





GALETTE: a Lightweight XDP Dataplane on your Raspberry Pi

Kyle A. Simpson  [0000-0001-8068-9909], Chris Williamson  [0000-0003-1108-9402],
Douglas J. Paul  [0000-0001-7402-8530], Dimitrios P. Pazaros  [0000-0003-0939-378X]
University of Glasgow, Glasgow, Scotland
kylesimpson1@acm.org

Abstract—IoT and sensor networks are now a critical part of public infrastructure. At the same time, they remain infamous for becoming insecure as new exploits arise. Software dataplanes give us the power to retrofit security functions, and are well-researched in datacentres. Yet the server-grade hardware such frameworks are optimised for is a poor fit for vulnerable low-power, low-space IoT gateways. *Single-board computers* (SBCs) are a cheaper and better fit on all these metrics, yet no *service function chaining* (SFC) approaches are tailored to these devices. In addition, modern OS features like XDP give us the capability to minimise power use and provide the lowest latency processing these devices can offer—meaning quicker response to network events, suited to the needs of the network edge.

We present GALETTE, a device-portable SFC framework designed for the inexpensive defence of IoT networks by SBCs. GALETTE builds on Linux’s XDP framework to provide a CPU-efficient, low latency dataplane. Due to SBC hardware designs, we divide traffic between an XDP fast path and userland, which lets us schedule expensive packet analysis without harming normal traffic. Our API makes it easy to write network functions (NFs) which compile to both eBPF and native code, while being portable across heterogeneous SBCs. Testbed evaluations show how this is more efficient, faster, and uses less power than *AF_PACKET* on Raspberry Pi.

I. INTRODUCTION

IoT and sensor networks—particularly legacy installations—are infamous for their (perceived) lack of security. Earlier devices have often been built without incorporating ‘security by design’, leading to widespread compromise via malware like *Mirai* [1]. Although such high-publicity mishaps have caused a sea change in how security is treated by vendors and legislation [31], long-term rollout of security updates can be hard to guarantee as new exploits and attacks become public. Yet when the sensor networks built on such abandoned components underpin critical infrastructure, this leaves administrators in the difficult position of having to pursue fleetwide device replacements. This can be prohibitively expensive—particularly if these vulnerable networks are physically remote. Ideally, we desire an inexpensive and dynamically reconfigurable way to retrofit security functions into such networks with a single up-front cost as the threat landscape evolves.

Software dataplanes are a powerful, long-standing tool for flexible network traffic processing, and offer the necessary flexibility to defend such networks. Since *Click*’s [16] debut decades ago, *Service Function Chaining* (SFC) has been key in realising composable packet processing for measurement and security. The community seeks always to improve its

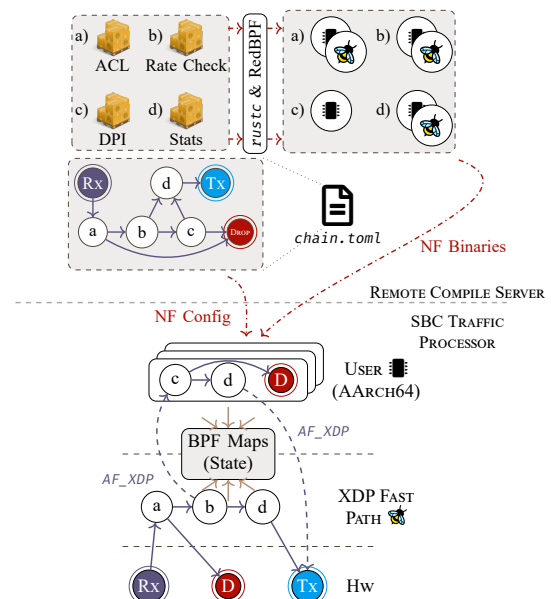


Figure 1. GALETTE splits traffic processing between an XDP fast path and userland, which makes the best possible use of SBC device resources. Individual NFs are written in Rust and remotely compiled to native and eBPF code, written with a single API that abstracts over runtime details.

design and performance: via accelerated network stacks like *DPDK* [2], safer language tooling [19], or novel hardware features like *Trusted Execution Environments* (TEEs) to improve *security* [21] in untrusted, multitenant infrastructure. As a result, we can now attest that packets are processed by the right functions without interference at high traffic rates—provided we have high-performance, well-supported hardware.

