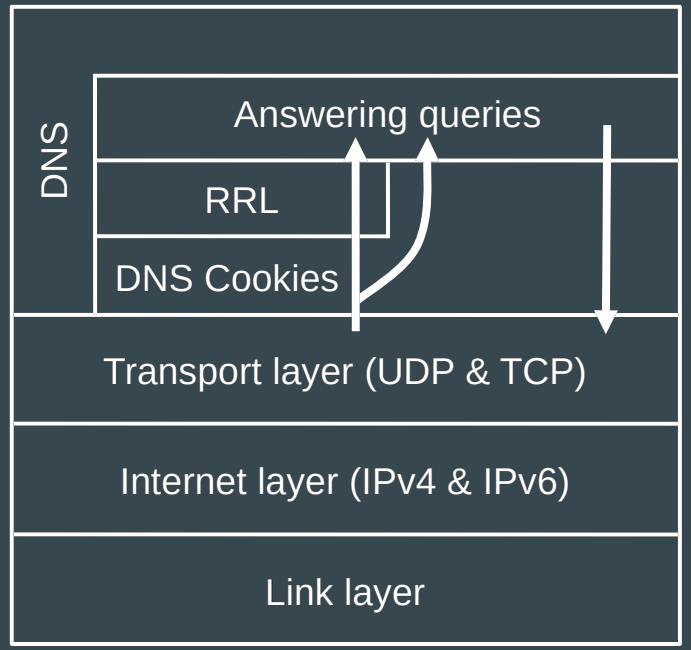




DNS Cookies - Also Creating Cookies … ongoing

- Outgoing eBPF on Traffic Control (TC) layer
- Edit Socket Buffer instead of packet
- Can grow with:
  - bpf_skb_change_tail()
- Checksum recalculations with:
  - bpf_skb_store_bytes()
- Connect in with out with:
  - BPF_MAP_TYPE_LRU_HASH
- Outgoing less performant, but…

… Augmenting … Interoperable



DNS Cookies - lessons learned

More complex programs can be made (and verified!) by composing & combining smaller programs with bpf_tail_call()s

Meta data can be used to pass and share (limited) data between XDP/eBPF programs
(as long as they are all for incoming, or all for outgoing traffic)

Outgoing DNS answers can be associated with data from incoming DNS requests with Least Recently Used (LRU) Maps.

Verifying DNS Cookies with XDP **and** user space DNS software doing Cookie too, makes perfectly sense especially when doing RRL with XDP already

Taking over DNS Cookie handling completely introduces process dependencies

Concluding ...

Programming eBPF is fighting the verifier, but…

A lot is possible!

XDP and eBPF is a very good fit for plain old UDP based DNS.
because per packet processing.

Less suitable for TCP based DNS, and probably impossible for DoT and DoH

We think using XDP to augment an existing DNS service is an exciting new idea, and a great new tool in the DNS operator’s toolbox

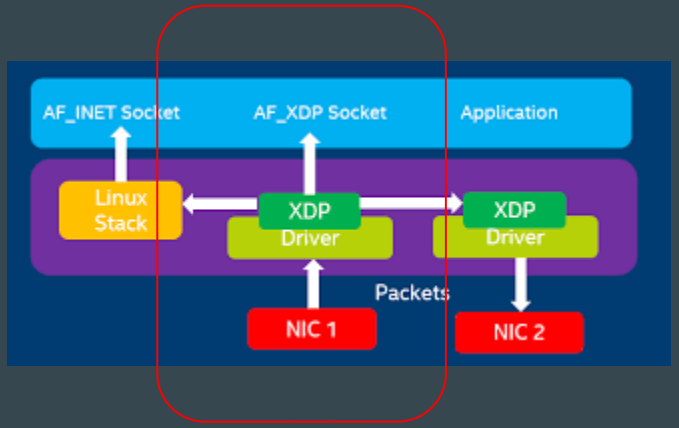
Ongoing work

Currently investigating offloading to actual hardware (Netronome SmartNICs);

This means we can dive into performance measurements, but also performance comparisons (kernel vs hardware offload);

Looking ahead

- **AF_XDP support for NSD**
  Adapt NSD to use the AF_XDP socket type provided by BPF/XDP
- **Hot self-managing cache**
  Write outgoing answers in a LRU hashmap, answer queries directly from XDP
- **Zone sharding / load balancing**
  Load balance based on the qname, so that nameservers only have to load part of (big) zones.
- **root zone from XDP?**



XDPeriments: Tinkering with DNS and XDP

...

{willem,luuk}@nlnetlabs.nl

https://github.com/NLnetLabs/XDPeriments

https://blog.nlnetlabs.nl/tag/research/