Operating System-Level Load Distribution for Network Telemetry Data Collection

Master's Thesis

Eduard Bachmakov

Tutor: Tobias Bühler (ETHZ), Thomas Graf (Swisscom)

Supervisor: Prof. Dr. Laurent Vanbever

March–September 2021

Acknowledgments

I would like to thank all the numerous people for their direct or indirect contributions to this thesis.

First of all, thank you Thomas, for your mentoring, sharing of your extensive expertise & experience, and for consistently providing valuable feedback. Thank you to Marco and *all* the others on the Swisscom side for your continued advice and support.

On the INSA Lyon side, thank you, Pierre and Alex, especially for joining me on countless deep dives into various internals.

Finally, on the ETH end, many thanks to Tobias for the tutoring and to Laurent for offering this opportunity.

Abstract

As networks grow larger, network telemetry data collection becomes both more of a necessity as well as a challenge scaling with the extent of the network. In order to distribute network telemetry traffic among network telemetry collection endpoints, multiple layers of load balancing are used. Targeting the last layer—from the network interface to the collecting processes—we designed and implemented a system tailored to addressing practical challenges of running such a collection system while surpassing the existing system in both usability and performance.

Contents

1 Introduction & Problem Statement **1**

2 Background **4**

2.1 Linux Packet Path & Connection Establishment 4

2.2 The SO_REUSEPORT Socket Option 5

2.3 eBPF 8

2.3.1 Overview 8

2.3.2 Program Capabilities 9

2.3.3 BPF API vs Kernel API 9

2.3.4 Development and Deployment Pipeline 10

2.3.5 *Libbpf* 11

2.3.6 BPF CO-RE 11

2.4 Related Work 12

3 Network Environment & System Setup **13**

3.1 Network Environment 13

3.2 Dataflow 14

3.3 Components 17

3.3.1 *HAProxy* 18

3.3.2 *Pmacct* 18

3.3.3 Remaining Components 19

4 Design **20**

4.1 Requirements 20

4.2 Overview 22

4.2.1 Kernel vs BPF 25
4.3 Detailed Design 25
4.3.1 Kernel 26
4.3.2 Interface with Userspace 28
4.3.3 Userspace 29
4.4 Behavioral Aspects 31
4.5 End-to-End 31

5 Evaluation **36**
5.1 Performance 39

6 Outlook & Future Work **43**

7 Summary **45**

Bibliography **45**

A Looking closer at *nfacctd* **I**
A.1 Execution of a BPF Userspace program I

B Looking *closely* at *reuseport_kern* **VI**
B.1 Intermediate Representation VI
B.2 JIT-Compiled Program XI

C BMP Session Handoff Challenges **XVII**

Chapter 1

Introduction & Problem Statement

Large network operators such as Internet Service Providers (ISPs) or data center companies manage networks with a large number of network devices.

These devices may be of various makes and models and run a variety of network protocols. Yet, they collectively maintain distributed state required for even basic operation. Visibility into this state is essential for the allocation of capacity, fault detection & recovery, and maintenance of business continuity.

Due to the heterogeneous nature of devices in operation, the operator can only rely on standardized network telemetry protocols [48]. However, despite standardization, it is up to the vendor as to which standards to implement, at which version, and at which level of compliance.[1] The protocols span three different perspectives of the network:

Control Plane Most prominently: routing. For example, the BGP Monitoring Protocol (BMP) exports internal BGP RIB and peering state, enriched with peering, routing instance, and route-policy metadata. This protocol uses TCP as its transport layer. [45]

Forwarding Plane Flow monitoring protocols such as IPFIX provide summarized accounting data on traffic volume with data plane and network processor dimensions. This protocol uses UDP as its transport layer. [8, 19]

[1] ... and how much to charge for it.

Topology Device properties such as, for example, an enumeration of its interfaces and their current state, can be described using the YANG modeling language, according to YANG schema stored on-device in YANG datastores, and pushed using Netconf or YANG-Push. Netconf uses TCP as its transport layer. YANG-Push can use TCP or UDP for its transport layer: UDP for transporting best-effort accounting data and TCP for transporting essential state changes. [9]

Combining this data allows an operator to create near-realtime inferences on the state of the network as a whole as well as on particular aspects of interest. For example, augmenting forwarding plane information with BMP-derived information on BGP communities allows mapping traffic to services, such as identifying specific MPLS traffic as a specific customer's VPN traffic.

These protocols typically push data via TCP or UDP to a data collection system. A network telemetry protocol-dependent combination of collector daemons is responsible for accepting, validating, filtering, post-processing, and forwarding the data into a storage system that is shared among all the collector daemons. This storage system can serve as a data source for an online analytical processing (OLAP) frontend used by a human operator. Additionally, it may serve as a uniform data model for automation to be built upon.

By definition, the network devices feeding the collector daemons are both numerous and omnipresent throughout the network. With dynamic traffic allocation/shaping, it is safe to assume that any subset of them is involved in some flow at some point. Therefore, broad observability can only be achieved by monitoring the maximal number of devices. In other words, all devices are expected to push network telemetry. While, conceptually, it only takes a single collector daemon to collect and organize the incoming data, doing so poses both scalability and reliability risks, such as overload, lack of resiliency, network partitions. Thus, without any levels of indirection, this is a many-to-many actor problem.

One ISP seeking network telemetry protocol spanning and cross-perspective observability by using this approach is the Swiss national telecommunications provider *Swisscom*. This thesis is based on a collaboration of the author with Swisscom's Network Analytics Architect Thomas Graf.

In the context of the previously introduced challenges, Swisscom uses multiple load balancing layers to provide an even distribution of load to collector daemons:

- Anycast, the advertisement of a set of virtual IP addresses (VIPs)

from multiple, distinct locations in the network, provides a high-level, geographical partitioning of the network telemetry input streams.

- Equal-cost multi-path (ECMP) routing balances input streams within a single advertisement region across the available paths from the originating network device to the VIP.
- On-host balancing distributes the load of receiving and processing the packet contents among a pool of processes/threads.

This thesis will focus primarily on the on-host load balancing aspect.

After this introductory chapter, chapter 2 describes certain aspects of the Linux networking stack implementation as well as the extended Berkeley Packet Filter and its ecosystem necessary to understand some of the core differentiators of the design. In chapter 3 we introduce both the existing setup used for network telemetry data collection and the environment in which it is running. Chapter 4 describes the challenges to be addressed as well as the design chosen and implemented, while chapter 5 determines its effectiveness. Competing efforts and steps not (yet) taken make up chapter 6. Finally, an overall summary concludes the core work in chapter 7.

Chapter 2

Background

The final design (chapter 4) uses some relatively new or less commonly known functionalities of the Linux networking stack. Here, we present a refresher on how the Linux kernel processes packets, focused on how new TCP connections are established. Subsequently, we elaborate on a specific option that modifies this flow. Finally, we conclude with an introduction to the extended Berkeley Packet Filter, the core technology enabling our use case.

These concepts are presented mostly independently in this chapter, however, the design will tie them together.

All references to source files in this and in later chapters are relative to the Git repository *torvalds/linux.git* at tag *v5.12* [54].[1]

2.1 Linux Packet Path & Connection Establishment

While one could argue that the receive path starts with the device interrupt or within the device-specific driver logic, for most intents and purposes, it is more useful to visualize the receive path starting at the more generic abstractions as illustrated in in figure 2.1. Netfilter hooks, which many Linux users are familiar with, serve as a mental guide in correlating the internal pipeline stages with the "public API."

The socket lookup step will be of particular interest. For IPv4 it comes into play after the routing decision taken within `ip_rcv_finish()`[2] has determined to locally deliver the packet. `ip_local_deliver()`[3] passes it off to the higher

[1] Specifically: commit 9f4ad9e425a1d3b6a34617b8ea226d56a119a717.
[2] net/ipv4/ip_input.c:415
[3] net/ipv4/ip_input.c:240

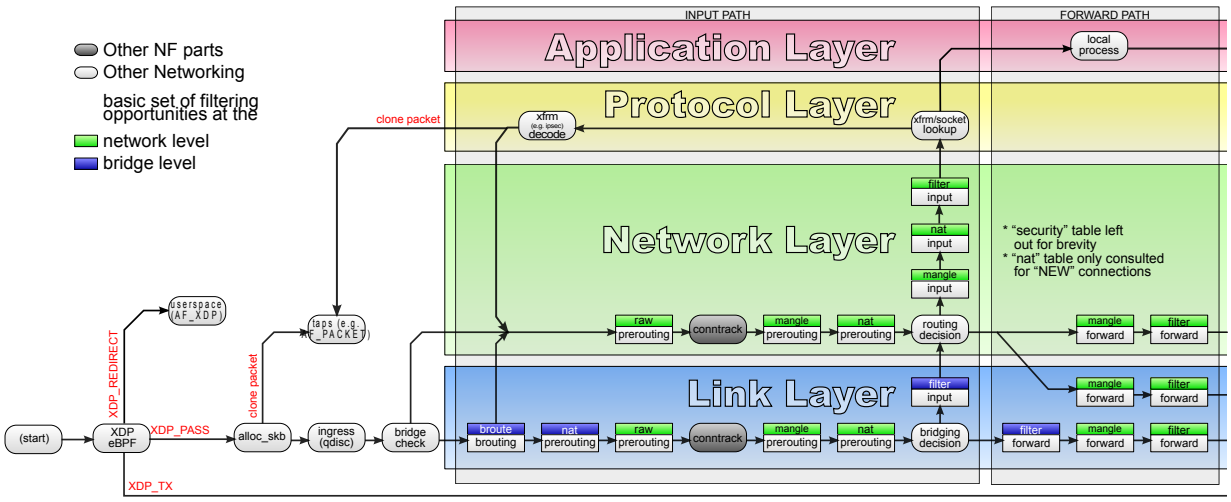

Figure 2.1: Input and forward path of network packets in the Linux kernel. Adapted from [15].

layer, e.g., `tcp_v4_rcv()`[4]. The call chain for the TCP socket lookup starting at that point for both IP protocols is shown in figure 2.2.

2.2 The SO_REUSEPORT Socket Option

Linux's implementation of the POSIX sockets API allows the use of *socket options* (set via the setsockopt(2) system call [33]) to modify the default behavior offered by the kernel. Starting with version 3.9, the kernel supports the SO_REUSEPORT option [34, 25].

When enabled, this allows the calling process's user ID to create multiple sockets bound to the same combination of IP address and TCP/UDP port. One software product advertising its use for throughput improvements and latency reduction is the NGINX HTTP server [21]. Some of the performance gains of using the option (as opposed to multiple threads accepting on a shared socket) stem from increased parallelism due to lack of lock contention during socket/file descriptor allocation and within the accept queue operations (see `inet_csk_accept()`[5]).

[4] net/ipv4/tcp_ipv4.c:1924
[5] net/ipv4/inet_connection_sock.c:463

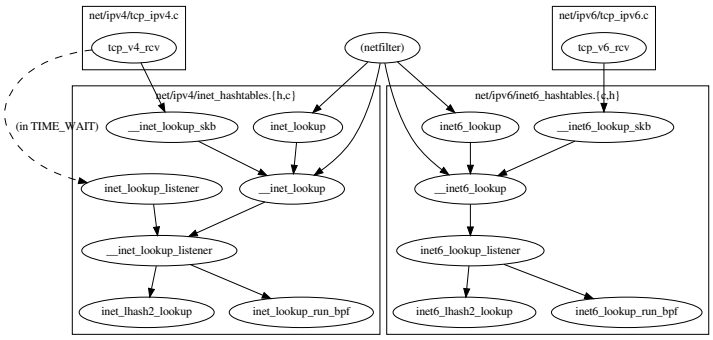

Figure 2.2: Function call hierarchy during TCP socket lookup with source file location. The common path is to look up the socket in (IP, port)-indexed hashmaps for every new TCP segment received. Specific types of eBPF programs can hook into this lookup. Notation: caller → callee.

Under the hood,

- in the case of TCP transport, each incoming connection is assigned a SO_REUSEPORT enabled socket only once during connection establishment while
- in the case of UDP transport, each incoming datagram is assigned to a SO_REUSEPORT enabled socket whenever it is received—statelessly.

The actual assignment decision is based on a hashed 5-tuple of the incoming packet.[6] This socket option extends the regular socket lookup call chain as depicted in figure 2.3. Three new paths are added:

1. Pure SO_REUSEPORT based selection selection as described above.
2. Opportunistic selection using a classical BPF program, which falls back on #1 if unsuccessful [34].
3. Exact selection using an extended BPF program.

[6] net/core/sock_reuseport.c:288

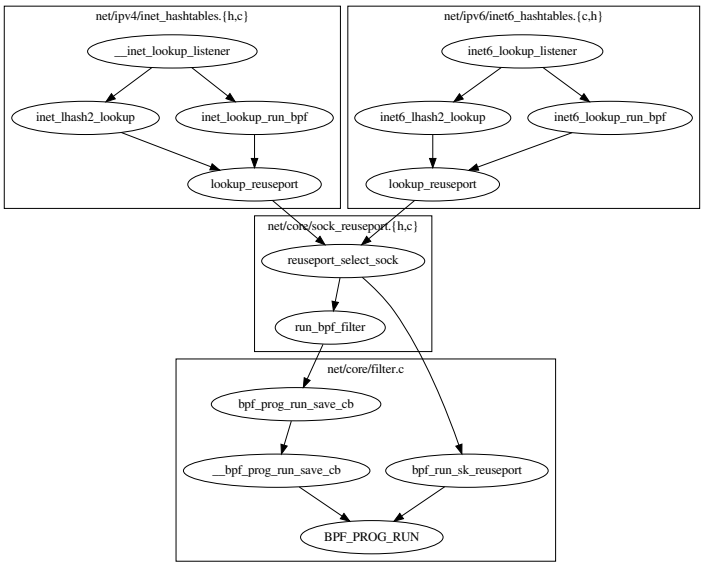

Figure 2.3: Function call hierarchy during TCP socket lookup with enabled SO_REUSEPORT socket option. Continuation of figure 2.2. `run_bpf_filter()` is the entry point into a classical BPF program path while `bpf_run_sk_reuseport()` is the extended BPF equivalent. Notation: caller → callee.

2.3 eBPF

The extended Berkeley Packet Filter (eBPF) is a Linux kernel subsystem allowing developers and administrators to create programs that attach at pre-defined hooks in the kernel. At the lowest level, these programs are strings of bytecode instructions targeting the eBPF virtual machine. The restricted instruction set, the constrained execution environment, and the in-kernel eBPF Verifier effectively create a sandboxed environment for safe execution.

eBPF was developed as an extension of the Berkeley Packet Filter (BPF) [36] in 2014 [12]. In newer literature, "eBPF" is also used to refer to the ecosystem as a whole.

A note on terminology: The term "BPF" will refer to the extended Berkeley Packet Filter exclusively. When a distinction becomes necessary it will referred to as "eBPF" while the classical Berkeley Packet Filter will be called "cBPF".

2.3.1 Overview

A BPF program is a collection of logic that can be attached to specific points in the kernel's control flow. Many hooks covering many use cases are available. These include traffic classification on qdiscs, function tracing on syscall/kernel function entry, and socket selection on the networking receive path.

Conceptually, a BPF program can be thought of as a function with a single argument, the context. At the time of its invocation, there is no other state (such as global variables). All a program can do is operate on the data provided in the context, use immediate values, and call BPF helpers.