Yet the limited space and power available in IoT or edge deployments—to say nothing of their physical vulnerability—mean that we would be better served by inexpensive, low-power devices than by server-class hardware. *Single-Board Computers* (SBCs) like Raspberry Pis fall into this category, but are computationally weaker while lacking the hardware and driver support necessary for the above advances in SFC, like *DPDK* support and TEEs. Additionally, such SBCs often run Linux, allowing us to run standard software and take advantage of advances in its kernel and network stack.

To our benefit, raw computational power is not always needed. While SFCs at datacentre scale must handle millions of packets per second, data rates of IoT sensor networks can peak at 100–1000 kbit/s according to industrial partners we’ve

consulted with, while otherwise remaining mostly dormant. However, time-critical functionality like cybersecurity functions or network fault analytics still require the minimal response times that local dataplane processing provides. This gives us scope to use SBC hardware as part of an in-situ ‘Goldilocks’ solution, allowing administrators to choose a packet processing device with just the right size, cost, and power draw for a network’s expected ingress/egress rates—without needing to steer traffic to an external cloud provider for scrubbing. This keeps latency—and thus the reaction time to adverse security events—as low as possible.

Where existing works look at SFC through the lens of data-centre networking, our research question is how best we can enable traffic processing on SBC devices by taking advantage of modern kernel features to implement the most performant, lowest latency dataplanes possible. We present GALETTE—a device-portable SFC framework and runtime based on the modern XDP [10] and *AF_XDP* [5] network stack bypass. GALETTE (Figure 1) explores the SFC feasibility of SBCs at various price-performance points (e.g., Raspberry Pis and Intel NUCs), and provides a consistent platform for all Linux-capable SBCs. We use a two-tier strategy, splitting *network functions* (NFs) between XDP and userland. NFs are written in *Rust* [28] against a simple API, and are compiled on a remote control node into both native and eBPF binaries. However, (*AF_XDP*’s true value for SBC devices is that its performance scales gracefully with driver support, which buys improvements like zero-copy and elimination of SKB allocation. Our framework requires only a modern Linux version, and benefits from the strong sandboxing and validation of eBPF (§II). While primarily designed for retrofitting cybersecurity, our work can also be used for arbitrary NFs such as secure tunnelling of traffic [30], local reduction of sensor and telemetry data for external processing [26], or compressed ML/RL models [24, 27].

A. Contributions

This paper contributes:

- an analysis of the differences between SBC and server hardware which prevent us from taking full advantage of XDP (§III),
- control- and data-plane designs built on (*AF_XDP*) to allow efficient SFC installation, reconfiguration, and scheduling on SBCs in spite of these differences (§IV),
- a simple API and chain format for building SFCs to target both eBPF and userland environments (§IV-A–IV-B),
- a quantitative evaluation of the forwarding performance and resource use of GALETTE against other usable software dataplanes in SBCs, as well as how our two-tier design allows effective loadbalancing and scheduling of expensive NFs (§V–VI).

II. BACKGROUND

eBPF: The *extended Berkeley Packet Filter* (eBPF) is a simple RISC VM bytecode language used to execute arbitrary user-written code in Linux kernel contexts at runtime [8]. eBPF

programs can be attached to various *hooks* or *tracepoints* in the kernel for precise instrumentation or additional control over, e.g., network stack behaviour. Each valid kernel hook exposes a different set of BPF helpers—a limited set of privileged operations allowed by the kernel in a given function, such as buffer manipulation or adjusting the header pointer for a packet SKB. Programs are restricted similarly to constrained embedded targets: dynamic memory allocation is impossible, and there is no support for floating point arithmetic. Programs may hold state between invocations or share it with userland code via *BPF Maps*, which abstract over arrays, hash maps, and more specialised structures like longest-prefix match tries.

The Linux kernel does some heavy lifting to ensure this functionality is safe and performant. The main mechanism for doing so is its *eBPF Verifier*, which ensures that all pointer and map accesses are length-checked and that the program is guaranteed to terminate in a bounded time. Programs themselves are each limited to 4096 instructions, 512 B of stack space, and may be linked via tail-calls up to a maximum 32 programs. Additionally, all used BPF helpers must be permitted by the call site, and only pointers supplied by the kernel (or maps) have the type and length information needed to be dereferenced. Kernel versions 5.3 onwards allow for the use of trivially bounded loops without unrolling. Following verification, BPF programs are JIT-compiled for the host architecture by translating BPF instructions to their native machine code equivalents—program code is then further hardened by write protecting memory and constant blinding to prevent the insertion of exploitable native code gadgets [12].

