DNS Response Latency Monitoring

Martina Callea
Student ID: 0352145

University of Rome Tor Vergata

ICT and Internet Engineering
Academic Year 2024/2025

Indice

1 Basic Task

1.1 Goal

The goal of the **Basic Task** is to implement an eBPF program attached via XDP that intercepts outgoing DNS queries in a virtual topology.
The program must:

- operate both on **IPv4** and **IPv6** traffic;
- identify DNS queries by checking the **UDP destination port 53**;
- extract the **source IP address** and the **DNS transaction ID**;
- store a **timestamp** in a BPF hash map `dns_query_ts_map`, using a key composed of:
  - IP version;
  - DNS ID;
  - source IP address (IPv4 or IPv6).

1.2 Implementation

The eBPF program `xdp_dns_latency` was written in C using the `libbpf` framework. It was compiled using Clang with BPF target support, relying on the Linux kernel's BTF information (`vmlinux.h`) to ensure compatibility with kernel data structures.
The resulting object file `netprog.bpf.o` was built in the container, outside the VM, and then copied into the target virtual machine.

The eBPF program attaches to a network interface via XDP and intercepts packets at the earliest stage in the kernel. Its purpose is to identify outgoing DNS queries (both IPv4 and IPv6) and store a timestamp in a BPF hash map.

1.2.1 Map Definition

The following block defines a BPF hash map called `dns_query_ts_map`, which uses a composite key `dns_query_key`, defined in the header file `netprog.h`. It includes fields such as the IP version, DNS transaction ID, and source IP address (IPv4 or IPv6), and is used both at compile-time and by user-space tools to correctly interpret the map content. The value is a `__u64` timestamp.

```
struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct dns_query_key);
    __type(value, __u64);
    __uint(max_entries, 1024);
} dns_query_ts_map SEC(".maps");
```

1.2.2 XDP Entry Point

The main function uses the `SEC("xdp")` section and is executed for every packet received on the interface.

```
SEC("xdp")
int xdp_dns_latency(struct xdp_md *ctx) {
```

Inside it, we extract the pointers to the beginning and end of the packet data:

```
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
```

1.2.3 Parsing Ethernet Header

The Ethernet header is parsed and bounds-checked:

```
    struct ethhdr *eth = data;
    if ((void *)(eth + 1) > data_end)
        return XDP_PASS;
```

1.2.4 Protocol dispatch

We read the protocol field and use it to decide whether we are dealing with IPv4 or IPv6:

```
__u16 h_proto = bpf_ntohs(eth->h_proto);
void *nh = data + sizeof(*eth);
```

1.2.5 IPv4 branch

If the packet is IPv4 and UDP, we parse the headers and check for DNS destination port (53):

```
if (h_proto == ETH_P_IP) {
    struct iphdr *ip = nh;
    if ((void *)(ip + 1) > data_end || ip->protocol != IPPROTO_UDP)
        return XDP_PASS;

    struct udphdr *udp = (void *)(ip + 1);
    if ((void *)(udp + 1) > data_end)
        return XDP_PASS;

    if (bpf_ntohs(udp->dest) != DNS_PORT)
        return XDP_PASS;
```

The DNS header is checked to ensure it contains at least 2 bytes (DNS ID):

```
    __u8 *dns = (void *)(udp + 1);
    if (dns + 2 > (unsigned char *)data_end)
        return XDP_PASS;
```

Finally, we build the key and update the map:

```
    struct dns_query_key key = {};
    key.ip_version = 4;
    key.dns_id = ((__u16)dns[0] << 8) | dns[1];
    key.src_ip4 = ip->saddr;

    __u64 ts = bpf_ktime_get_ns();
    bpf_map_update_elem(&dns_query_ts_map, &key, &ts, BPF_ANY);
    return XDP_PASS;
}
```