BPF helpers are functions specifically created for use by BPF programs. They cover a broad collection of functionality, including obtaining random values, invalidating hashes on socket buffers, setting socket options, obtaining user/kernel stacks, or sending messages to userspace via DebugFS [31].

BPF programs keep state by utilizing BPF maps. These maps form a family of in-kernel data structures, ranging from generic arrays and hash maps to attachment point-specific, custom data structures that can be read from and written to using specific BPF helpers. These maps are also accessible from userspace via file descriptors. Thus, they form the primary means of communication between a live BPF program and the corresponding userspace peer.

BPF programs and maps are reference-counted and deleted once no longer used. Persistence and sharing can be achieved by *pinning* them to a path on BPFFS ("BPF File System"), thereby creating pseudo-files. These can be used by other processes to obtain references to the BPF objects.

To ensure safety and security, BPF programs are checked by the BPF Verifier at load time. As one part of its duties, it ensures memory safety, limits instruction count, and verifies guaranteed termination [37]. Specific examples illustrating verification and its challenges regarding safety are presented in [50].

2.3.2 Program Capabilities

A BPF program is associated with a single BPF Program Type. The program type determines multiple aspects of execution. These include

- the set of BPF instructions/opcodes the program is allowed to use. This is verified by the BPF Verifier during loading.
- the set of BPF helpers the program is allowed to call. This is verified by the BPF Verifier during loading.
- the points at which the program can be attached. This is verified in the context of the system call attaching the program.

For example, a SK_REUSEPORT program cannot directly access socket buffer ("skb") data via the BPF_LD_ABS and BPF_LD_IND BPF instructions (representing direct and indirect packet access) while a program of type SOCKET_FILTER can do so.[7] A concise description of a selected set of BPF program types is available in [16].

Kernel 5.12.0 supports 30 different BPF Program Types.[8]

2.3.3 BPF API vs Kernel API

It is often desirable to expose only a limited subset of kernel-internal data structures to BPF programs to prevent dangerous modifications or information leaks. This is done via the use of per-program type context structures [49]. The following is an example of this mechanism regulating access to socket buffer ("skb") data.

[7] kernel/bpf/verifier.c:8367
[8] include/uapi/linux/bpf.h:177

```
const struct bpf_verifier_ops sk_filter_verifier_ops = {
        .get_func_proto         = sk_filter_func_proto,
        .is_valid_access        = sk_filter_is_valid_access,
        .convert_ctx_access     = bpf_convert_ctx_access,
        .gen_ld_abs             = bpf_gen_ld_abs,
};
```

Listing 2.1: Verifier options for SOCKET_FILTER BPF programs.

The BPF_PROG_TYPE macro ties `struct __sk_buff` on the BPF side to `struct sk_buff` as the kernel internal structure for SOCKET_FILTER programs.[9] Upon program loading, the BPF Verifier uses program type specific validation and rewriting functions to convert from one to the other. Listing 2.1 shows the actual mapping of these functions for our example program type.[10] Here, `sk_filter_is_valid_access()` is responsible for ensuring that the BPF program is allowed to access the remote port but disallowed from inspecting the timestamp. For accesses found valid, `bpf_convert_ctx_access()` "parses" access to the context and rewrites the addresses and/or offsets to ones which are valid in the backing `struct sk_buff`.

2.3.4 Development and Deployment Pipeline

Between conception and execution, a BPF program potentially traverses a multitude of stages:

1. Business logic is written in a higher-level language such as (a restricted version of) C.[11]
2. Using LLVM's *Clang* compiler, C code is compiled into LLVM Intermediate Representation (IR).
3. The LLVM BPF backend compiles this IR into BPF assembly or directly into an ELF object file.
4. A BPF loader reads the ELF object and creates the associated data structures on the running kernel via a series of calls to the bpf() system call. It also resolves references to these data structures within the actual program. [5]

[9] include/linux/bpf_types.h:5

[10] net/core/filter.c:9833

[11] Even higher level abstractions are possible as well. For example, bpftrace provides a custom AWK inspired language tailored to dynamic tracing [43].

5. The actual loading of the program, triggered by `bpf(BPF_PROG_LOAD,...)`, features a mandatory program verification step in which the in-kernel BPF Verifier establishes various safety properties such as termination, memory safety, etc. [32]

6. A program that passes the BPF Verifier may optionally be compiled just-in-time, targeting the underlying hardware architecture. [10]

7. A loaded program is represented by a file descriptor which then may be used to attach it to the respective hook.

2.3.5 *Libbpf*

libbpf is a C library developed as part of the Linux kernel source tree [55]. It provides an extensive API to represent BPF programs, maps, and other associated objects. It also implements a BPF program loader, including diagnostic self-tests and extensive error report and handling options.

2.3.6 BPF CO-RE

In Linux, a dynamic executable using shared libraries requires the dynamic loader to resolve symbol references and perform relocation. While this constitutes additional complexity, it does allow the decoupling of components. With the appropriate precautions for maintaining ABI compatibility, these components can evolve independently.

The BPF equivalent of this functionality is BPF CO-RE ("Compile Once—Run Everywhere") [39]. The core idea revolves around enabling portability of compiled BPF programs across kernel versions. With its fast pace of development and the immense configuration space, kernel compatibility is all but guaranteed. For example, memory layout changes may be introduced by a different kernel configuration enabling an optional feature which, in turn, causes the addition of feature-specific fields in commonly used structures. Listing 2.2 contains an illustration of this.

A combination of three components can address these situations: a kernel that is compiled with BTF information[12] encoding its data structures, the ELF binary that is compiled with BTF information symbolically expressing data access, and a BPF loader that is able to perform relocations using

[12]BPF Type Format (BTF) is an efficient format for debug metadata, comparable with DWARF.

```
1  struct sock {
2          struct sock_common      __sk_common;
3  [...]
4          int                     sk_forward_alloc;
5  #ifdef CONFIG_NET_RX_BUSY_POLL
6          unsigned int            sk_ll_usec;
7          unsigned int            sk_napi_id;
8  #endif
9          int                     sk_rcvbuf;
10 };
```

Listing 2.2: Example of the configuration dependent memory layout problem. If a hypothetical BPF accessing `sock->sk_rcvbuf` was compiled against a kernel configured with NET_RX_BUSY_POLL enabled, the relative offset would no longer be correct if run on kernel with NET_RX_BUSY_POLL disabled. In this case, the in-kernel verifier would reject the program at load time.

BTF information by resolving the latter using the former. This enables a large degree of kernel version-independence for deployment purposes for the compiled BPF ELF binary.

2.4 Related Work

In the proposed kernel patch *Socket migration for SO_REUSEPORT* Kuniyuki Iwashima proposes a mechanism that migrates established, but not yet accepted TCP connections to other SO_REUSEPORT-enabled sockets. [24]

Public prior art on the specific implementation chosen in chapter 4 appears to be absent, which proved a challenge during implementation.[13] However, the fact that this functionality was created and merged into the upstream Linux kernel itself is strong circumstantial evidence that these applications exist somewhere. One possible explanation is that the programs using this functionality are locked away in internal corporate code repositories.

More generic loadbalancers using BPF do exist and enjoy popularity in the "cloud native" space, with *Cillium* [23] being a prominent example.

[13] A search on sourcegraph.com, a service that provides usable code search for 1.2 million public Github repositories, finds no results for the regular expression "`\bsk_reuseport/\w`" when filtering out kernel-internal self-tests and libbpf support for this BPF program type. These strings correspond to ELF section prefixes that libbpf uses to determine the BPF program type, a common pattern. For "`\bsk_lookup/\w`," there is only one single result: part of a demo used during the 2020 eBPF Summit.

Chapter 3

Network Environment & System Setup

With the need for network telemetry data collection and aggregation established back in chapter 1, we can now proceed to look at the high-level constraints which a solution enabling such collection would need to obey. In this chapter, we describe the network environment in which we operate, the data collection flow, and the setup of the existing prototype systems.

3.1 Network Environment

Swisscom maintains an internal, isolated network segment called the *Swisscom IETF interoperability lab*, depicted visually in figure 3.1. The topology of this environment is explained in great detail in [46]. For our purposes, we focus on the node labeled *ietf-internal*. It acts as a host for network telemetry data collection and is already configured as the target endpoint in all applicable network devices within the lab for both the BMP and IPFIX protocols.

From a practical perspective, this means there is a steady stream of TCP connection requests and UDP datagrams aimed at this host at any given time.

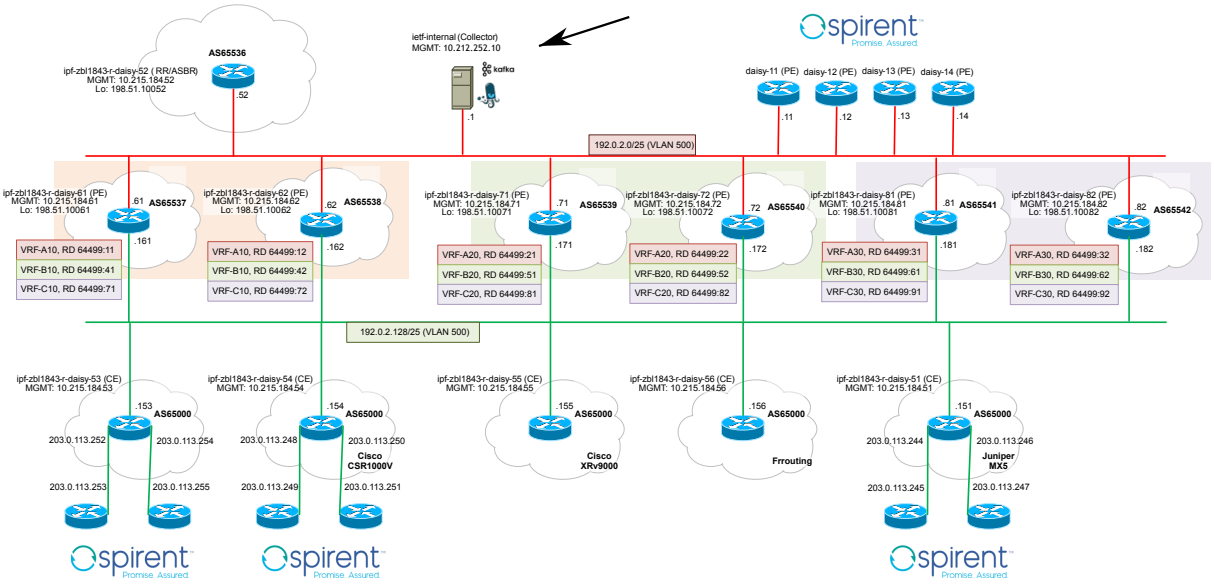

Figure 3.1: Topology of the Swisscom IETF interoperability lab. The collector daemons' host is located towards the top-center of this diagram.

3.2 Dataflow

From an end-to-end perspective, network devices send network telemetry data using various protocols such as BMP or IPFIX to well-known, internal static IP addresses. The resulting TCP and UDP traffic is routed to network hosts running collector daemons using anycast. These collector daemons combine the network telemetry streams into messages, as shown in listing 3.1. These messages are then published to a message broker, which (after an optional post-processing step) relays them to a storage system. This storage system can be queried live, e.g., by a network operator, or offline, e.g., by batch reporting or automation. Figure 3.2 summarizes the overall setup.

Combining the network telemetry streams at the individual collector level is a fundamental design decision of this system. This decision

- enables resource efficiency gains by distributing most of the computation across an arbitrary number of nodes close to the network telemetry sources,
- resolves per-source data completeness questions by enforcing that ei-

```
1  {
2    "event_type": "purge",
3    "label": "sgs01ro1010olt",
4    "comms": "60633:100_60633:265_60633:1001_60633:1032_64497
         ↪ :1528_64499:6000",
5    "ecomms": "RT:12429:20000001_RT:60633:1100001715",
6    "peer_ip_src": "138.187.57.53",
7    "src_comms": "60633:100_60633:204_60633:1004_60633:1020_60633
         ↪ :1034_60633:10004_60633:10031_60633:10044",
8    "src_ecomms": "RT:12429:30000001_RT:12429:32100001_RT
         ↪ :65511:1581_RT:65511:881581",
9    "iface_in": 33,
10   "iface_out": 47,
11   "mpls_vpn_rd": "2:4200005685:11",
12   "ip_src": "85.3.167.134",
13   "net_src": "85.3.164.0",
14   "ip_dst": "195.186.219.32",
15   ...
16 }
```

Listing 3.1: A (truncated,) combined telemetry message. Values in lines 4, 5, 7, 8, 11, and 13 originate from control plane telemetry. Values in lines 9, 10, 12, and 14 originate from forwarding plane telemetry. Values in lines 2, 3, and 6 represent metadata added by the collector daemon upon collection.

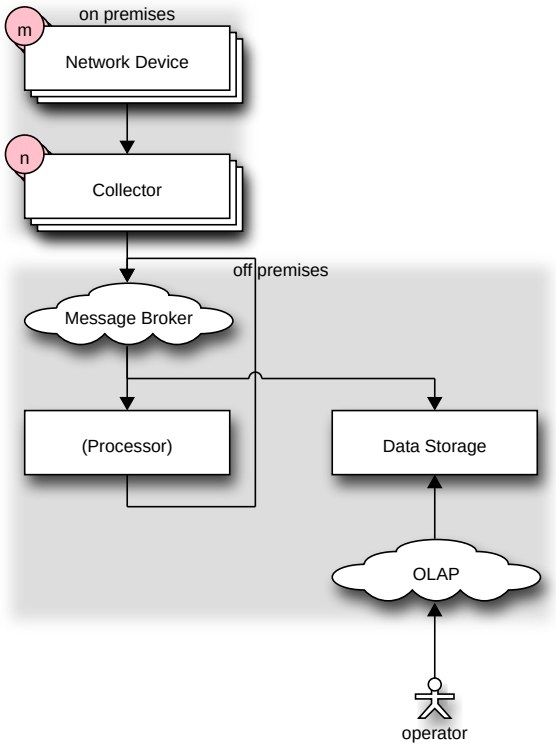

Figure 3.2: End-to-end data flow from the network device to the user's screen.

ther all messages received by the storage system are complete in relation to all perspectives, or entirely absent, and

- enables data aggregation of different collection protocols and perspectives at data-collection time, thus reducing the amount of data having to be ingested by the message broker.

A possible alternative approach would have been to merge the data late within the storage system (possibly even in a batch processing setting). In such a setting, the collector daemon would merely enrich the incoming information with collection-related metadata, convert it into a compatible message format and then forward it. One benefit of this approach would be a greater degree of flexibility in the analysis, as the pristine, original messages would be available for processing without the need to distribute new aggregation rules to all collector daemons. The drawback is that none of the benefits of the the original decision are realized.

3.3 Components

On the host, the incoming network telemetry is forwarded to the collector daemons (instances of nfacctd, a part of pmacct) by a loadbalancer (HAProxy). Figure 3.3 zooms in on this sub-aspect.