XDP: The *eXpress Data Path* (XDP) is a network-stack bypass mechanism for Linux, allowing eBPF programs to handle network packets at the earliest possible stage in the kernel [10]. XDP hooks may modify packet contents and adjust header lengths (enabling en/decapsulation), and may quickly redirect packets to be transmitted on their arrival port or any other NIC. Packets may also be redirected to other registered XDP hooks via eBPF tailcalls, or to the standard network stack. The stack bypass is completed by *AF_XDP* [5], which allows packets to be passed from XDP hooks to userland, allowing packets which need more complex processing to entirely skip the Linux network stack. This follows a netmap-like design [22], passing UMEM frame descriptors between contexts using a set of ring buffers. To take advantage of multicore systems and scale to high packet rates, XDP runs one eBPF context in parallel for each hardware receive queue exposed by a NIC.

XDP is particularly valuable compared to other stack or kernel bypass approaches like *DPDK* [6] because it does not require dedicated driver support. All NICs are supported by default by placing the eBPF program after SKB allocation; performance benefits scale gracefully with driver support, enabling the hook to run *before* SKB creation or in a zero-copy mode. Capable hardware like SmartNICs [18] or *hXDP* [3] may even offload eBPF code to the NIC for true kernel bypass.

III. MOTIVATION

While (AF_)XDP SFC frameworks are well-studied (§VII), SBCs—particularly those in edge or IoT networks—are differentiated from datacentre or cloud environments in key ways that existing frameworks do not account for:

- *Raw compute performance and resources.* SBCs such as Raspberry Pis have slower CPU clocks, fewer cores, and less RAM than server-grade hardware, making it harder to run more costly NFs within packet deadlines. However, this further amplifies the impact of host networking stack overheads on packet processing latency and throughput. This ties into two useful counterpoints raised in §I: IoT and sensor networks exhibit low (sub-Mbit/s) data rates, so low-power processing nodes are an ideal fit; equally, the lower up-front cost of these platforms makes them more attractive for installation in vulnerable and remote environments. Yet we must still take advantage of advances in modern network stacks to minimise power use and reduce latency—meaning quicker response to network events, particularly in security-focussed SFCs.
- *Rx/Tx queues and use of NIC virtual functions.* XDP’s degree of parallelism scales according to the number of Rx queues exposed by a NIC. Datacentre NICs targeting 10GbE and above allow Rx queues and virtual functions to be created on demand, thus XDP-based frameworks can easily scale to have as many pipelines as are required to meet bandwidth needs. On the contrary, we find that GbE and below NICs common to SBCs—including those with DPDK support, such as the Intel I219-V—expose only a single Rx/Tx queue. Though many SBCs are multicore devices, this mismatch prevents the XDP datapath from scaling to fit these extra cores. As a result, SFCs on this single pipeline are vulnerable to tail latency spikes and throughput losses caused by *more expensive NFs which may only be needed by a subset of traffic*.

A concrete use case is in how an SBC-based SFC framework could be used to secure legacy IoT or sensor networks at minimal cost, as in §I. Assuming exploit-based attacks rather than volumetric DDoS traffic (which would have to be effectively dealt with upstream), the majority of network packets traversing an IoT gateway are likely to be normal, and should not require complex processing like *deep packet inspection* (DPI)—which would block an SBC’s sole XDP thread. Most importantly, we need a framework with the agility to install, update, and reconfigure chains of NFs to defend against attacks as they evolve.

IV. DESIGN

We present here our design and implementation for GALETTE, which takes advantage of advances in Linux host networking—XDP—to provide a portable, device independent framework for deploying SFCs onto modest-spec heterogeneous devices irrespective of machine architecture.