1.2.6 IPv6 branch

If the packet is IPv6, the structure and logic are similar. The IPv6 header and UDP header are parsed, and the DNS packet is handled the same way:

```
    else if (h_proto == ETH_P_IPV6) {
        struct ipv6hdr *ip6 = nh;
        if ((void *)(ip6 + 1) > data_end || ip6->nexthdr != IPPROTO_UDP)
            return XDP_PASS;

        struct udphdr *udp = (void *)(ip6 + 1);
        if ((void *)(udp + 1) > data_end)
            return XDP_PASS;

        if (bpf_ntohs(udp->dest) != DNS_PORT)
            return XDP_PASS;

        __u8 *dns = (void *)(udp + 1);
        if (dns + 2 > (unsigned char *)data_end)
            return XDP_PASS;

        struct dns_query_key key = {};
        key.ip_version = 6;
        key.dns_id = ((__u16)dns[0] << 8) | dns[1];
        __builtin_memcpy(&key.src_ip6, &ip6->saddr, 16);

        __u64 ts = bpf_ktime_get_ns();
        bpf_map_update_elem(&dns_query_ts_map, &key, &ts, BPF_ANY);
        return XDP_PASS;
    }
```

In all other cases, the packet is allowed to pass through:

```
    return XDP_PASS;
}
```

1.3 Testing

The eBPF program was tested inside a virtual network environment created within the VM.
The environment was set up using a Bash script (`xdp_icmpv6_drop.sh`), which:

- creates three Linux network namespaces: `h0`, `r0`, and `h1`;
- connects them using `veth` pairs to simulate two end-hosts and a central router;
- assigns IPv4 and IPv6 addresses to each interface;
- mounts a BPF filesystem and attaches the compiled eBPF object file `netprog.bpf.o` to one or more interfaces of `r0` via XDP;
- opens a `tmux` session with three windows, one per namespace, for interactive testing.

To test the eBPF program, DNS queries were manually issued from namespace `h0` towards `h1`.
The following IP configuration was used:

- `h0`: `10.0.0.1` (IPv4), `cafe::1` (IPv6)
- `h1`: `10.0.2.1` (IPv4), `beef::1` (IPv6)

The queries were sent from `h0` using the `dig` command, targeting both IPv4 and IPv6 addresses of `h1`.
Example commands:

```
dig @10.0.2.1 google.com
dig -6 @beef::1 google.com
```

Although `h1` was not running a DNS server (hence returning "connection refused"), this was not an issue for the purpose of the basic task. The goal was to ensure that outgoing DNS queries were visible at the router `r0`, where the eBPF program was attached.
The program successfully intercepted these packets, parsed their DNS ID and source IP, and stored a timestamp in the BPF map. The entries were later inspected using `bpftool` from inside `r0`.
After dumping the contents of the BPF map to a file:

```
bpftool map dump pinned /sys/fs/bpf/netprog/maps/dns_query_ts_map -j > map.txt
```

a Python script was executed to convert the binary map content into a human-readable format. The script decodes IP addresses (both IPv4 and IPv6), DNS transaction IDs, and the associated timestamps.

```
python3 parse_map.py
```

The output confirmed that the eBPF program correctly identified and recorded the DNS queries:

```
DNS QUERY TIMESTAMPS (from map.txt)

- IP:                              cafe::1 | ID: 43775 | Time (ns): 7955700169173
- IP:                            10.0.0.1 | ID: 20289 | Time (ns): 7958690975366
```

This demonstrates that both IPv6 and IPv4 DNS queries originating from `h0` were successfully intercepted and logged by the eBPF program.
The code used to parse the map entries can be found in the next page.

```python
#!/usr/bin/env python3

import json
import socket
import ipaddress
from pathlib import Path

def parse_entry(entry):
    try:
        key = entry["formatted"]["key"]
        value = entry["formatted"]["value"]
    except KeyError:
        return None  # salta se manca "formatted"

    ip_version = key.get("ip_version", "?")
    dns_id = key.get("dns_id", "?")

    if ip_version == 4:
        ip_raw = key["src_ip4"]
        ip = socket.inet_ntoa(int(ip_raw).to_bytes(4, "little"))
    elif ip_version == 6:
        ip6_bytes = bytes(key["src_ip6"])
        ip = str(ipaddress.IPv6Address(ip6_bytes))
    else:
        ip = "UNKNOWN"

    return f"- IP: {ip:<39} | ID: {dns_id:<5} | Time (ns): {value}"

def main():
    path = Path("map.txt")
    if not path.exists():
        print("File 'map.txt' not found.")
        return

    with open(path) as f:
        data = json.load(f)

    print("\n DNS QUERY TIMESTAMPS (from map.txt)\n")
    for entry in data:
        line = parse_entry(entry)
```

```
        if line:
            print(line)

if __name__ == "__main__":
    main()
```