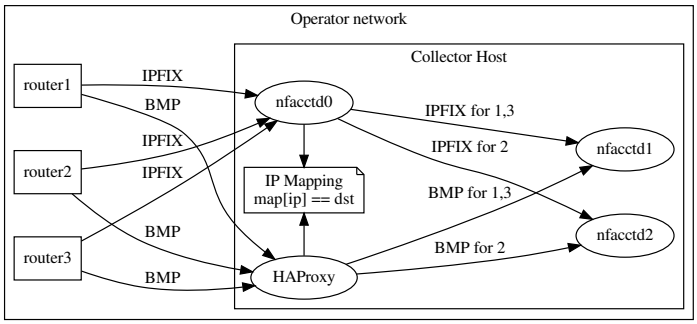

Figure 3.3: Network telemetry flow. Network devices send traffic to a single endpoint per protocol, pointing at either an HAProxy instance or a nfacctd instance acting as a proxy. Static mapping from the network telemetry source's IP address to a nfacctd backend is responsible for load balancing.

3.3.1 *HAProxy*

HAProxy is a free and open source software loadbalancer and proxy for HTTP and TCP [51]. It terminates the TCP connection from the network telemetry sender and forwards the payload to the network telemetry collector daemons. The core benefit it provides is that the forwarding logic is configurable. This allows us to map specific network telemetry senders to specific network telemetry collector daemons without changing neither sender nor collector daemon configuration. While the amount of configuration work is comparable for the initial setup, any incremental changes are vastly simplified.

In the context of network telemetry collection within our environment, the HAProxy was configured using static configuration. It was set up to route to a static set of backends (i.e., IP address:port combinations), each representing a collector endpoint on the local host. A static lookup table (technically: an HAProxy ACL) is used to inspect the network telemetry source's address and to determine the backend to route to.

The maintenance effort of keeping these two mappings current was considered a major pain point.

3.3.2 *Pmacct*

The pmacct project consists of a family of related software tools and daemons for the purpose of collecting, aggregating, and exporting network telemetry [35]. One of the components of the software suite is the *nfacctd* daemon which collects IPFIX telemetry (the "nf" prefix refers to NetFlow, a predecessor to IPFIX).

There are multiple instances of nfacctd running on *ietf-internal*, each acting in one of two capacities:

1. as actual collector daemons, processing, aggregating, and forwarding the network telemetry data, specifically BMP (via their BMP plugins) and IPFIX, or

2. as UDP load balancers, simply directing incoming UDP traffic to one of the instances running as actual collector daemons (see previous point).

In the latter mode it makes use of two additional lookup tables

1. to tag incoming packets based on their source addresses, and

2. to forward tagged packets to collector daemons with the corresponding tag.

The first of these tables was considered another major pain point.

As configured in the environment, the daemon combines/aggregates both telemetry types and feeds the resulting messages to a message broker.

3.3.3 Remaining Components

The later parts of the pipeline have little influence on this thesis but help illustrate the entirety of the setup: *Apache Kafka* [2] serves as the message broker while *Apache Druid* [1] is responsible for storage. *Imply Pivot* [22] provides a web-based user interface for ad-hoc reporting and analysis of the collected data.

Chapter 4

Design

Bearing in mind the available options and the context of the destination environment, we designed and implemented an extension to the pmacct-based collection system from chapter 3.

4.1 Requirements

As the deployment target of the solution is an existing environment carrying customer data, the system is subject to a diverse set of requirements. Some are core functional requirements imposed by the problem statement, while others account for challenges network engineers, network operators, and system administrators face in the pursuit of meeting service level agreements safely and reliably.

Device scoped, cross-protocol balancing. First and foremost, all network telemetry streams of the same origin must be balanced to the same destination, i.e., collector daemon. Due to the diversity of the network telemetry protocol stack, this is required across all transport protocols. In practice, a single network device must establish its BMP session with the same collector daemon process to which it is sending its IPFIX datagrams. Otherwise, their data cannot be correlated and/or aggregated.

Stable, persistent balancing. Individual collector daemon failures must not cause dissociation of the incoming network telemetry streams. Failover behavior must be identical between all transport protocols, despite the sticky nature of TCP connections.

Fault isolation. Individual collector daemon failures must not exhibit negative side effects on other collector daemons. If running near capacity in configurations without load shedding, a single failure would shift the load across non-failing instances, thereby bringing them into a state of overload, resulting in cascading failures affecting the whole system.

Use of existing tools (pmacct). The pre-existing collection pipeline relies on software from the pmacct project to perform the actual ingestion, aggregation, and export. Furthermore, any changes to the system must remain compatible and not lose existing functionality.

Restart stability. With multiple collector daemons per host active at any time, coordination and interaction become relevant. Consider the case of a software update to the collector daemon binary requiring a restart thereof. A synchronized restart of all running collector daemon processes is not only challenging to coordinate with a conceptually unbound number of collector daemons, but also introduces safety and reliability issues. Should the updated version prove broken, the rollback process is bound to increase downtime. This is also the case in the context of cascading failures triggered by load spikes common during the initial startup phase of processes. It is thus necessary to avoid unnecessary coupling of collector daemon process lifetimes, allowing the administrator to launch or terminate them one at a time.

Unified handling of TCP & UDP. One of the core value propositions of the collector daemon is to aggregate disparate network telemetry protocols. As these protocols use different transport protocols (TCP for BMP, UDP for IPFIX), there is a corresponding need to handle these uniformly.

Stable programming interface. As there is no guarantee that all collector daemon hosts will run the same versions of the dependency stack, using a stable interface is the first prerequisite for stable and maintainable operation. Requiring source code changes on *each* (re-)deployment is not deemed acceptable.

Stable binary interface. By extension of the previous point, even a recompilation without any actual changes introduces friction and constitutes a moving part liable to breakage (at 3 am on a Sunday morning in particular).

Low configuration overhead. Provisioning configuration derived from the ever-changing device inventory or network topology of a large network operator is a difficult task. This creates high deployment and maintenance burdens on operators, especially in setups without centralized network controllers maintaining globally consistent models but rather traditional distributed protocols (hopefully) reaching eventual consistency. Minimizing the amount of configuration necessary to start and operate the system and, in particular, allowing for agnosticism regarding the identity and location of network telemetry originators avoids this set of problems entirely.

Performance parity. Any solution addressing the challenges above should not consume more system resources than the existing HAProxy based system.

4.2 Overview

Given the constraints listed in section §4.1, the SO_REUSEPORT option, as described in section §2.2, offers a compelling starting point. With all collector daemons binding to the same IP:port pairs (per network telemetry protocol), this establishes load balancing without the use of intermediaries such as a reverse proxy. Unfortunately, simply enabling this socket option does not allow the customization of the mapping determining how connections/UDP datagrams are actually distributed across the participating sockets. This causes two distinct issues:

1. It is impossible to ensure that both TCP traffic and UDP traffic originating from the same network telemetry source are assigned to sockets associated with a single collector daemon process since the protocol field serves as an input to the hashing function.

2. It is impossible to ensure that single protocol traffic originating from the same network telemetry source is assigned to the socket associated with the correct collector daemon process, as the network telemetry protocols allow for the use of ephemeral source ports. A trivial example of this situation is the restart of the BMP daemon on a network router resulting in a different TCP source port.

The most straightforward approach to address these issues would be to customize the hashing algorithm used to locally assign TCP connections/UDP

datagrams to sockets. By using only the network telemetry source's IP address as an input to the hashing function, cross-protocol traffic would be consistently hashed to the same sockets. Yet, more problems surface:

1. It is impossible to avoid race conditions influencing the final hash values. TCP connections are distributed among all available sockets during connection setup. Assuming that there is little control over the rate and timing of incoming connections unless the startup is perfectly synchronized, the assignment will not be balanced. The situation is even worse upon disruption of the steady operating state: If a single collector daemon process out of N dies, all connections associated with the socket are reset. If the network telemetry sources attempt to set up new connections before the offending collector daemon process is restarted, it may happen that these are established with one of the remaining $N - 1$ processes. This both increases the load on the existing processes and starves the restarted one of work.

2. The assignment target itself remains unstable. The flow hash indexes into a sequence of sockets albeit the ordering on that sequence is not guaranteed even if the index itself was stable. Even if the sequence was guaranteed to be in insertion order, this would result in a cross-protocol race condition, unless there was an avenue to impose a partial order on the TCP-UDP socket pair creation *at all times*.

Furthermore, there is no standardized facility to customize the inputs to the used hashing function. The only way to accomplish this is to modify the logic in place and release a new, deployment-internal kernel version. As per subsequent section 4.2.1, this is rather undesirable.

However, with our newly gained background in BPF from section §2.3, we now have an additional API available to us to resolve these shortcomings. In particular, in Linux 4.19 Martin KaFai Lau implemented the BPF program type SK_REUSEPORT [26]. A BPF program of this type attaches to a SO_REUSEPORT-enabled socket. It is run whenever a new TCP connection targeting the bound address is being set up or when a UDP packet for said address is received. SK_REUSEPORT programs are able to inspect the contents of the socket buffer that triggered its execution in the first place and must decide whether a connection is established or reset instead (in case of UDP: whether the datagram is processed or dropped). Furthermore, it is also able to assign the incoming connection/datagram to a socket. With this building block, we are now able to resolve the roadblocks enumerated previously.

This is not the only option, however. One alternative, even more generic approach is the creation of a BPF program of type SK_LOOKUP. First introduced into Linux 5.9, it allows for yet more programmability of the lookup procedure. In fact, the illustrating example was to map an IP range onto a single socket [47]. SK_LOOKUP programs do have significant drawbacks, however:

1. The attachment point for SK_LOOKUP is a network namespace rather than a socket. This incurs additional administrative overhead in the case of creating custom namespaces for the collector daemon to run in, as well as the difficulties of lifecycle management thereof. While attaching to the initial/root namespace is possible, doing so breaks any veneer of isolation and reduces the composability of deployment. Any misconfiguration or bugs may now affect overall host health rather than being confined to the collector daemon, especially since a BPF program attached to the initial namespace will outlive the processes originally attaching it.

2. Having been introduced in kernel release 5.9 (October 2020) only comparatively recently, this BPF program type was not available on the kernels deployed on the collector machines which were running release 4.18 (first released in August 2018). We are not aware of any backports to 4.18 either.

In contrast, XDP (eXpress Data Path) represents an option engaging at the earliest point in the packet processing pipeline (c.f. section §2.1). Before any further processing, or even the allocation of a socket buffer ("skb"), the XDP BPF program would inspect the raw data from the device driver[1] and determine whether to pass it to an AF_XDP socket in userspace or to disregard it by passing it back into the pipeline [20]. Aside from the natural complexity of implementing a (partial) userspace networking stack, XDP is not subject to any netfilter processing or queuing disciplines, creating yet more administrative difficulty.

With this landscape of approaches in mind, we designed and implemented a solution built upon the first viable choice: a SK_REUSEPORT BPF program.

[1] Certain devices/drivers feature offload support XDP [41]. In this case, the program is run even earlier.

4.2.1 Kernel vs BPF

Much of the same functionality can also be achieved by creating Linux kernel modules or implementing the changes in-place in a locally maintained fork of the Linux kernel source tree. While this method allows for the maximal freedom in choice of design and/or a custom tailoring most precisely matching the requirements, there are significant downsides to this approach.

A kernel fork, however minimal, introduces a high burden in both upfront cost and ongoing maintenance cost. Unless there already is an established process of handling custom patches in place, at the very least this implies setting up dedicated software build pipelines that patch the changes into incoming distribution kernel updates. Associated manual conflict resolution would further aggravate the situation, as there are no promises of code compatibility between releases or even revisions[2]. Major changes requiring a rebase might end up incurring complexity equivalent to a rewrite from scratch. Modified kernels further require an additional stage release qualification pipeline. In the case of a security update, this introduces a previously unnecessary tradeoff between release safety and security.

For production machines in particular, additional kernel updates consume precious error budget and make it harder to maintain service level objectives.

Worse still, with little chance of the upstream Linux developer community accepting environment-tailored changes into the main branch, this malaise is potentially perpetual.

In environments with requirements on code provenance/signing—possibly certified by another party—custom kernel code is likely entirely untenable.

Thus, writing and maintaining actual kernel code is *prohibitively* costly given the availability of *any* alternatives.

4.3 Detailed Design

With the design space explored, we can take a closer look at the implemented approach. It consists of three parts: the kernel side implemented in BPF, the userspace side as an extension to nfacctd, and the BPF maps serving as an interface between them. In the following, each will be covered in its own subsection.

[2] In kernel version 5.12.1, “5” is the major release, “12” is the minor release, and “1” is the revision.

```
struct sk_reuseport_md {
        /*
         * Start of directly accessible data. It begins from
         * the tcp/udp header.
         */
        __bpf_md_ptr(void *, data);
        __bpf_md_ptr(void *, data_end);
        /*
         * Total length of packet (starting from the tcp/udp
            ↪ header).
         */
        __u32 len;
        __u32 eth_protocol;
        __u32 ip_protocol;      /* IP protocol. e.g.
            ↪ IPPROTO_TCP, IPPROTO_UDP */
        __u32 bind_inany;       /* Is sock bound to an INANY
            ↪ address? */
        __u32 hash;             /* A hash of the packet 4
            ↪ tuples */
};
```

Listing 4.1: Context of a SK_REUSEPORT BPF program. Some comments have been omitted for brevity.

For the rest of this section, we will primarily describe TCP without (significant) loss of generality. Except when called out explicitly, replacing "connection" with "datagram" and "once, on establishment" with "per packet" effectively describes the UDP equivalent.

4.3.1 Kernel

Within the BPF program itself, three core pieces of functionality are needed: source IP address based hashing, socket lookup, and feedback to the kernel networking stack on whether to establish or drop the connection.

For SK_REUSEPORT programs, the available context[3] (listing 4.1) does not expose network and transport protocol metadata directly. While we can access (see section 2.3.3) parts of the socket buffer through the `data` member, the accessible range starts only at the transport layer header. Therefore we cannot directly inspect properties such as the source IP address of the incoming connection request. However, as per this program's protocol,[4] it

[3] include/uapi/linux/bpf.h:4521
[4] net/core/filter.c:10161

can retrieve this data via the `skb_load_bytes_relative()` BPF helper. With its help, we read an IPv4 header's length of bytes from `BPF_HDR_START_NET` (i.e., the start of the network header) thereby granting us access to the source IP address after all.

The hashing function itself is virtually unchanged from the one in the traditional path. The function used by IPv4/AF_INET, `inet_ehashfn()`[5], is a thin wrapper around the Jenkins hash function[6]. Since we cannot call arbitrary functions from BPF programs, and as this particular function is not exposed through a BPF helper, the implementation itself was lifted from the Linux source tree. One of the inputs to the hash function, the seed, provides an additional benefit when exported into userspace: It enables the creation of tools that can simulate the traffic allocation. By extracting the hash function setup into a standalone helper that has access to the seed, an operator can easily inspect the current address-to-process mapping. As a corollary, the operator can manipulate the value currently in use to conduct debugging or simulate failure conditions.

The return value of a SK_REUSEPORT program is a verdict on whether or not to accept the connection. This provides the API necessary to circumvent all race conditions of SO_REUSEPORT, as there now is an option to reject specific connection attempts until the software stack is in a ready state. Thus, we can introduce the notion of a *fixed* number of hash buckets, i.e., number independent of the number of sockets actually bound to our address. While implementing failover support might appear opportune here, the BMP protocol would make this difficult at higher levels of the stack, see appendix C.

Each participating collector daemon process's socket is registered in a custom lookup table. A special BPF helper, `bpf_sk_select_reuseport()`, can access this table using an array index and (via side-effect) establish the association between the currently handled connection and the target socket. If the hash indexes into a row that is empty, i.e. not backed by a corresponding socket, the collector daemon associated with that row is considered temporarily unavailable. These connections are rejected.