We use a hybrid XDP and userland solution to bring low-latency SFCs to inexpensive SBC hardware, enabling the ret-

```
# -- NF & Map definitions --
[functions.access-control.maps]
allow-list = { type = "lpm-trie", size = 65535 }

[functions.weak-classifier]
maps = { flow-state = "_" }

[functions.dpi]
maps = { flow-state = "_" }
disable_xdp = true

[maps.flow-state]
type = "hash_map"
size = 65535

# -- Chain definition --
[[links]]
from = "rx"
to = ["access-control"]

[[links]]
from = "access-control"
to = ["tx", "weak-classifier"]

[[links]]
from = "weak-classifier"
to = ["tx", "!dpi", "drop"]

[[links]]
from = "dpi"
to = ["tx", "drop"]
```

Listing 1. A security-focussed function chain. Cheaper classification is kept in the XDP datapath, while expensive analyses are pushed into userland.

rofitting and modification or defence of IoT & sensor networks (fig. 1). Chains are composed of individual crates written in Rust—each compiled to *both eBPF and native machine code* by a remote control plane machine—against a simple API which unifies packet accesses in both environments. This enables easy dual compilation into both eBPF and native machine code while guaranteeing memory safety, but crucially allows one chain to compile to SBCs with different CPU architectures. Userland-only (*non GALETTE-native*) NFs may link to external C libraries. As modern SBCs are multicore but their NICs typically support only a single XDP thread, we push expensive or complex NFs (possibly required by only a subset of traffic) out of the main datapath to exploit extra cores for load balancing while eliminating packet stalls and minimising extra latency.

A. Chain format

SFCs are defined by administrators via a user-friendly TOML format, which affords control over BPF map definitions as well as where individual NFs are run. Listing 1 shows a practical example of a security-focussed chain: packets are processed by an ACL backed by an LPM trie map, while suspicious packets are also checked by a cheaper statistical classifier which may trigger an expensive DPI operation in userland. Any map may be shared between NFs by pointing to a `[maps.<x>]` block.

Every chain starts from `"rx"`, and routes from there to its NFs and the special destinations `"tx"`, `"pass"`, `"abort"`, and `"drop"`. These correspond to most of the standard XDP actions—XDP_PASS forwards a packet to the standard host networking stack, and is useful if control plane traffic must be sent in-band (i.e., a bump-in-the-wire installation at a site with

```

#![no_std]
pub use nf::*;

#[maps]
pub struct Maps { count: (u32, u64) }

pub enum Action { Continue }

pub fn packet<M1>(<mut pkt: impl Packet,
mut maps: Maps<M1>
) -> Action where M1: Map<u32, u64>,
{
    if let Some(bytes) = pkt.slice(12) {
        // bytes: &mut [u8]
        let (src_mac, rest) = bytes.split_at_mut(6);
        src_mac.swap_with_slice(&mut rest[..]);

        if let Some(n) = maps.count.get(60) {
            maps.count.put(60, 8(n + 1));
        }
    }

    Action::Continue
}

```

Listing 2. A counting macswap function in GALETTE. GALETTE-native NFs are written in Rust with a simple API shared between eBPF and native targets.

poor cellular service). The number of valid *to* destinations is determined and checked during code generation (§IV-C).

Listing 1 also demonstrates two ways that users can manage the placement of NFs—they may explicitly run any NF in userland with the “!dest” syntax, or set *disable_xdp* = *true* to do this implicitly and skip eBPF compilation (§IV-C). Control over these mechanisms is explicit to allow users to define pure XDP chains and avoid cross-compilation, or to effortlessly load-balance a single chain between userland and XDP. The exception is that calls are forcibly moved to userland if the chain length exceeds 32 NFs.

B. Shared NF API

Compiling to eBPF requires that we mark an NF’s crate as *#![no_std]*, yet there are standard functions that NF authors desire use of like random number generation, timestamping, map declaration/access, and packet headroom adjustments. Also, though both NF environments provide packet access via a byte buffer, the eBPF verifier requires that packet accesses are length checked in a particular way to pass static verification (i.e., such that Rust’s pointer-length slices are inadequate). To this end, we provide a small standard library and a family of traits which enable programmers to write GALETTE-native NFs.

Listing 2 shows how a simple MAC swapping NF is implemented in GALETTE. The *Packet* trait abstracts over different packet access semantics in userland and XDP, implemented for UMEM frames and XDP contexts, respectively. Handling of maps requires a custom procedural macro, *#[maps]*, which converts (K, V) pairs into generic *Map<K, V>*s and exposes these types such that they can be programmatically manipulated when building concrete NF skeletons for each target.