2 Intermediate Task

2.1 Goal

The goal of the Intermediate Task is to extend the eBPF program developed in the Basic Task to measure the Round-Trip Time (RTT) of DNS queries. This must be done in kernel space by:

- saving the timestamp of outgoing DNS queries,
- matching the corresponding DNS responses,
- computing the RTT,
- storing the result in a hash map accessible from user-space.

2.2 Implementation

2.2.1 BPF Maps

This eBPF program uses two BPF maps.
The first map stores timestamps for outgoing DNS queries.

```
struct {
__uint(type, BPF_MAP_TYPE_HASH);
__type(key, struct dns_query_key);
__type(value, __u64);
__uint(max_entries, 1024);
} dns_query_ts_map SEC(".maps");
```

The second map stores the computed Round-Trip Time between DNS query and response.

```
struct {
__uint(type, BPF_MAP_TYPE_HASH);
__type(key, struct dns_query_key);
__type(value, __u64);
__uint(max_entries, 1024);
} dns_rtt_map SEC(".maps");
```

2.2.2 DNS Packet Handling

The core logic is implemented in an inline function that constructs the key and either stores the timestamp or computes RTT depending on whether the packet is a DNS query or response.

```
static __always_inline int handle_dns_packet(...)
```

2.2.3 Protocol and DNS Parsing

The main function attached via XDP first extracts the protocol:

```
__u16 h_proto = bpf_ntohs(eth->h_proto);
```

Then it verifies bounds and processes IPv4. The DNS ID is extracted, and the QR bit (bit 7 of the DNS flags) is used to determine if the packet is a query (0) or a response (1). Based on this, the client IP address is selected: src for queries and dst for responses. This ensures that the same key is used in both directions to correctly match query and response pairs. The same logic is repeated for IPv6 with adjusted header types.

```
if (h_proto == ETH_P_IP) {
...
if (ip->protocol != IPPROTO_UDP)
return XDP_PASS;
...
__u8 *dns = (void *)(udp + 1);
if ((void *)(dns + 12) > data_end)
return XDP_PASS;
__u16 dns_id = ((__u16)dns[0] << 8) | dns[1];
__u8 flags = dns[2];
__u8 is_response = flags >> 7; // QR bit: 0 = query, 1 = response
void *ip_ptr = is_response ? &ip->daddr : &ip->saddr; //srcIP for queries, dstIP for resp
return handle_dns_packet(dns, data_end, 4, dns_id, ip_ptr, is_response);
}
```

2.3 Testing

The eBPF program was tested in the virtual environment previously described, using `h0` as a DNS client and `h1` as a DNS server (via `dnsmasq`).
For IPv4:

- In `h1`, the DNS server was launched using:

  ```
  ip netns exec h1 dnsmasq --no-daemon --no-resolv --listen-address=10.0.2.1
  ```

- In `h0`, the server was queried with:

  ```
  dig @10.0.2.1 google.com
  ```

The RTTs were captured by the eBPF program and stored in the map `dns_rtt_map`.
From `r0`, the map was exported using:

```
bpftool map dump pinned /sys/fs/bpf/netprog/maps/dns_rtt_map -j -p > map.json
```

Finally, the file was parsed using a Python script to convert binary IP addresses and display readable output, confirming the correct matching and measurement of DNS RTTs.

```
DNS RTT Results

- IP: 10.0.0.1                                  | ID: 64075 | RTT: 1.584 ms
- IP: 10.0.0.1                                  | ID: 16559 | RTT: 8.389 ms
- IP: 10.0.0.1                                  | ID: 7160  | RTT: 1.003 ms
```

For IPv6:

- In `h1`, the DNS server was launched using:

  ```
  ip netns exec h1 dnsmasq --no-daemon --no-resolv --listen-address=beef::1
  ```