Due to a lack of use cases within our environment, support for IPv6 was not implemented. Attaching the BPF program to AF_INET6 sockets will hash octets 4–7 of IPv6 source address, breaking the balancing for the general case.

Some parameters of the program, such as the bucket count and the hash

[5] net/ipv4/inet_hashtables.c:31
[6] include/linux/jhash.h

seed, always require values. For simplicity, these are initialized on the first run. The program is available for inspection in [4][7] or, in inlined form, in appendix B.

4.3.2 Interface with Userspace

BPF maps are the core mechanism to share data between the kernel and userspace. In total, we make use of four maps:

nonce is a 1×1 array that contains a value initialized at startup. Despite its name, it is is closer to a seed and serves as an input to the Jenkins hash function as implemented by the kernel[8]. It must be shared among all processes balancing the same load.

size is a 1×1 array that contains that contains the intended number of processes balancing the same load. It must be shared among all processes balancing the same load.

tcp_balancing_targets & udp_balancing_targets are maps of type REUSEPORT_SOCKARRAY. They are populated with the references to all the listening sockets of all processes balancing the same load. They must be shared among all these processes. Our implementation uses one for TCP and one for UDP. The use of two separate maps is not strictly necessary. Instead, the respective sockets could be packed into a single map and accessed via somewhat more involved index calculation. One implementation could do the hash calculation using $\text{key}_{\text{TCP}} := h(\text{ip}_{\text{src}}) \mod \texttt{size}$, and $\text{key}_{\text{UDP}} := \text{key}_{\text{TCP}} + \texttt{size}$ while ensuring that the map has at least $2\,\texttt{size}$ capacity.

Listing 4.2 demonstrates how maps are defined, using *tcp_balancing_targets* as an example.

Since all maps must be shared, a sharing mechanism is needed. One option is to have only the chronologically first collector daemon process load the maps and distribute the file descriptors to the other processes using Unix Domain Sockets. This would require implementing something akin to a client-server service and/or a peer-to-peer consensus algorithm. There is a much simpler alternative, however: BPF Object Pinning. This allows us to use a filesystem path for map sharing.

[7] reuse/reuseport_kern.c
[8] include/linux/jhash.h:63

```
struct {
  __uint(type, BPF_MAP_TYPE_REUSEPORT_SOCKARRAY);
  __type(key, u32);
  __type(value, u64);
  __uint(max_entries, MAX_BALANCER_COUNT);
  __uint(pinning, LIBBPF_PIN_BY_NAME);
} tcp_balancing_targets SEC(".maps");
```

Listing 4.2: Definition of the *tcp_balancing_targets* map. For map type of REUSEPORT_SOCKARRAY, key and value size are fixed. As it is an array-type map, *key* is simply an array index. *value* is a kernel address of the socket structure. When inspected from userspace, these values are replaced by socket cookies to prevent leakage of kernel addresses. *max_entries* is the upper bound on the row number (*MAX_BALANCER_COUNT* is a compile time constant supplied by the operator). Finally, *pinning* instructs libbpf to pin the map into BPFFS.

Using pinning as the mechanism provides a major beneficial side effect. With references to maps exposed in BPFFS, they can now easily be inspected, manipulated, and deleted using *bpftool* [30]. This empowers an operator to create appropriate monitoring or conduct debugging if necessary. It also relieves the userspace component of the system from handling the rarest and most complicated edge cases. The drawback is that these references are now shared system resources that can experience various failures. An actual, real-life example would be the occurrence of naming collisions caused by multiple collector daemons reusing the same pin paths.

Unlike maps, the BPF program itself is not pinned in BPFFS. Each collector daemon loads a new program on startup (presumably identical to the last). This also serves as an implicit update mechanism that does not require the creation and inspection of version metadata.

The map definitions are available for inspection in [4][9].

4.3.3 Userspace

To make use of the presented load balancing mechanism, all of pmacct's daemons in scope need to implement the following functionality:

1. The socket that will receive traffic from the targeted network telemetry

[9] reuse/reuseport_kern.c

protocol needs to be SO_REUSEPORT enabled. This is done using setsockopt() [33].

2. The ELF binary file representing the BPF program needs to be opened and loaded into memory. Libbpf provides the BPF object abstraction for this.

3. The BPF Object needs to be configured. This includes setting the pin path and its prefix in particular.

4. References to the BPF maps need to be acquired. Depending on whether a map's pin path already contains a reference to an existing map, either that map needs to be opened or a new map has to be created.

5. The BPF program needs to be relocated to work with the running kernel and map references within the instruction streams need to be resolved.

6. The BPF program needs to be loaded into the kernel.

7. The loaded BPF program needs to be attached to the socket. This is done using setsockopt(). If the socket is in state LISTEN already, it will no longer receive additional connections.

8. This socket now needs to be inserted into the respective REUSEPORT_SOCKARRAY map.

If these steps are executed successfully, the collector daemon will then receive traffic for the configured fraction of all possible source IP addresses.

This is only a high-level overview of the steps involved. A more detailed description of what happens at the system level is available in section §A.1.

To enable and make use of this functionality, pmacct receives three new configuration options:

reuseport_hashbucket_count is the option that declares the how many collector daemons the operator intendeds to use to balance the load. In the case where the corresponding BPF maps already exist, it also serves as a sanity check preventing mismatching configuration between multiple collector daemons.

reuseport_hashbucket_index specifies which bucket the process setting this value represents. Values are in $[0, \text{reuseport\_hashbucket\_count} - 1)$.

reuseport_bpf_prog specifies the filesystem path at which the ELF binary containing the BPF program can be found.

The functionality described in this section was implemented for pmacct's nfacctd and its BMP plugin. As a result of the implementation, pmacct gained a dependency on libbpf $\geq$ 0.4.0.

The changes to pmacct are available for inspection in [3].

4.4 Behavioral Aspects

Since each collector daemon is configured independently, a risk of incompatible configuration arises. The three configuration options can be partitioned into two groups: *quasi-static* and *dynamic*. For the former, a meaningful value needs to be set once and rarely—if ever—changed. This is the case for the path to the BPF ELF binary as well as for the daemon's own bucket number; while any number of daemons may be running at any time, each has its own configuration file specifying its bucket number. This is not the case for the total number of buckets, however. Scaling this number up (or down) may be needed periodically due to organic changes in network device deployments. Since changes to this setting affect all concurrently running collector daemons, additional sanity checks were implemented to verify consistency. As a result, to change the total number of buckets intentionally, all collector daemon processes need to be terminated and the map be deleted. Otherwise, the number has to be changed imperatively from outside pmacct (e.g., using *bpftool*).

There can only be one SK_REUSEPORT BPF program per reuseport group (i.e., the set of sockets binding to the same IP:port combination), with the most recently attached one winning out [34]. Attaching the program on reuseport group with unsuspecting members will thus result in a "hostile takeover," starving them of new TCP connections/UDP packets. While unlikely to happen by accident, it may be considered surprising behavior, especially during migrations from "just" SO_REUSEPORT to the BPF program presented here.

4.5 End-to-End

So far, most aspects of the design have been described in isolation. In this section, we will illustrate their interaction in chronological order using the

following scenario: In an initially empty world, a single collector daemon starts up and accepts an incoming TCP connection.

While “maps”, “ebpf”, and “kernel” are drawn as three separate systems in the following figures, this was done for illustrative purposes only. They are all part of the Linux kernel.

At some point during its startup phase, nfacctd starts initializing the socket. After enabling the SO_REUSEPORT option, the daemon can now bind the socket to the desired address (figure 4.1).

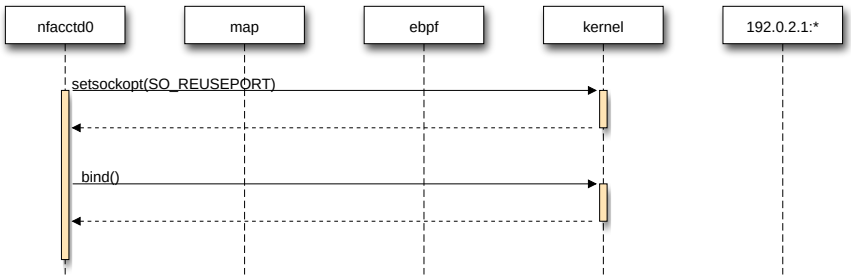

Figure 4.1: End-to-end: Prerequisites. A collector daemon (“nfacctd0”) configures its socket. The raw IP address represents a (currently idle) client.

nfacctd now starts initializing the BPF related components (figure 4.2). This is accomplished primarily by the use of the bpf() system call API [32]. It creates the maps it uses to keep state and to communicate with userspace, and resolves references to them within the instruction of the BPF program. After successfully loading the program into the kernel, it attaches it to the socket. The core functionality is now ready to be used, but, in its current state, would reject all ingress as destinations have yet to be configured. To alleviate that nfacctd now inserts its own socket as a target into the *tcp_balancing_targets* map and puts the socket into the listening state. The setup is now complete. An in-depth description of these steps can be found in section §A.1.

Let us consider the situation of a network telemetry source attempting to connect to the configured endpoint on the host running nfacctd (figure 4.3). After receiving the TCP packet, the kernel eventually determines that this packet is destined for local delivery and the control flow reaches the socket lookup procedures. It finds a SK_REUSEPORT program attached and calls the program. The program then inspects the headers and computes a hash based on the IP source address. It looks up the number of nfacctd daemons

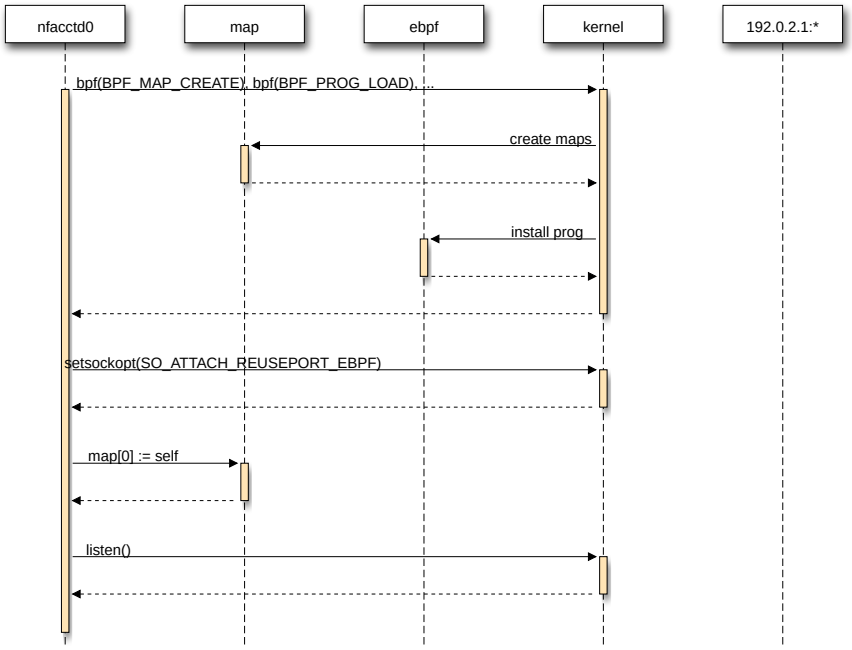

Figure 4.2: End-to-end: Setup. All BPF initialization is contained in this phase. After the self-registration in the socket lookup, there are no more BPF-specific actions to be taken from the userspace program.

among which to balance, and determines the target hash bucket implicitly setting the target socket during the lookup. In this case, the socket in the target bucket belongs to our originally nfacctd instance ("nfacctd0"). Since the lookup succeeded, the BPF program instructs the kernel to continue with the handshake.

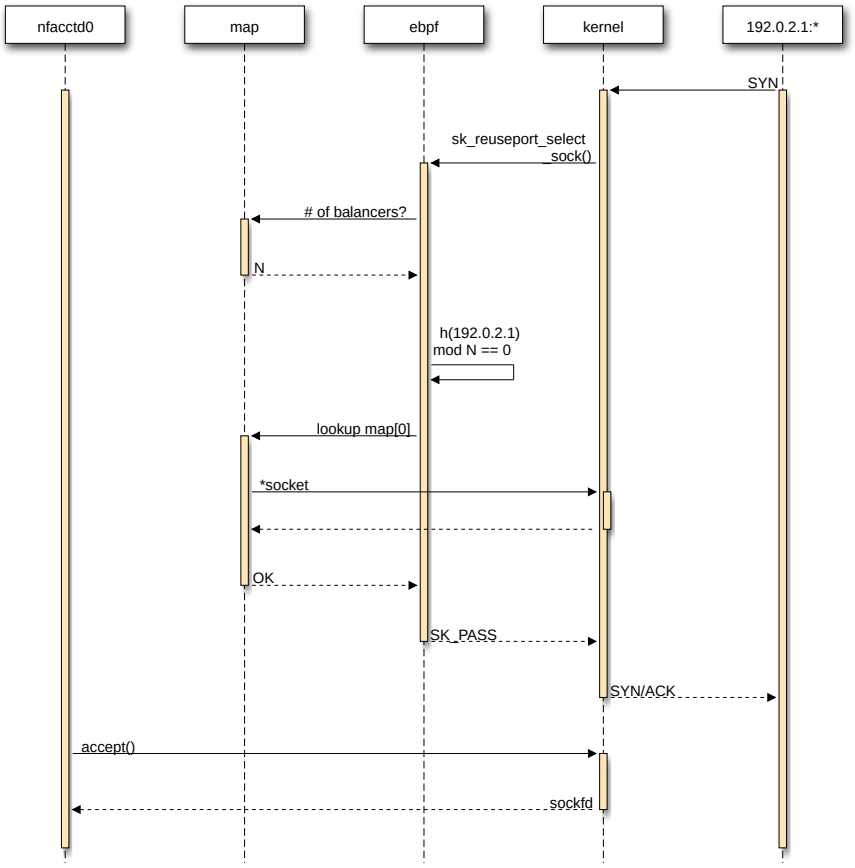

Figure 4.3: End-to-end: Receiving a connection. During socket allocation, the BPF program is executed to determine the target socket. In this case, this was successful, and the connection setup was permitted to proceed (final ACK not shown).

With the connection established, actual communication no longer involves anything BPF related (figure 4.4).

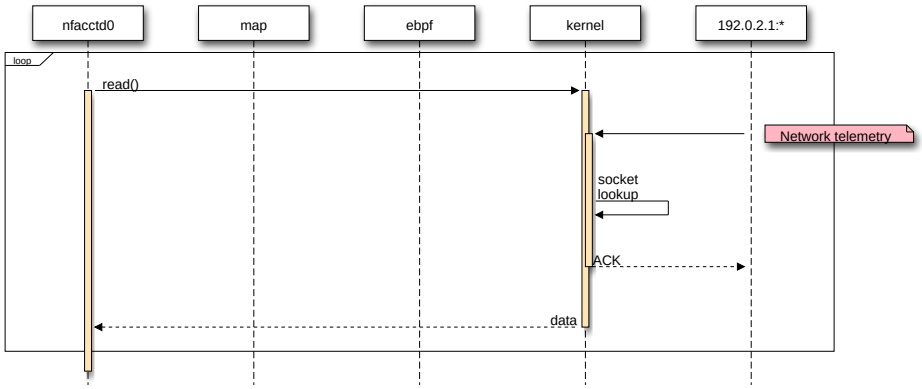

Figure 4.4: End-to-end: Regular data flow.