Userland-only NFs do not require the *#![no_std]* tag, and are thus free to call standard library functions, as well as link to and call precompiled libraries and native code. This frees users

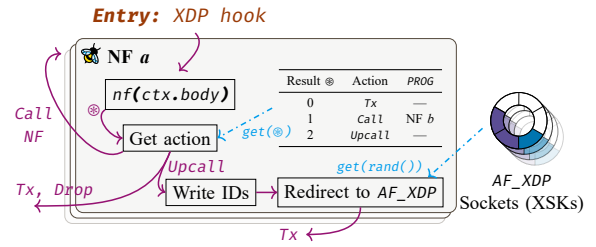


Figure 2. Packet processing in the XDP Fast Path (NF maps omitted).

from having to reimplement complex well-tested NFs, but also allows administrators to limit and reason about the unsafety of (possibly proprietary) C codebases. These currently use eBPF maps, and so key/value types must be plain-old-data structs. In future these NFs could instead be given direct access to efficient userland shared data structures.

C. Compilation

SBCs lack the resources to (re-)compile NFs. Compilation of NFs in an SFC is thus performed by an external controller machine which is trusted from the outset (§IV-E), where eBPF binaries (“elf”) and dylibs (“so”) are incrementally sent over the network. The root *lib.rs* file for each NF is parsed into an AST using the ‘syn’ library, from which we extract:

- the type name of the first *struct* decorated with *#[maps]* plus its number of fields, and
- the return type of the main *packet()* function as well as the number of *enum* variants it has.

We compare this against the chain definition file to ensure that there is a one-to-one correspondence between these and the expected number of maps and output branches. From this, we then generate source code and crates which compile to an eBPF program and dylib. Userland NFs require little additional code. Generation of XDP NFs is slightly more involved: we insert a declaration for each map required by user code (with key and value types exposed and inserted via the NF API—§IV-B), allocate maps for *AF_XDP* sockets, and storage for NF chain actions and next hops via the *PROG_ARRAY* map type.

D. Dataplane operation

XDP (fig. 2): On arrival, the main body of the NF is executed and its output is used as an index to the map of actions defined by the chain—e.g., *XDP_TX/DROP*, NF calls. Although eBPF maps have a small runtime lookup cost, we make use of them to store action mappings to enable dynamic, fast reconfiguration without needing to recompile any NFs.

When calling another NF, we perform an XDP tail-call into a *PROG_ARRAY* using the same index. When upcalling to userland via *AF_XDP*, packets arrive without explicitly including XDP state but do include user-defined metadata. As *AF_XDP* sockets rely on *single-producer-single-consumer* (SPSC) rings to move frames between userland and the kernel, we create one socket per running userland thread and load balance between them using the *bpf_get_prandom_u32()* helper function. However, as any chain allows many discrete XDP→Userland transitions, we need to signal the identity of

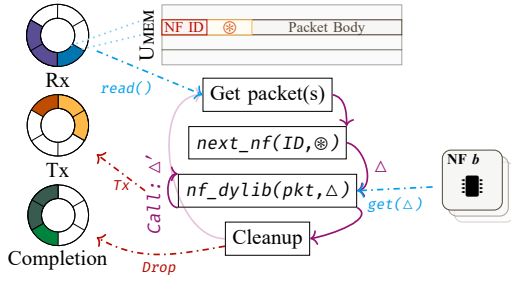


Figure 3. Packet processing in userland (single-thread, NF maps omitted).

the callee NF so that packet processing can resume. To work around this we assign a unique ID to each live NF. When an upcall is required, we expand a packet’s metadata by 8 B to store this ID and the action index—inspecting the kernel source code, this resolves to a cheap pointer adjustment within the larger headroom block rather than a *memcpy*.

Userland (fig. 3): Packets are received on the Rx ring of each userland thread’s *AF_XDP* socket, and XDP metadata is read from the UMEM frame to establish the callee and target NFs. This allows us to look up the function pointer given by the required NF’s dynamic library, and packets are then processed by run-to-completion. We choose this model as there is no native mechanism to pass packets back to XDP—subsequent GALETTE-native NFs are also executed as dylibs (which can be better-compiled than the in-kernel JIT). Some other XDP-based approaches like *Polycube* [13] choose to downcall using a TAP device after every ‘slow-path’ NF executes. We choose not to pay the extra repeated overhead needed to do so, while also keeping packet pressure off the sole XDP fast-path thread.