- In `h0`, the server was queried with:

  ```
  dig -6 @beef::1 google.com
  ```

The RTT map was retrieved and parsed in the same way, confirming the successful measurement of IPv6 queries as well.

```
DNS RTT Results

- IP: ca:fe:00:00:00:00:00:00:00:00:00:00:00:00:00:01 | ID: 55947 | RTT: 2.076 ms
- IP: ca:fe:00:00:00:00:00:00:00:00:00:00:00:00:00:01 | ID: 52631 | RTT: 1.054 ms
- IP: ca:fe:00:00:00:00:00:00:00:00:00:00:00:00:00:01 | ID: 57701 | RTT: 11.163 ms
```

The following Python script was used to parse the JSON output from the map and display the results in a human-readable format. It converts the binary IP addresses from little-endian format for IPv4 and formats IPv6 addresses correctly.

```
import json
import socket
from pathlib import Path

def ip_from_u32_le(val):
    return socket.inet_ntoa(val.to_bytes(4, 'little'))

def main():
    path = Path("map.json")
    if not path.exists():
        print("File 'map.json' non trovato.")
        return

    with open(path) as f:
        data = json.load(f)

    print("\nDNS RTT Results\n")
    found = False
    for entry in data:
        try:
            key = entry["formatted"]["key"]
            value = entry["formatted"]["value"]
        except KeyError:
            continue  # skip if not formatted

        ip_version = key.get("ip_version", "?")
        dns_id = key.get("dns_id", "?")
        rtt_ns = value

        if ip_version == 4:
            ip = ip_from_u32_le(key["src_ip4"])
        elif ip_version == 6:
            ip = ":".join(f"{b:02x}" for b in key["src_ip6"])
        else:
```

```
            ip = "?"

        print(f"- IP: {ip:<39} | ID: {dns_id:<5} | RTT: {rtt_ns / 1e6:.3f} ms")
        found = True

    if not found:
        print("Nessun dato 'formatted' trovato in map.json.")

if __name__ == "__main__":
    main()
```

3 Advanced Task

3.1 Goal

The goal of the Advanced Task is to implement an eBPF program that measures DNS query-response round-trip time (RTT) and maintains a histogram of latencies directly in kernel space. The values are stored in a power-of-two scaled histogram (log2). Only valid DNS exchanges with RTT below a threshold (0.5 seconds) are considered. The histogram is then exported through bpffs for post-processing.

3.2 Implementation

3.2.1 Map and Constant Definitions

The program uses two BPF maps:

- `dns_query_ts_map`: a hash map that stores timestamps for outgoing DNS queries.
- `rtt_histogram`: an array map that aggregates the count of observed RTTs into buckets of increasing powers of two.

```
#define DNS_PORT 53
#define MAX_BUCKETS 30
#define MAX_RTT_NS 500000000ULL

struct {
    __uint(type, BPF_MAP_TYPE_HASH);
    __type(key, struct dns_query_key);
    __type(value, __u64);
    __uint(max_entries, 1024);
} dns_query_ts_map SEC(".maps");

struct {
    __uint(type, BPF_MAP_TYPE_ARRAY);
    __type(key, __u32);
    __type(value, __u64);
    __uint(max_entries, MAX_BUCKETS);
} rtt_histogram SEC(".maps");
```

3.2.2 Log-Scaled Buckets

A log2 scale is used for the histogram to provide higher resolution for small RTT values while compressing larger values into fewer buckets. This approach is suitable for latency distributions, which are typically skewed toward small values. It enables detailed analysis of typical behavior and maintains a compact and efficient representation, even when occasional slow packets are observed.
The `log2_bucket()` function calculates the power-of-two bucket for a given RTT:

```
static __always_inline __u32 log2_bucket(__u64 latency_ns) {
    __u32 i = 0;
    latency_ns >>= 1;
    while (latency_ns > 0 && i < MAX_BUCKETS - 1) {
        latency_ns >>= 1;
        i++;
    }
    return i;
}
```