Chapter 5

Evaluation

With both high-level design and low-level implementation details fresh in our minds, we can now evaluate the system in light of the requirements established earlier in section §4.1.

Device scoped, cross-protocol balancing. By balancing based only on the IP source address, while disregarding transport layer type and metadata, we were able to ensure that any network telemetry from that address is routed to the same collector daemon. This requirement was met.

It should be noted that there is a major simplifying assumption at play: all applicable network telemetry is sent using the same interface. While this is guaranteed to be true in the targeted deployment scenario, this is not a universal truth. If this assumption were to be invalidated, there would be at least two possible approaches to addressing the resulting gap. First, one could introduce one additional layer in indirection: a map from source IP address to device ID. In this setup, the BPF program would do a map lookup to retrieve the device ID and then hash that ID to determine the target socket. Secondly, the BPF could inspect the *unencrypted* packet payload itself to find common identifiers. The former option requires the creation and maintenance of the mapping, while the latter requires additional packet dissecting logic per supported network telemetry protocol.

Stable, persistent balancing. The balancing mechanism itself is not dependent on whether a balancing target exists (i.e., a collector daemon has registered its socket). By simply rejecting those connections and implicitly

communicating to the network telemetry source to retry, we avoid moving sticky connections to other collector daemons. This requirement was met.

Fault isolation. The logic of the previous requirements can be extended to this requirement: by rejecting connections/packets in failure cases rather than shifting them to other collector daemons we prevent overload and cascading failures. This also works in the case where the root cause of failure is on the other side: should a network telemetry source send a packet of death, that source will *not* be passed around until all collector daemons are in a crash-loop. This requirement was met.

Use of existing tools (pmacct). The userspace logic is entirely contained within pmacct. This requirement was met.

Restart stability. As previously elaborated, both the hashing and the set of slots for balancing targets are stable and independent. Thus, any participating collector daemon can start/stop/crash at any time in any order. This requirement was met.

That said, the isolation is not perfect. The BPF program itself is a shared resource. If a misconfigured collector daemon were to replace the BPF program with one that always rejects all connections and packets, this would constitute a denial-of-service against that host's collector daemons. This situation can be prevented by isolating the act of program *loading* into a separate, tightly controlled program. Collector daemons would be able to attach/detach existing programs at will, but not attach entirely unacceptable ones. However, with all collector daemons having to run as the same user ID as per SO_REUSEPORT's security model, some trust is fundamentally required.

Unified handling of TCP & UDP. There is only a single BPF program that can be attached to both SOCK_STREAM and SOCK_DGRAM sockets. The userspace logic itself is entirely protocol agnostic as well. This requirement was met.

Stable programming interface. eBPF is a well-established subsystem of the Linux kernel and has been around for over seven years, and Linux's mantra of "don't break userspace" needs no introduction. The additional dependency on pmacct, libbpf, is self-contained and can be bundled when

targeting sufficiently old systems (in fact, that is how pmacct was compiled on the IETF Lab machine). This requirement was met.

Stable binary interface. Quoting the relevant official documentation's answer in its entirety,

> Q: Does BPF have a stable ABI?
>
> A: YES. BPF instructions, arguments to BPF programs, set of helper functions and their arguments, recognized return codes are all part of ABI. However there is one specific exception to tracing programs which are using helpers like bpf_probe_read() to walk kernel internal data structures and compile with kernel internal headers. Both of these kernel internals are subject to change and can break with newer kernels such that the program needs to be adapted accordingly. [27]

This requirement was met.

Low configuration overhead. We added three configuration options to pmacct, two of which are only relevant during the first setup. In return, this rendered obsolete

- the entirety of HAProxy server configuration,
- any lookup tables providing routing information to the above,
- a dedicated configuration for a nfacctd instance serving as the UDP loadbalancer,
- any tagging rules classifying the input for the above,
- lookup tables mapping tags to collector daemon endpoints, and
- any processes required to update any of the above.

Best of all, enabling network telemetry on new network devices no longer requires any configuration changes on the collector daemons' hosts. This requirement was met.

5.1 Performance

It is now time to look at the last item on our list of requirements, performance.

The existing, HAProxy-based deployment is not documented to have reached a saturation point or overload point attributable to the proxy with respect to a maximum number of connections or packets per second. Rather, the collector daemons themselves have already been established to be the limiting factor according to company internal benchmarks. In this context, the improvements to the parts of the pipeline that are traversed *before* the collector daemon reads from a socket are of limited use to the overall system.

As for nfacctd itself—as the cost of running the additional userspace code is only paid once during startup—a negative change in the saturation or overload characteristics is unlikely.

To measure resource usage, we ran both the existing HAProxy-based system as well as the BPF based one for one hour each while a BMP load generator [53] was simulating 512 clients. Each system was configured with two collector daemons performing actual data aggregation and forwarding. This likely represents an edge case of maximal usage savings, assuming that the number of deployed HAProxy and nfacctd-as-a-loadbalancer instance scales sublinearly with the number of nfacctd instances acting as collector daemons.

The test itself ran in a virtual machine configured with 8 Intel Xeon E5-2620 v4 cores at 2.10GHz and 64GB of RAM. The operating system was CentOS Linux 8 with Linux kernel version 4.18.0-240.22.1.el8_3.x86_64 (unmodified distribution version). nfacctd was based on commit 056ef7e in [3].

All processes of each system were started in a cgroup with the cpu accounting and memory controllers [28, 29] enabled. The raw numbers were collected by a script that would read out the total usage numbers from the cpuacct.stat and memory.stat pseudo-files of that cgroup every (wall clock) second.

The resulting measurements are depicted in figure 5.1 for CPU usage and in figure 5.2 for memory usage. We can see that, while the memory usage pattern does not show visible differences, there is a significant reduction of CPU usage throughout the operating period. In total, we save about one-fifth of CPU usage (table 5.1). The reduction in time spent in kernel mode should prove particularly beneficial on shared hardware.

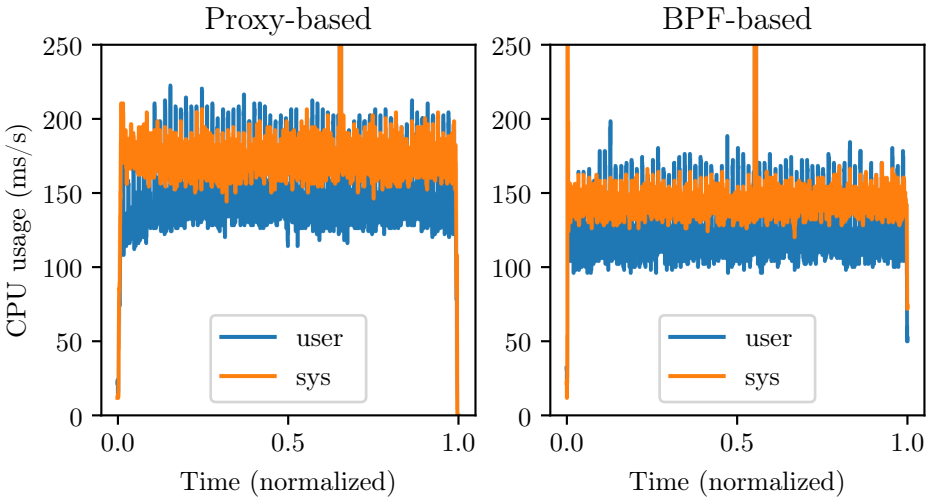

Figure 5.1: Comparison of CPU utilization between the existing system and the one described in this thesis. Benchmark timestamps are normalized to 1. A value of 1000 corresponds to the full utilization of one CPU core for the duration of one second.

Measure	Proxy	BPF	Δ
user	56 957	48 387	−17%
system	66 388	55 319	−20%
total	123 345	103 706	−19%

Table 5.1: Raw, total, cumulative CPU usage of two representative benchmark runs. Units are USER_HZ (1/100 s on this system).

The slow ramp-up on the proxy-based setup is due to an artificial limitation of the rate of creation of BMP clients, spreading that process over the course of one minute. Without this slowdown, on every attempt,

- a large number of connections would fail to be established (> 50% failure rate; no retries) and
- an HAProxy worker would terminate due to a segmentation fault.

In a company-internal review of this observation, this behavior was classified as a known issue. In contrast, there was no corresponding issue in the BPF-based setup, causing the vertical, initial spike.

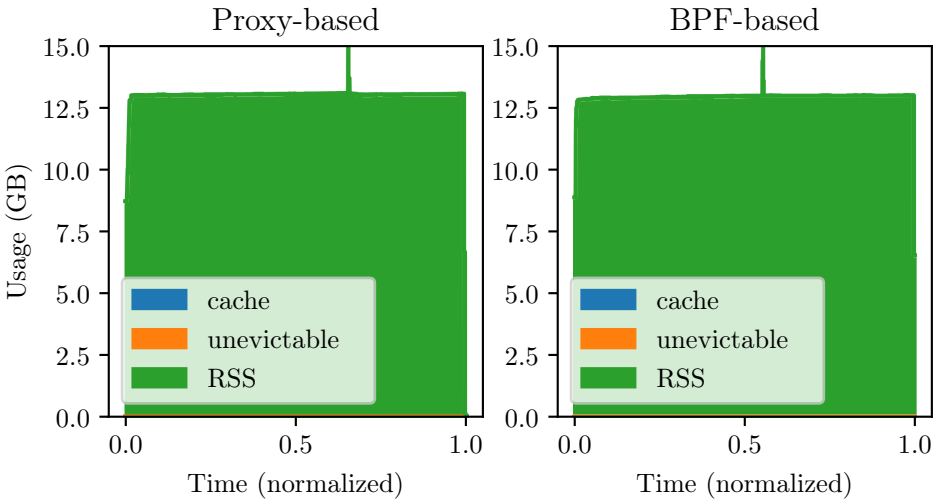

Figure 5.2: Comparison of RAM utilization between the existing system and the one described in this thesis. The median usage amounts to 14.0 GB for the proxy-based setup and 13.9 GB for the BPF-based one. The metrics are stacked, but the cache/unevictable metrics are trumped by the resident set size. Benchmark timestamps are normalized to 1.

In both setups, a large usage spike after the midpoint of the benchmark can be observed. It consumes an additional 2–3 GB of memory, saturates at least one CPU core, and appears during every single run, lasting for 15–20 seconds. Given our cgroup-based measuring setup, by process of elimination, it must originate in nfacctd, however, we were unable to determine the root cause. We do *not* believe it has any impact on our conclusions.

While the proxy-based setup runs two additional process trees (HAProxy & nfacctd-as-a-proxy), this is hardly visible. Spot checks indicated that they take up less than 150 MB RSS total, a non-zero, but otherwise negligible overhead.

We can therefore conclude that this last requirement was met as well.

Chapter 6

Outlook & Future Work

pmacct upstreaming. With the value of this work demonstrated in chapter 5 for Swisscom, it would be a logical next step to lower the barrier of usage for other interested parties. One of the hurdles is that these prospective users have to run a patch set on top of pmacct, which can be removed by integrating the change in upstream pmacct. We anticipate this to be a design challenge: As the first integration of a BPF program into the pmacct codebase, major decisions will have to be taken regarding API design. As shown in chapter 4, the interfacing of the BPF program and the userspace part (i.e., pmacct) is highly dependent on the BPF map definitions. Generalizing our implementation such that other BPF programs (and, more interestingly, other *types* of BPF programs) can be substituted with the right abstractions requires careful thinking.

A first step could be to only offer the program itself, as written, as a singular feature. With the program's behavior well understood and the integration points clear, the remaining challenge would be focused on productionization, e.g., monitoring, optionality (i.e., being able to enable/disable the functionality per network telemetry protocol), etc.

Third-party integrations. Within the scope of this thesis, two collector daemons received BPF program support. This coverage can be extended further. In fact, work is already underway for the out-of-tree *C-Collector for UDP-notif* library [44] which implements support for the (current draft of) the UDP-Notif protocol [56].

Extended performance analysis. It is possible to profile BPF programs using bpftool [30]. Doing so requires the availability of hardware performance counters which were not exposed in the virtual machine in the IETF lab environment. With deployment in production targeting bare metal, this should become feasible.

Cross-platform considerations. In its current form, the eBPF program is limited to Linux as its operating environment. While some eBPF porting work to FreeBSD was done in [18], there is no indication of general availability. However, in May 2021 Microsoft announced work on a Microsoft Windows implementation of BPF [38, 52]. Should support for SK_REUSEPORT type BPF programs be introduced, this would allow for cross-platform support from a single codebase.

Code simplification. Kernel version 5.13 introduces a facility to expose kernel functions to BPF without creating custom BPF helper functions [13]. If this were to be applied to the hashing function used in our BPF program, the program would be significantly simplified (c.f. appendix B).

Chapter 7

Summary

Network telemetry data collection is a basic requirement in modern networks. As we have shown throughout this work, it features some unique challenges due to the distributed nature of the problem and the specific requirements imposed on the data routing that need to be accounted for and handled explicitly.

In this thesis, we designed and implemented a system addressing these challenges and requirements. We extend our target environment's existing network telemetry collection infrastructure to support in-kernel loadbalancing, utilizing the Linux kernel's eBPF subsystem for assigning TCP connection and UDP datagram to collection endpoints according to our own business logic. In effect, we introduced device scoped, cross-protocol balancing of network telemetry streams across an arbitrary number of collection endpoints, enabling reliable network monitoring by ensuring robust correlation of network telemetry data at the collection endpoint. Our implementation requires less maintenance effort and much less configuration overhead than the previously existing architecture.

We evaluated this implementation and established that it not only covered and met all imposed requirements, but also performed more efficiently than the previously existing system.