As SBC hardware allows us to bind only a single Rx queue for XDP, we are forced to use a single UMEM pool between all *AF_XDP* sockets. In turn, UMEM frame cleanup and recycling can only occur on a single thread. We delegate these tasks to the first userland thread, using ring buffers between it and threads 2–*n* to correctly handle packet drop/abort actions.

E. Control plane

GALETTE-enabled SBCs are setup with pre-shared TLS keys and certificates to enable mutual authentication with a known control plane server. The SBC device periodically contacts this server to retrieve chain information, NF binaries, and to synchronise eBPF map contents. In our current prototype, devices also specify their kernel version and provide *vmLinux* debug information—the underlying *redbpf* toolchain does not yet support eBPF CO-RE [17]. The SBC’s control plane component then loads eBPF programs and *dlopen*s dylibs, installs shared maps and chain actions plus *PROG_ARRAY* entries for tail calls, and finally links the initial NF to the XDP hook.

Reconfigurability: Chains are received by the SBC as a graph of links between 128bit NF UUIDs. If the retrieved chain differs from the one installed locally, the SBC requests any NFs whose UUIDs are needed¹. eBPF’s *PROG_ARRAY* maps

¹We do not conditionally download the XDP & userland variants of an NF. This reduces the latency of a chain reconfiguration (i.e., in response to unexpected load), at the cost of an increased initial transfer time.

contain only pointers to other eBPF programs, and element-wise are guaranteed to update atomically. However, chain-wide updates require more care as multiple actions associated with an NF may be updated, or we may need to apply several updates concurrently². This can be performed analogously to Xing *et al.* [33]’s ‘program consistency’; we may recursively build a replacement chain as a tree from changed NFs or links, before atomically replacing the tail-call into the tree’s root. Duplicated but unchanged NFs refer to the same eBPF maps as their live counterparts. In the worst case this is equivalent to rebuilding the entire chain, but it should be noted that this is feasible here without sacrificing consistency guarantees because, while computationally limited, SBCs are less constrained than, e.g., P4 switches in usable memory and live program space. Userland replacement of individual NFs is simplified by the use of function trampolines.

F. Limitations

XDP hooks allow for a maximum 32 tail calls—currently we force an upcall at this threshold. While we could joint-compile NFs to keep longer chains in the ‘fast path’, this requires that we also explicitly deoptimise programs in response to chain changes. While this may also add some performance benefit, the cost of each tailcall is negligible [13]. The source code requirement for fast path NFs appears, at first, to be overly restrictive. A future extension of GALETTE can apply the same technique as *SafeBricks* [21] to allow for proprietary code to be safely compiled in a TEE hosted on the remote compile server. Other facets of IoT deployments complicate the control plane operation of GALETTE—for instance, unpredictable downtime and the lack of ECC memory threaten long-term integrity of cryptographic keys. We are developing mechanisms based on *physical uncloneable functions* [9] to reauthenticate new ephemeral keys for the control plane.

V. EVALUATION

We perform a testbed evaluation of GALETTE to understand how it performs in raw throughput and latency against existing stacks, as well as how the two-tier execution framework allows us to handle more expensive NFs in a scalable way. We describe here the experimental design used to investigate these system properties. As SBCs include machines at many different price-performance points, we aim to show these characteristics on both Raspberry Pis (~£30, 1.4–3.7 W) and i7-equipped Intel NUCs (~£500, TDP 28 W).

A. Testbed setup

Our testbed comprises the below machines:

- Compile** AMD Ryzen 9 5900X (12 × 3.7 GHz), 32 GiB RAM, WSLArch 5.15.79 via WSL2.
- TrafGen**, NUC Intel NUC8i7BEK (4 × 4.5 GHz), 16 GiB RAM, Ubuntu Server 22.04 5.15.0-56-lowlatency.
- RPi** Raspberry Pi Model 3B (4 × 1.2 GHz), 1 GiB RAM, Raspberry Pi OS 11 (AArch64) 5.15.74-v8+.