3.2.3 DNS Packet Handling

The core logic is implemented in `handle_dns_packet()`, which stores or compares timestamps and updates the histogram if the latency is below the threshold:

```
static __always_inline int handle_dns_packet(...) {
    ...
    if (!is_response) {
        bpf_map_update_elem(...);
    } else {
        __u64 *tsp = bpf_map_lookup_elem(...);
        if (tsp) {
            __u64 delta = now - *tsp;
            if (delta <= MAX_RTT_NS) {
                __u32 bucket = log2_bucket(delta);
                __u64 *count = bpf_map_lookup_elem(...);
                if (count)
                    __sync_fetch_and_add(count, 1);
            }
            bpf_map_delete_elem(...);
        }
    }
    return XDP_PASS;}
```

3.2.4 XDP Program Logic

The main function `xdp_dns_latency()` extracts IP and UDP headers, verifies bounds, and calls `handle_dns_packet()` with the correct IP and DNS ID, distinguishing between IPv4 and IPv6:

```
SEC("xdp")
int xdp_dns_latency(struct xdp_md *ctx) {
    void *data = (void *)(long)ctx->data;
    void *data_end = (void *)(long)ctx->data_end;
    struct ethhdr *eth = data;
    if ((void *)(eth + 1) > data_end)
        return XDP_PASS;

    __u16 h_proto = bpf_ntohs(eth->h_proto);
    void *nh = data + sizeof(*eth);

    if (h_proto == ETH_P_IP) {
        struct iphdr *ip = nh;
        ...
        void *ip_ptr = is_response ? &ip->daddr : &ip->saddr;
        return handle_dns_packet(...);
    } else if (h_proto == ETH_P_IPV6) {
        struct ipv6hdr *ip6 = nh;
        ...
        void *ip_ptr = is_response ? &ip6->daddr : &ip6->saddr;
        return handle_dns_packet(...);
    }
    return XDP_PASS;
}
```

3.3 Testing

The testing methodology for the Advanced Task reuses the same virtual environment and procedure described in the Intermediate Task. In particular, `h0` acts as the DNS client and `h1` runs a minimal DNS server via `dnsmasq`, supporting both IPv4 and IPv6 queries.
The DNS queries are issued using `dig`, and the eBPF program monitors the traffic via XDP to track query-response pairs. Unlike the Intermediate Task, instead of storing individual RTTs, the program updates an in-kernel histogram based on the observed latency (using a `BPF_MAP_TYPE_ARRAY` map).
The resulting histogram was exported from the pinned map `rtt_histogram` using:

```
bpftool map dump pinned /sys/fs/bpf/netprog/maps/rtt_histogram -j -p > hist.json
```

The JSON output was then parsed using the following Python script, which reconstructs the power-of-two latency buckets and displays a readable histogram of measured DNS latencies.
In the first test, 4 IPv4 DNS packets were sent. The output shows the distribution of the measured latencies for each packet, grouped into their respective logarithmic buckets.

```
(...)
0.524 ms - 1.049 ms : 1 packet(s)
1.049 ms - 2.097 ms : 2 packet(s)
2.097 ms - 4.194 ms : 0 packet(s)
4.194 ms - 8.389 ms : 1 packet(s)
(...)
```

In the second test, 4 IPv6 DNS packets were sent, and the corresponding histogram of the observed latencies is shown below, again grouped into their respective logarithmic buckets.

```
(...)
1.049 ms - 2.097 ms : 2 packet(s)
2.097 ms - 4.194 ms : 1 packet(s)
4.194 ms - 8.389 ms : 0 packet(s)
8.389 ms - 16.777 ms : 1 packet(s)
(...)
```

The following Python code parses the exported histogram and prints a human-readable summary of the measured DNS latencies:

```python
import json

def bucket_ns_to_ms(index):
    """Converte il bucket log2 in un intervallo approssimato di latenza in ms"""
    low = 1 << index
    high = (1 << (index + 1)) - 1
    return low / 1e6, high / 1e6

with open("hist.json") as f:
    data = json.load(f)

print("\nDNS RTT Histogram\n")
for entry in data:
    idx = int(entry["key"][0], 0)
    count = int(entry["value"][0], 0)
    low, high = bucket_ns_to_ms(idx)
    print(f"{low:.3f} ms - {high:.3f} ms : {count} packet(s)")
```