Bibliography

[1] Apache Software Foundation. Apache druid. https://druid.apache.org/. [Online; accessed September 2021]. 19

[2] Apache Software Foundation. Apache kafka. https://kafka.apache.org/. [Online; accessed September 2021]. 19

[3] Bachmakov, E. pmacct (branch "reuse"). https://github.com/eduarrrd/pmacct/tree/reuse. [Online; accessed September 2021]. 31, 39

[4] Bachmakov, E. reuseport. https://github.com/eduarrrd/reuseport. [Online; accessed September 2021]. 28, 29

[5] Bauer, L., and Crequy, A. Exploring bpf elf loaders at the bpf hackfest. https://kinvolk.io/blog/2018/10/exploring-bpf-elf-loaders-at-the-bpf-hackfest/, October 2018. [Online; accessed September 2021]. 10

[6] Bernaschi, M., Casadei, F., and Tassotti, P. Sockmi: a solution for migrating tcp/ip connections. In *15th EUROMICRO International Conference on Parallel, Distributed and Network-Based Processing (PDP'07)* (2007), pp. 221–228. XX

[7] Cheng, Y., Chu, J., Radhakrishnan, S., and Jain, A. Tcp fast open. RFC 7413, RFC Editor, December 2014. http://www.rfc-editor.org/rfc/rfc7413.txt. XVIII

[8] Claise, B., Trammell, B., and Aitken, P. Specification of the ip flow information export (ipfix) protocol for the exchange of flow information. STD 77, RFC Editor, September 2013. http://www.rfc-editor.org/rfc/rfc7011.txt. 1

[9] Clemm, A., and Voit, E. Subscription to yang notifications for datastore updates. RFC 8641, RFC Editor, September 2019. http://www.rfc-editor.org/rfc/rfc8641.txt. 2

[10] Corbet, J. A jit for packet filters. https://lwn.net/Articles/437981/, April 2011. [Online; accessed September 2021]. 11

[11] Corbet, J. Tcp connection repair. https://lwn.net/Articles/495304/, May 2012. [Online; accessed September 2021]. XIX

[12] Corbet, J. Bpf: the universal in-kernel virtual machine. https://lwn.net/Articles/599755/, May 2014. [Online; accessed September 2021]. 8

[13] Corbet, J. Calling kernel functions from bpf. https://lwn.net/Articles/856005/, May 2021. [Online; accessed September 2021]. 44

[14] CRIU Team. Checkpoint/restore in userspace (criu). https://criu.org/Main_Page. [Online; accessed September 2021]. XX

[15] Engelhardt, J. File:netfilter-packet-flow.svg — wikimedia commons, the free media repository. https://commons.wikimedia.org/w/index.php?title=File:Netfilter-packet-flow.svg&oldid=564855543, 2019. [Online; accessed September 2021]. 5

[16] Fleming, M. A thorough introduction to ebpf. https://lwn.net/Articles/740157/, December 2017. [Online; accessed September 2021]. 9

[17] Graf, T., Lucente, P., Francois, P., and Gu, Y. Bmp (bgp monitoring protocol) seamless session. Internet-Draft draft-tppy-bmp-seamless-session-00, IETF Secretariat, February 2021. https://www.ietf.org/archive/id/draft-tppy-bmp-seamless-session-00.txt. XVIII

[18] Hayakawa, Y. ebpf implementation for freebsd. In *BSDCan 2018*. The BSD Conference, 2018. 44

[19] Hofstede, R., Čeleda, P., Trammell, B., Drago, I., Sadre, R., Sperotto, A., and Pras, A. Flow monitoring explained: From packet capture to data analysis with netflow and ipfix. *IEEE Communications Surveys & Tutorials 16*, 4 (2014), 2037–2064. 1

[20] Høiland-Jørgensen, T., Brouer, J. D., Borkmann, D., Fastabend, J., Herbert, T., Ahern, D., and Miller, D. The express data path: Fast programmable packet processing in the operating system kernel. In *Proceedings of the 14th International Conference on Emerging Networking EXperiments and Technologies* (New York, NY, USA, 2018), CoNEXT '18, Association for Computing Machinery, pp. 54–66. 24

[21] Hutchings, A. Socket sharding in nginx release 1.9.1. https://www.nginx.com/blog/socket-sharding-nginx-release-1-9-1/, May 2015. [Online; accessed September 2021]. 5

[22] Imply Corporation. Imply pivot. https://imply.io/. [Online; accessed September 2021]. 19

[23] Isovalent. Cilium. https://cilium.io/. [Online; accessed September 2021]. 12

[24] Iwashima, K. [rfc patch bpf-next 0/8] socket migration for so_reuseport. https://lore.kernel.org/lkml/20201117094023.3685-1-kuniyu@amazon.co.jp/, 2020. [Online; accessed September 2021]. 12

[25] Kerrisk, M. The so_reuseport socket option. https://lwn.net/Articles/542629/, March 2013. [Online; accessed September 2021]. 5

[26] Lau, M. K. [patch bpf-next 0/9] introduce bpf_map_type_reuseport_sockarray and bpf_prog_type_sk_reuseport. https://lore.kernel.org/netdev/20180808075917.3009181-1-kafai@fb.com/, 2018. [Online; accessed September 2021]. 23

[27] Linux Kernel Contributors. Bpf design q&a. https://www.kernel.org/doc/Documentation/bpf/bpf_design_QA.rst, 2021. [Online; accessed September 2021]. 38

[28] Linux Kernel Contributors. Cpu accounting controller. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/cpuacct.html, 2021. [Online; accessed September 2021]. 39

[29] Linux Kernel Contributors. Memory resource controller. https://www.kernel.org/doc/html/latest/admin-guide/cgroup-v1/memory.html, 2021. [Online; accessed September 2021]. 39

[30] Linux Kernel Documentation. *BPFTOOL-PROG(8)*, kernel 5.12 ed., June 2021. 29, 44

[31] The Linux man-pages project. *BPF-HELPERS(7) Linux Programmer's Manual*, 5.12 ed., March 2021. 8

[32] The Linux man-pages project. *BPF(2) Linux Programmer's Manual*, 5.12 ed., March 2021. 11, 32

[33] The Linux man-pages project. *GETSOCKOPT(2) Linux Programmer's Manual*, 5.12 ed., March 2021. 5, 30

[34] The Linux man-pages project. *SOCKET(7) Linux Programmer's Manual*, 5.12 ed., March 2021. 5, 6, 31

[35] Lucente, P., et al. pmacct. http://www.pmacct.net/. [Online; accessed September 2021]. 18

[36] McCanne, S., and Jacobson, V. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter* (1993), vol. 46. 8

[37] Miller, D. S. Bpf verifier overview. https://www.spinics.net/lists/xdp-newbies/msg00185.html, 2017. [Online; accessed September 2021]. 9

[38] Monnet, Q. Implementing ebpf for windows. https://lwn.net/Articles/857215/, June 2021. [Online; accessed September 2021]. 44

[39] Nakryiko, A. Bpf portability and co-re. https://facebookmicrosites.github.io/bpf/blog/2020/02/19/bpf-portability-and-co-re.html, February 2020. [Online; accessed September 2021]. 11

[40] Naseer, U., Niccolini, L., Pant, U., Frindell, A., Dasineni, R., and Benson, T. A. Zero downtime release: Disruption-free load balancing of a multi-billion user website. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication* (New York, NY, USA, 2020), SIGCOMM '20, Association for Computing Machinery, pp. 529–541. XIX

[41] Red Hat Customer Content Services. *A guide to configuring and managing networking in Red Hat Enterprise Linux 8, Chapter 52.*

Understanding the eBPF networking features in RHEL, 8 ed., May 2019. 24

[42] Rescorla, E. The transport layer security (tls) protocol version 1.3. RFC 8446, RFC Editor, August 2018. http://www.rfc-editor.org/rfc/rfc8446.txt. XVIII

[43] Robertson, A., et al. bpftrace. https://github.com/iovisor/bpftrace. [Online; accessed September 2021]. 10

[44] Sampic, T., Rosensthiel, A., Huang, A., Francois, P., and Frénot, S. C-collector for udp-notif. https://github.com/insa-unyte/udp-notif-c-collector. [Online; accessed September 2021]. 43

[45] Scudder, J., Fernando, R., and Stuart, S. Bgp monitoring protocol (bmp). RFC 7854, RFC Editor, June 2016. http://www.rfc-editor.org/rfc/rfc7854.txt. 1, XVII

[46] Sgier, L. Visualizing bgp rib changes into forwarding plane by leveraging bmp and ipfix. Master's thesis, ETH Zurich, Zurich, 2020-10. 13

[47] Sitnicki, J. [patch bpf-next v5 00/15] run a bpf program on socket lookup. https://lore.kernel.org/netdev/20200717103536.397595-1-jakub@cloudflare.com/, 2020. [Online; accessed September 2021]. 24

[48] Song, H., Qin, F., Martinez-Julia, P., Ciavaglia, L., and Wang, A. Network telemetry framework. Internet-Draft draft-ietf-opsawg-ntf-07, IETF Secretariat, February 2021. https://www.ietf.org/archive/id/draft-ietf-opsawg-ntf-07.txt. 1

[49] Starovoitov, A. [patch net-next 0/2] bpf: allow extended bpf programs access skb fields. https://lore.kernel.org/netdev/1426213271-8363-1-git-send-email-ast@plumgrid.com/, 2015. [Online; accessed September 2021]. 9

[50] Starovoitov, A. Safe programs, the foundation of bpf (ebpf summit 2020). https://youtu.be/AV8xY318rtc, November 2020. [Online; accessed September 2021]. 9

[51] Tarreau, W., et al. Haproxy. https://www.haproxy.org/. [Online; accessed September 2021]. 18

[52] Thaler, D., and Gaddehosur, P. Making ebpf work on windows. https://cloudblogs.microsoft.com/opensource/2021/05/10/making-ebpf-work-on-windows/, May 2021. [Online; accessed September 2021]. 44

[53] Tollini, M. bmp_scenarios. https://github.com/marcotollini/bmp_scenarios. [Online; accessed September 2021]. 39

[54] Torvalds, L., et al. Linux kernel. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=9f4ad9e425a1d3b6a34617b8ea226d56a119a717, 2021. 4

[55] Wang, N., et al. libbpf. https://github.com/libbpf/libbpf. [Online; accessed September 2021]. 11

[56] Zheng, G., Zhou, T., Graf, T., Francois, P., and Lucente, P. Udp-based transport for configured subscriptions. Internet-Draft draft-ietf-netconf-udp-notif-03, IETF Secretariat, July 2021. https://www.ietf.org/archive/id/draft-ietf-netconf-udp-notif-03.txt. 43

Appendix A

Looking closer at *nfacctd*

A.1 Execution of a BPF Userspace program

The following listings are traces of a cold start of nfaccd's BMP plugin containing the additional functionality. In particular, we are looking at invocations of the bpf() system call from start until nfacctd starts accepting connections.

We initialize the process by setting the SO_REUSEPORT on the socket represented by file descriptor 14. This will be the listening socket for network telemetry.

```
setsockopt(14, SOL_SOCKET, SO_REUSEPORT, [1], 4) = 0
```

Upon opening of the BPF ELF binary, libbpf analyzes the contents and extracts information about the programs it contains and maps it defines. This happens entirely in userspace within the execution of the calling thread. The first system interaction we see is libbpf performing a load of a trivial BPF program as sanity check to ensure basic BPF functionality is available with this kernel.

```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SOCKET_FILTER,
    ↪ insn_cnt=2, insns=0x7fd3ae54c280, license="GPL",
    ↪ log_level=0, log_size=0, log_buf=NULL, kern_version=
    ↪ KERNEL_VERSION(0, 0, 0), prog_flags=0, prog_name="",
    ↪ prog_ifindex=0, expected_attach_type=
    ↪ BPF_CGROUP_INET_INGRESS, prog_btf_fd=0,
    ↪ func_info_rec_size=0, func_info=NULL, func_info_cnt=0,
    ↪ line_info_rec_size=0, line_info=NULL, line_info_cnt=0},
    ↪ 120) = 15
```

Since this was successful, libbpf now loads the BTF debug information about our program and maps into the kernel. The program source line annotations in the listings in appendix B are synthesized from this information.

(The maps and program loads are, again, probes determining whether the kernel supports specific features; in this case: global data sections.)

```
bpf(BPF_BTF_LOAD, {btf="
    ↪ \237\353\1\0\30\0\0\0\0\0\0\20\0\0\0\20\0\0\0\5\0\0\0\1
    ↪  \0\0\0\0\0\0\1"..., btf_log_buf=NULL, btf_size=45,
    ↪ btf_log_size=0, btf_log_level=0}, 120) = 15
bpf(BPF_BTF_LOAD, {btf="
    ↪ \237\353\1\0\30\0\0\0\0\0\0000\0\0\0000\0\0\0\t
    ↪ \0\0\0\1\0\0\0\0\0\0\1"..., btf_log_buf=NULL, btf_size
    ↪ =81, btf_log_size=0, btf_log_level=0}, 120) = 15
bpf(BPF_BTF_LOAD, {btf="
    ↪ \237\353\1\0\30\0\0\0\0\0\0\08\0\0\08\0\0\0\t
    ↪ \0\0\0\0\0\0\0\0\0\0\1"..., btf_log_buf=NULL, btf_size
    ↪ =89, btf_log_size=0, btf_log_level=0}, 120) = 15
bpf(BPF_BTF_LOAD, {btf="\237\353\1\0\30\0\0\0\0\0\0\f\0\0\0\f
    ↪ \0\0\0\7\0\0\0\1\0\0\0\0\0\0\20"..., btf_log_buf=NULL,
    ↪ btf_size=43, btf_log_size=0, btf_log_level=0}, 120) = -1
    ↪ EINVAL (Invalid argument)
bpf(BPF_BTF_LOAD, {btf="
    ↪ \237\353\1\0\30\0\0\0\0\0\0000\0\0\0000\0\0\0\t
    ↪ \0\0\0\1\0\0\0\0\0\0\1"..., btf_log_buf=NULL, btf_size
    ↪ =81, btf_log_size=0, btf_log_level=0}, 120) = 15
bpf(BPF_BTF_LOAD, {btf="\237\353\1\0\30\0\0\0\0\0\0$\5\0\0$
    ↪ \5\0\0r\5\0\0\0\0\0\0\5\0\0\4"..., btf_log_buf=NULL,
    ↪ btf_size=2734, btf_log_size=0, btf_log_level=0}, 120) =
    ↪ 15
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=32, max_entries=1, map_flags=0, inner_map_fd
    ↪ =0, map_name="", map_ifindex=0, btf_fd=0, btf_key_type_id
    ↪ =0, btf_value_type_id=0}, 120) = 16
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SOCKET_FILTER,
    ↪ insn_cnt=5, insns=0x7fd3ae54c0b0, license="GPL",
    ↪ log_level=0, log_size=0, log_buf=NULL, kern_version=
    ↪ KERNEL_VERSION(0, 0, 0), prog_flags=0, prog_name="",
    ↪ prog_ifindex=0, expected_attach_type=
    ↪ BPF_CGROUP_INET_INGRESS, prog_btf_fd=0,
    ↪ func_info_rec_size=0, func_info=NULL, func_info_cnt=0,
    ↪ line_info_rec_size=0, line_info=NULL, line_info_cnt=0},
    ↪ 120) = 17
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=4, max_entries=1, map_flags=0x400 /* BPF_F_???
    ↪  */, inner_map_fd=0, map_name="", map_ifindex=0, btf_fd
    ↪ =0, btf_key_type_id=0, btf_value_type_id=0}, 120) = 16
```

With these support operations completed for us, we can now deal with the BPF maps, starting with *tcp_balancing_targets*. Since this map is intended to be pinned at a custom path in BPFFS, we first try to find it there. Since that proved unsuccessful, we then create it from scratch and pin it to the intended path. Note that the *btf_fd* argument explicitly ties the map to the freshly loaded BTF debug information.

(The program load command is probing the kernel for support for the *prog_name* and *map_name* attributes.)

```
bpf(BPF_OBJ_GET, {pathname="/sys/fs/bpf/pmacct/
    ↪ tcp_balancing_targets", bpf_fd=0, file_flags=0}, 120) =
    ↪ -1 ENOENT (No such file or directory)
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_SOCKET_FILTER,
    ↪ insn_cnt=2, insns=0x7fd3ae54bf00, license="GPL",
    ↪ log_level=0, log_size=0, log_buf=NULL, kern_version=
    ↪ KERNEL_VERSION(0, 0, 0), prog_flags=0, prog_name="test",
    ↪ prog_ifindex=0, expected_attach_type=
    ↪ BPF_CGROUP_INET_INGRESS, prog_btf_fd=0,
    ↪ func_info_rec_size=0, func_info=NULL, func_info_cnt=0,
    ↪ line_info_rec_size=0, line_info=NULL, line_info_cnt=0},
    ↪ 120) = 16
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
    ↪  key_size=4, value_size=8, max_entries=128, map_flags=0,
    ↪ inner_map_fd=0, map_name="tcp_balancing_t", map_ifindex
    ↪ =0, btf_fd=15, btf_key_type_id=7, btf_value_type_id=11},
    ↪ 120) = 16
bpf(BPF_OBJ_PIN, {pathname="/sys/fs/bpf/pmacct/
    ↪ tcp_balancing_targets", bpf_fd=16, file_flags=0}, 120) =
    ↪ 0
```

This process is repeated for each of the remaining three maps:

```
bpf(BPF_OBJ_GET, {pathname="/sys/fs/bpf/pmacct/
    ↪ udp_balancing_targets", bpf_fd=0, file_flags=0}, 120) =
    ↪ -1 ENOENT (No such file or directory)
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_REUSEPORT_SOCKARRAY,
    ↪  key_size=4, value_size=8, max_entries=128, map_flags=0,
    ↪ inner_map_fd=0, map_name="udp_balancing_t", map_ifindex
    ↪ =0, btf_fd=15, btf_key_type_id=7, btf_value_type_id=11},
    ↪ 120) = 17
bpf(BPF_OBJ_PIN, {pathname="/sys/fs/bpf/pmacct/
    ↪ udp_balancing_targets", bpf_fd=17, file_flags=0}, 120) =
    ↪ 0

bpf(BPF_OBJ_GET, {pathname="/sys/fs/bpf/pmacct/size", bpf_fd=0,
    ↪  file_flags=0}, 120) = -1 ENOENT (No such file or
    ↪ directory)
```

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=4, max_entries=1, map_flags=0, inner_map_fd=0,
    ↪  map_name="size", map_ifindex=0, btf_fd=15,
    ↪ btf_key_type_id=7, btf_value_type_id=7}, 120) = 18
bpf(BPF_OBJ_PIN, {pathname="/sys/fs/bpf/pmacct/size", bpf_fd
    ↪ =18, file_flags=0}, 120) = 0

bpf(BPF_OBJ_GET, {pathname="/sys/fs/bpf/pmacct/nonce", bpf_fd
    ↪ =0, file_flags=0}, 120) = -1 ENOENT (No such file or
    ↪ directory)
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=4, max_entries=1, map_flags=0, inner_map_fd=0,
    ↪  map_name="nonce", map_ifindex=0, btf_fd=15,
    ↪ btf_key_type_id=7, btf_value_type_id=7}, 120) = 19
bpf(BPF_OBJ_PIN, {pathname="/sys/fs/bpf/pmacct/nonce", bpf_fd
    ↪ =19, file_flags=0}, 120) = 0
```

All user-defined maps are now present. However, there is one more map that is implicitly created during BPF ELF binary compilation. That map is then frozen to prevent modifications using the BPF_MAP_FREEZE command (enum identifier 16_{16}).

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=8, max_entries=1, map_flags=0x480 /* BPF_F_???
    ↪  */, inner_map_fd=0, map_name="reusepor.rodata",
    ↪ map_ifindex=0, btf_fd=15, btf_key_type_id=0,
    ↪ btf_value_type_id=50}, 120) = 20
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=20, key=0x7fd3ae54c0d0, value
    ↪ =0x7fd3b5437000, flags=BPF_ANY}, 120) = 0
bpf(0x16 /* BPF_??? */, 0x7fd3ae54c010, 120) = 0
```

At this point, we load the BPF program itself. libbpf performs any applicable CO-RE relocation, resolves references to maps with the previously obtained file descriptors, and actually performs the loading. Unfortunately, *strace* incorrectly identifies the program type as FLOW_DISSECTOR. The program itself works correctly, however. Note that the CGROUP_INET_INGRESS attachment point is misleading for SK_REUSEPORT type programs. It is simply the symbolic name of the enum value 0.

(The loading of the CGROUP_SOCK BPF program is part of the feature detection for *expected_attach_type* field in the system call argument.)

```
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_CGROUP_SOCK,
    ↪ insn_cnt=2, insns=0x7fd3ae54bf60, license="GPL",
    ↪ log_level=0, log_size=0, log_buf=NULL, kern_version=
    ↪ KERNEL_VERSION(0, 0, 0), prog_flags=0, prog_name="",
    ↪ prog_ifindex=0, expected_attach_type=
```

```
    ↪ BPF_CGROUP_INET_SOCK_CREATE, prog_btf_fd=0,
    ↪ func_info_rec_size=0, func_info=NULL, func_info_cnt=0,
    ↪ line_info_rec_size=0, line_info=NULL, line_info_cnt=0},
    ↪ 120) = 21
bpf(BPF_PROG_LOAD, {prog_type=BPF_PROG_TYPE_FLOW_DISSECTOR,
    ↪ insn_cnt=144, insns=0x7fd3a810b5b0, license="GPL",
    ↪ log_level=0, log_size=0, log_buf=NULL, kern_version=
    ↪ KERNEL_VERSION(4, 18, 0), prog_flags=0, prog_name="
    ↪ _selector", prog_ifindex=0, expected_attach_type=
    ↪ BPF_CGROUP_INET_INGRESS, prog_btf_fd=15,
    ↪ func_info_rec_size=8, func_info=0x7fd3a80f9210,
    ↪ func_info_cnt=1, line_info_rec_size=16, line_info=0
    ↪ x7fd3a8109d70, line_info_cnt=45}, 120) = 21
```

In a next step libbpf attempts to bind the metadata map to the program to prevent deletion caused by a zero reference count. On the target machine this BPF system call command is supported by neither *strace* nor the kernel itself.

```
bpf(BPF_MAP_CREATE, {map_type=BPF_MAP_TYPE_ARRAY, key_size=4,
    ↪ value_size=32, max_entries=1, map_flags=0, inner_map_fd
    ↪ =0, map_name="", map_ifindex=0, btf_fd=0, btf_key_type_id
    ↪ =0, btf_value_type_id=0}, 120) = 22
bpf(0x23 /* BPF_??? */, 0x7fd3ae54be60, 120) = -1 EINVAL (
    ↪ Invalid argument)
```

nfacctd now reads the *size* map to inspect if it is already set—which is not the case—and intializes accordingly.

```
bpf(BPF_MAP_LOOKUP_ELEM, {map_fd=18, key=0x7fd3ae54c4b4, value
    ↪ =0x7fd3ae54c4c0}, 120) = 0
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=18, key=0x7fd3ae54c4b4, value
    ↪ =0x7fd3ae54c4c0, flags=BPF_ANY}, 120) = 0
```

Now, nfacctd can attach the BPF program to the socket.

```
setsockopt(14, SOL_SOCKET, SO_ATTACH_REUSEPORT_EBPF, [21], 4) =
    ↪  0
```

Finally, nfacctd registers itself in the first hash bucket for incoming TCP traffic.

```
bpf(BPF_MAP_UPDATE_ELEM, {map_fd=16, key=0x7fd3ae54c4f0, value
    ↪ =0x7fd3ae54c610, flags=BPF_ANY}, 120) = 0
```

(The remaining program execution does no longer involve BPF.)

Appendix B

Looking *closely* at *reuseport_kern*

reuseport_kern is the name of BPF ELF binary. The actual program is simply named *_selector()*. For brevity, the program was re-compiled without the code paths used for diagnostics and debugging.

B.1 Intermediate Representation

Lines starting with semicolons are the original C code and correspond to the intermediate representation of the BPF byte code in subsequent lines. For nested logic (e.g., a memory load within a switch statement), the line may be repeated multiple times.

```
enum sk_action _selector(struct sk_reuseport_md * reuse):
; enum sk_action _selector(struct sk_reuseport_md *reuse) {
   0: (bf) r6 = r1
   1: (18) r2 = map[id:1214]
; switch (reuse->ip_protocol) {
   3: (79) r1 = *(u64 *)(r6 +8)
   4: (61) r1 = *(u32 *)(r1 +536)
   5: (54) w1 &= 65280
   6: (74) w1 >>= 8
; switch (reuse->ip_protocol) {
   7: (15) if r1 == 0x6 goto pc+4
   8: (b7) r0 = 0
   9: (55) if r1 != 0x11 goto pc+151
  10: (18) r2 = map[id:1215]
  12: (7b) *(u64 *)(r10 -40) = r2
```

```
  13: (bf) r3 = r10
;
  14: (07) r3 += -24
; bpf_skb_load_bytes_relative(reuse, 0, &ip, sizeof(struct
    ↪ iphdr), (u32)BPF_HDR_START_NET);
  15: (bf) r1 = r6
  16: (b7) r2 = 0
  17: (b7) r4 = 20
  18: (b7) r5 = 1
  19: (85) call sk_reuseport_load_bytes_relative#5774496
; const u32 *balancer_count = bpf_map_lookup_elem(&size, &zero)
    ↪ ;
  20: (18) r1 = map[id:1216]
  22: (18) r2 = map[id:1274][0]+0
  24: (07) r1 += 272
  25: (61) r0 = *(u32 *)(r2 +0)
  26: (35) if r0 >= 0x1 goto pc+3
  27: (67) r0 <<= 3
  28: (0f) r0 += r1
  29: (05) goto pc+1
  30: (b7) r0 = 0
  31: (bf) r8 = r0
; if (!balancer_count || *balancer_count == 0) {  //
    ↪ uninitialized by userspace
  32: (15) if r8 == 0x0 goto pc+2
; if (!balancer_count || *balancer_count == 0) {  //
    ↪ uninitialized by userspace
  33: (61) r1 = *(u32 *)(r8 +0)
; if (!balancer_count || *balancer_count == 0) {  //
    ↪ uninitialized by userspace
  34: (55) if r1 != 0x0 goto pc+10
; bpf_map_update_elem(&size, &zero, balancer_count, BPF_ANY);
  35: (18) r8 = map[id:1274][0]+4
  37: (18) r1 = map[id:1216]
  39: (18) r2 = map[id:1274][0]+0
  41: (18) r3 = map[id:1274][0]+4
  43: (b7) r4 = 0
  44: (85) call array_map_update_elem#124624
  45: (bf) r1 = r10
;
  46: (07) r1 += -24
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
  47: (61) r7 = *(u32 *)(r1 +12)
; n = bpf_map_lookup_elem(&nonce, &zero);
  48: (18) r1 = map[id:1217]
  50: (18) r2 = map[id:1274][0]+0
  52: (07) r1 += 272
  53: (61) r0 = *(u32 *)(r2 +0)
  54: (35) if r0 >= 0x1 goto pc+3
```

```
  55: (67) r0 <<= 3
  56: (0f) r0 += r1
  57: (05) goto pc+1
  58: (b7) r0 = 0
  59: (bf) r9 = r0
;
  60: (b7) r2 = 0
; if (n == 0) {
  61: (15) if r9 == 0x0 goto pc+78
;
  62: (dc) r7 = be32 r7
; if (*n == 0) {
  63: (61) r0 = *(u32 *)(r9 +0)
; if (*n == 0) {
  64: (55) if r0 != 0x0 goto pc+2
; *n = bpf_get_prandom_u32();
  65: (85) call bpf_user_rnd_u32#13456
; *n = bpf_get_prandom_u32();
  66: (63) *(u32 *)(r9 +0) = r0
; initval += JHASH_INITVAL + (3 << 2);
  67: (07) r0 += -559038725
  68: (18) r1 = 0xfffc0000
  70: (bf) r3 = r0
  71: (5f) r3 &= r1
; static inline __u32 rol32(__u32 word, unsigned int shift) {
   ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
   ↪ ; }
  72: (77) r3 >>= 18
  73: (bf) r2 = r0
  74: (67) r2 <<= 14
  75: (4f) r2 |= r3
; static inline __u32 rol32(__u32 word, unsigned int shift) {
   ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
   ↪ ; }
  76: (bf) r4 = r2
  77: (67) r4 <<= 11
; __jhash_final(a, b, c);
  78: (87) r2 = -r2
  79: (18) r3 = 0xffe00000
  81: (bf) r5 = r2
  82: (5f) r5 &= r3
; static inline __u32 rol32(__u32 word, unsigned int shift) {
   ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
   ↪ ; }
  83: (77) r5 >>= 21
; __jhash_final(a, b, c);
  84: (1f) r4 -= r5
; a = ip + initval;
  85: (bf) r3 = r0
```

```
  86: (0f) r3 += r7
; __jhash_final(a, b, c);
  87: (af) r3 ^= r2
  88: (0f) r3 += r4
  89: (18) r4 = 0xffffff80
  91: (bf) r5 = r3
  92: (5f) r5 &= r4
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
  93: (77) r5 >>= 7
  94: (bf) r7 = r3
  95: (67) r7 <<= 25
  96: (4f) r7 |= r5
; __jhash_final(a, b, c);
  97: (bf) r4 = r3
  98: (af) r4 ^= r0
  99: (1f) r4 -= r7
 100: (18) r5 = 0xffff0000
 102: (bf) r0 = r4
 103: (5f) r0 &= r5
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 104: (77) r0 >>= 16
 105: (bf) r7 = r4
 106: (67) r7 <<= 16
 107: (4f) r7 |= r0
; __jhash_final(a, b, c);
 108: (bf) r5 = r4
 109: (af) r5 ^= r2
 110: (1f) r5 -= r7
 111: (18) r2 = 0xf0000000
 113: (bf) r0 = r5
 114: (5f) r0 &= r2
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 115: (77) r0 >>= 28
 116: (bf) r7 = r5
 117: (67) r7 <<= 4
 118: (4f) r7 |= r0
; __jhash_final(a, b, c);
 119: (bf) r2 = r5
 120: (af) r2 ^= r3
 121: (1f) r2 -= r7
 122: (bf) r3 = r2
 123: (5f) r3 &= r1
; static inline __u32 rol32(__u32 word, unsigned int shift) {
```

```
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 124: (77) r3 >>= 18
 125: (bf) r1 = r2
 126: (67) r1 <<= 14
 127: (4f) r1 |= r3
; __jhash_final(a, b, c);
 128: (af) r2 ^= r4
 129: (1f) r2 -= r1
 130: (18) r1 = 0xffffff00
 132: (bf) r3 = r2
 133: (5f) r3 &= r1
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 134: (77) r3 >>= 8
 135: (bf) r1 = r2
 136: (67) r1 <<= 24
 137: (4f) r1 |= r3
; __jhash_final(a, b, c);
 138: (af) r2 ^= r5
 139: (1f) r2 -= r1
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 140: (61) r1 = *(u32 *)(r8 +0)
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 141: (bf) r3 = r2
 142: (67) r3 <<= 32
 143: (77) r3 >>= 32
 144: (55) if r1 != 0x0 goto pc+2
 145: (ac) w3 ^= w3
 146: (05) goto pc+1
 147: (3f) r3 /= r1
 148: (2f) r3 *= r1
 149: (1f) r2 -= r3
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 150: (63) *(u32 *)(r10 -28) = r2
 151: (bf) r3 = r10
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 152: (07) r3 += -28
; if (bpf_sk_select_reuseport(reuse, targets, &key, 0) == 0) {
 153: (bf) r1 = r6
 154: (79) r2 = *(u64 *)(r10 -40)
 155: (b7) r4 = 0
 156: (85) call sk_select_reuseport#5773424
 157: (bf) r1 = r0
 158: (b7) r0 = 1
; if (bpf_sk_select_reuseport(reuse, targets, &key, 0) == 0) {
 159: (15) if r1 == 0x0 goto pc+1
 160: (b7) r0 = 0
```

```
; }
 161: (95) exit
```

B.2 JIT-Compiled Program

Lines starting with semicolons are the original C code and correspond to the x86-64 machine code in subsequent lines. For nested logic (e.g., a memory load within a switch statement), the line may be repeated multiple times.

```
enum sk_action _selector(struct sk_reuseport_md * reuse):
0xffffffffc03e07fc:
; enum sk_action _selector(struct sk_reuseport_md *reuse) {
   0:   nopl   0x0(%rax,%rax,1)
   5:   push   %rbp
   6:   mov    %rsp,%rbp
   9:   sub    $0x28,%rsp
  10:   push   %rbx
  11:   push   %r13
  13:   push   %r14
  15:   push   %r15
  17:   pushq  $0x0
  19:   mov    %rdi,%rbx
  1c:   movabs $0xffff8d38ee573800,%rsi
; switch (reuse->ip_protocol) {
  26:   mov    0x8(%rbx),%rdi
  2a:   mov    0x218(%rdi),%edi
  30:   and    $0xff00,%edi
  36:   shr    $0x8,%edi
; switch (reuse->ip_protocol) {
  39:   cmp    $0x6,%rdi
  3d:   je     0x0000000000000055
  3f:   xor    %eax,%eax
  41:   cmp    $0x11,%rdi
  45:   jne    0x000000000000028d
  4b:   movabs $0xffff8d38ee576800,%rsi
  55:   mov    %rsi,-0x28(%rbp)
  59:   mov    %rbp,%rdx
;
  5c:   add    $0xfffffffffffffe8,%rdx
; bpf_skb_load_bytes_relative(reuse, 0, &ip, sizeof(struct
    ↪ iphdr), (u32)BPF_HDR_START_NET);
  60:   mov    %rbx,%rdi
  63:   xor    %esi,%esi
  65:   mov    $0x14,%ecx
  6a:   mov    $0x1,%r8d
  70:   callq  0xffffffffd117d1b4
```

```
; const u32 *balancer_count = bpf_map_lookup_elem(&size, &zero)
   ↪ ;
  75:   movabs $0xffff8d2a9d2cb200,%rdi
  7f:   movabs $0xffffa5a506278000,%rsi
  89:   add    $0x110,%rdi
  90:   mov    0x0(%rsi),%eax
  93:   cmp    $0x1,%rax
  97:   jae    0x00000000000000a2
  99:   shl    $0x3,%rax
  9d:   add    %rdi,%rax
  a0:   jmp    0x00000000000000a4
  a2:   xor    %eax,%eax
  a4:   mov    %rax,%r14
; if (!balancer_count || *balancer_count == 0) {  //
   ↪ uninitialized by userspace
  a7:   test   %r14,%r14
  aa:   je     0x00000000000000b5
; if (!balancer_count || *balancer_count == 0) {  //
   ↪ uninitialized by userspace
  ac:   mov    0x0(%r14),%edi
; if (!balancer_count || *balancer_count == 0) {  //
   ↪ uninitialized by userspace
  b0:   test   %rdi,%rdi
  b3:   jne    0x00000000000000e4
; bpf_map_update_elem(&size, &zero, balancer_count, BPF_ANY);
  b5:   movabs $0xffffa5a506278004,%r14
  bf:   movabs $0xffff8d2a9d2cb200,%rdi
  c9:   movabs $0xffffa5a506278000,%rsi
  d3:   movabs $0xffffa5a506278004,%rdx
  dd:   xor    %ecx,%ecx
  df:   callq  0xffffffffd0c19be4
  e4:   mov    %rbp,%rdi
;
  e7:   add    $0xfffffffffffffe8,%rdi
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
  eb:   mov    0xc(%rdi),%r13d
; n = bpf_map_lookup_elem(&nonce, &zero);
  ef:   movabs $0xffff8d2a9d2c8000,%rdi
  f9:   movabs $0xffffa5a506278000,%rsi
 103:   add    $0x110,%rdi
 10a:   mov    0x0(%rsi),%eax
 10d:   cmp    $0x1,%rax
 111:   jae    0x000000000000011c
 113:   shl    $0x3,%rax
 117:   add    %rdi,%rax
 11a:   jmp    0x000000000000011e
 11c:   xor    %eax,%eax
 11e:   mov    %rax,%r15
;
```

```
 121:    xor    %esi,%esi
; if (n == 0) {
 123:    test   %r15,%r15
 126:    je     0x0000000000000228
;
 12c:    bswap  %r13d
; if (*n == 0) {
 12f:    mov    0x0(%r15),%eax
; if (*n == 0) {
 133:    test   %rax,%rax
 136:    jne    0x0000000000000141
; *n = bpf_get_prandom_u32();
 138:    callq  0xffffffffd0bfe9a4
; *n = bpf_get_prandom_u32();
 13d:    mov    %eax,0x0(%r15)
; initval += JHASH_INITVAL + (3 << 2);
 141:    add    $0xffffffffdeadbefb,%rax
 147:    mov    $0xfffc0000,%edi
 14c:    mov    %rax,%rdx
 14f:    and    %rdi,%rdx
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 152:    shr    $0x12,%rdx
 156:    mov    %rax,%rsi
 159:    shl    $0xe,%rsi
 15d:    or     %rdx,%rsi
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 160:    mov    %rsi,%rcx
 163:    shl    $0xb,%rcx
; __jhash_final(a, b, c);
 167:    neg    %rsi
 16a:    mov    $0xffe00000,%edx
 16f:    mov    %rsi,%r8
 172:    and    %rdx,%r8
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 175:    shr    $0x15,%r8
; __jhash_final(a, b, c);
 179:    sub    %r8,%rcx
; a = ip + initval;
 17c:    mov    %rax,%rdx
 17f:    add    %r13,%rdx
; __jhash_final(a, b, c);
 182:    xor    %rsi,%rdx
 185:    add    %rcx,%rdx
```

```
 188:    mov     $0xffffff80,%ecx
 18d:    mov     %rdx,%r8
 190:    and     %rcx,%r8
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 193:    shr     $0x7,%r8
 197:    mov     %rdx,%r13
 19a:    shl     $0x19,%r13
 19e:    or      %r8,%r13
; __jhash_final(a, b, c);
 1a1:    mov     %rdx,%rcx
 1a4:    xor     %rax,%rcx
 1a7:    sub     %r13,%rcx
 1aa:    mov     $0xffff0000,%r8d
 1b0:    mov     %rcx,%rax
 1b3:    and     %r8,%rax
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 1b6:    shr     $0x10,%rax
 1ba:    mov     %rcx,%r13
 1bd:    shl     $0x10,%r13
 1c1:    or      %rax,%r13
; __jhash_final(a, b, c);
 1c4:    mov     %rcx,%r8
 1c7:    xor     %rsi,%r8
 1ca:    sub     %r13,%r8
 1cd:    mov     $0xf0000000,%esi
 1d2:    mov     %r8,%rax
 1d5:    and     %rsi,%rax
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 1d8:    shr     $0x1c,%rax
 1dc:    mov     %r8,%r13
 1df:    shl     $0x4,%r13
 1e3:    or      %rax,%r13
; __jhash_final(a, b, c);
 1e6:    mov     %r8,%rsi
 1e9:    xor     %rdx,%rsi
 1ec:    sub     %r13,%rsi
 1ef:    mov     %rsi,%rdx
 1f2:    and     %rdi,%rdx
; static inline __u32 rol32(__u32 word, unsigned int shift) {
    ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
    ↪ ; }
 1f5:    shr     $0x12,%rdx
 1f9:    mov     %rsi,%rdi
```

```
 1fc:   shl    $0xe,%rdi
 200:   or     %rdx,%rdi
; __jhash_final(a, b, c);
 203:   xor    %rcx,%rsi
 206:   sub    %rdi,%rsi
 209:   mov    $0xffffff00,%edi
 20e:   mov    %rsi,%rdx
 211:   and    %rdi,%rdx
; static inline __u32 rol32(__u32 word, unsigned int shift) {
   ↪ return (word << (shift & 31)) | (word >> ((-shift) & 31))
   ↪ ; }
 214:   shr    $0x8,%rdx
 218:   mov    %rsi,%rdi
 21b:   shl    $0x18,%rdi
 21f:   or     %rdx,%rdi
; __jhash_final(a, b, c);
 222:   xor    %r8,%rsi
 225:   sub    %rdi,%rsi
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 228:   mov    0x0(%r14),%edi
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 22c:   mov    %rsi,%rdx
 22f:   shl    $0x20,%rdx
 233:   shr    $0x20,%rdx
 237:   test   %rdi,%rdi
 23a:   jne    0x0000000000000240
 23c:   xor    %edx,%edx
 23e:   jmp    0x0000000000000255
 240:   push   %rax
 241:   push   %rdx
 242:   mov    %rdi,%r11
 245:   mov    %rdx,%rax
 248:   xor    %edx,%edx
 24a:   div    %r11
 24d:   mov    %rax,%r11
 250:   pop    %rdx
 251:   pop    %rax
 252:   mov    %r11,%rdx
 255:   push   %rax
 256:   mov    %rdx,%r11
 259:   mov    %rdi,%rax
 25c:   mul    %r11
 25f:   mov    %rax,%rdx
 262:   pop    %rax
 263:   sub    %rdx,%rsi
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
 266:   mov    %esi,-0x1c(%rbp)
 269:   mov    %rbp,%rdx
; key = hash(__builtin_bswap32(ip.saddr)) % *balancer_count;
```

```
 26c:   add    $0xfffffffffffffe4,%rdx
; if (bpf_sk_select_reuseport(reuse, targets, &key, 0) == 0) {
 270:   mov    %rbx,%rdi
 273:   mov    -0x28(%rbp),%rsi
 277:   xor    %ecx,%ecx
 279:   callq  0xffffffffd117cd84
 27e:   mov    %rax,%rdi
 281:   mov    $0x1,%eax
; if (bpf_sk_select_reuseport(reuse, targets, &key, 0) == 0) {
 286:   test   %rdi,%rdi
 289:   je     0x000000000000028d
 28b:   xor    %eax,%eax
; }
 28d:   pop    %rbx
 28e:   pop    %r15
 290:   pop    %r14
 292:   pop    %r13
 294:   pop    %rbx
 295:   leaveq
 296:   retq
```

Appendix C

BMP Session Handoff Challenges

Our design has no provisions for dynamically moving traffic from one collector daemon ("monitoring station" in BMP parlance) to another, neither for failover nor for live re-balancing. One of the consequences is that it relies on manual actions performed by the administrator to change the number of collectors. This is motivated by the high degree of implicit state present in the BMP session design. Effectively, the application layer protocol is tightly coupled to the transport protocol.

As specified in section 3.3 of its RFC [45], the lifetime of the BMP session is tied to the lifetime of the TCP connection itself. As the protocol is purely unidirectional (section 3.2 of the RFC), each new BMP session establishment is usually followed by a high traffic burst due to the mandatory initial export of all of the router's BGP RIBs.

This is problematic in multiple ways. First, this behavior does not account for (short) network interruptions, and triggers costly state snapshot transfers even when they provide no new information relative to already buffered, incremental messages. Second, there is no inherent concept of session migration from one monitoring station to another on the same host.[1] Finally, there is the issue of resources. Given that the load profile shows up to an 80× difference in message rates between the initial burst and the steady-state operation, it can be tempting to provision for a non-peak load. This effectively amounts to an over-subscription of computing resources, a high-

[1] Note that we are not addressing IP mobility here. Its challenges are similar in kind, but larger in degree and with fewer options to address them.

risk situation in the face of host isolation or reboots. While CPU access can be throttled, running out of memory is much more difficult to handle. Although the instantaneous impact can be spread by rate-limiting TCP connection establishment, this proportionately reduces the value of near-realtime monitoring.

We will elaborate on multiple, partially composable ways to address some of these limitations. Each of these covers at least one of four aspects relevant for correct operation:

1. The router-internal state.
2. The state of the TCP connection, as seen by the router and the monitoring station host.
3. The state of the socket on the monitoring station, separately from (2).
4. The state of the monitoring station itself, which is necessary to correctly interpret an incoming BMP message. (Initially created by the router's state dump.)

Opportunistic session resumption. One option that is light on semantic changes is to stretch the definition of session termination by introducing the concept of session resumption. [17] is a proposal to do just that, built on TCP Fast Open (TFO) [7] and a timeout-based opportunistic resumption. The synchronization on session identity introduced by the notion of resumption is implicitly resolved through the TFO cookie.

This approach addresses the issue of short network interruptions and is fully compatible with our SO_REUSEPORT & BPF based design as well since the socket lockup and connection establishment are independent in this regard. TLS 1.3's 0-RTT data [42] could be used in a similar way at a higher layer, instead, should TFO not be available (or in addition if encryption was desired).

However, this approach will not help with issues arising from the lifecycle of the process acting as the monitoring station, such as a restart.

BMP protocol changes. The state dump following every new BMP session establishment is inherent to the protocol as there is no way for the monitoring station to indicate whether this state dump is desired. Introducing this capability both at initiation time and at arbitrary points during

the session lifetime constitutes another option. Here, the monitoring station could ensure that state dumps are only triggered as a last resort.

This approach requires additional monitoring station-side logic in determining correct message ordering. In the original protocol, this sequencing was implicit through the use of in-order, reliable message delivery. Conceptually, this can be thought of as using TCP sequence numbers imposing the ordering. Since the BMP message sequence is no longer confined to a single TCP connection, but allowing for possible arbitrary "resumptions," it is now up to the monitoring station to determine whether it has built up enough internal state to interpret the message correctly (or whether to request a new state dump). In terms of functionality, this supersedes the TFO approach. The shortened handshake of TFO may still be considered desirable though.

This option is fully compatible with our SO_REUSEPORT & BPF based design and should be backward compatible with BMP clients running the original protocol version.

Client-based socket migration. No changes to the protocol are necessary when using socket migration. In this approach, whenever a monitoring station is scheduled for shutdown, e.g., for an update to the BMP server binary, it would send the file descriptor referencing the established socket to another monitoring station over Unix Domain Socket for it to take over.

One example socket migration scheme in use is described in detail in [40]. These schemes, however, impose much additional complexity on the implementation intricacies of the monitoring station. For example, the migration target monitoring station has obtained enough state from the migration source to correctly interpret the new messages within a BMP session it has not established itself.

Another limiting factor is that these schemes are usually cooperative. If the main process terminates suddenly, there is nothing left to execute the migration.

This approach is fully compatible with our SO_REUSEPORT & BPF based design as the established socket is functionally independent of the listening one. However, additional work would be necessary to ensure a synchronized failover covering both transport protocols.

Host-based socket migration Socket migration can also be partially offloaded from the application to the host. Using the TCP connection repair mechanism [11], sockets can be created with a custom initial state rather

than by traversing the TCP state machine while consuming random inputs (e.g., the initial sequence numbers). With the monitoring stations doing periodic check-pointing to a shared medium, it could be possible to recover the state even in the context of sudden termination, thus effectively creating transparent failover. The Checkpoint/Restore In Userspace (CRIU) project is an example user of this mechanism [14]. A host-boundary crossing scheme is described in [6].

We were unable to determine compatibility with our SO_REUSEPORT & BPF based design.

Completely custom approach. Rather than relying on the kernel at all, the last approach would use XDP to filter the relevant TCP segments and pass them up to userspace to handle and implement arbitrary business logic, effectively replacing our design in the process. However: with great power comes great responsibility.