²While *BPF_MAP_TYPE_ARRAY_OF_MAPS* would be ideal for simplifying this logic, these cannot currently store *PROG_ARRAYS*.

All devices use a shared WiFi network for control plane traffic, with *Compile* used to build NFs for all target platforms, serve as the controller for GALETTE, and orchestrate all experiments. Our dataplane testbed consists of *TrafGen*, *NUC*, and *RPi*, which are connected over Ethernet via a TP-Link TL-SG108S GbE switch, although *RPi*'s built-in NIC supports only 100BASE-T. We note that the *RPi* NIC is a USB 2.0 peripheral (SMSC LAN9514), while the *NUC*'s NIC connects over PCIe (Intel I219-V). As a result, *RPi* packet arrivals are affected by the LAN9514 Ethernet controller's minimum USB polling interval of 1 ms [15, p. 32]. All dataplane devices were set up with DPDK 22.07, and Active State Power Management was disabled where possible for more reliable Ethernet performance. *RPi*'s Linux kernel was recompiled from the official Raspberry Pi OS source to include XDP support, the eBPF JIT, and BTF debug information (which are disabled by default). IRQ throttling was disabled on both *NUC* and *TrafGen* to minimise forwarding latency.

Throughput and latency are measured by *TrafGen* using Pktgen-DPDK [32], generating and receiving traffic at the target speed and packet size for bursts of 11 s. We assess system load via CPU and, where possible, power measurements. CPU measures were recorded by reading `/proc/stat` every 0.5 s on the device under test. Power measurements for *RPi* were recorded using an RDTech UM24C USB load monitor, polled every 0.25 s over Bluetooth from *TrafGen*. We run each experiment over 10 trials.

We have implemented our SFC compiler and core dataplane functionality in Rust v1.65.0, while using v1.59.0 to compile eBPF binaries due to LLVM version limits of the redbpf library. Our code and data are publicly available as open source artefacts [25]. The Arch Linux GCC 9.3.0 toolchain was used to cross-compile for AArch64 due to libc version dependencies in our target machines.

B. Experiments

Optimal dataplane performance: We install a simple Macswap NF via GALETTE to measure its optimal forwarding throughput and latencies on a variety of rates (0.1, 1, 10, 50 and 100 Mbit/s) and packet sizes (64, 128, 256, 512, 1024, 1280 and 1518 B). We are interested in how both our XDP fast path and the slower userland *AF_XDP* datapath behave in these metrics and resource utilisation—particularly under the high packet-per-second requirements imposed by smaller packets. The single userland thread is pinned to core 1. Uncertainties in tables represent 95% CIs, while latency boxplots show {1, 25, 50, 75, 99}th percentiles. We compare GALETTE against baselines provided by DPDK's *testpmd* application to establish maximum throughput and minimum forwarding latency, using a Macswap NF. On *RPi* this lets us compare against *AF_PACKET* for packet access under optimal conditions, while on *NUC* we may also compare against the performance of the *e1000e poll-mode driver* (PMD).

As *AF_XDP* supports polling on sockets from userland, we investigate the effects of reading from the userland socket via polling (*Poll/P*) or blocking I/O (*IRQ*). We set a minimum-

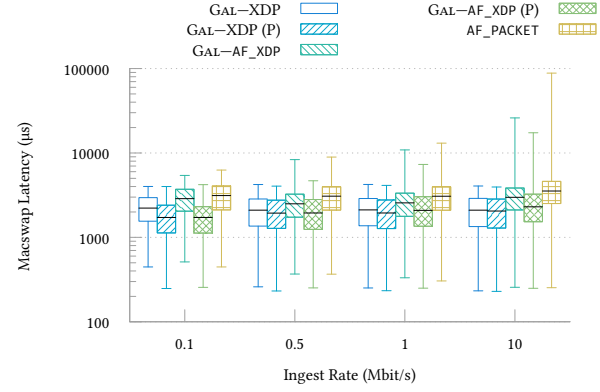


Figure 4. Macswap latency distributions for 64 B packets on *RPi* for GALETTE vs. *AF_PACKET*. The XDP fast path outperforms *AF_PACKET*, has better tail latencies at higher rates, and improves by polling at low data rates.

Table I
RASPBERRY PI MACSWAP NF THROUGHPUT AND RESOURCE USE, 64 B PACKETS.

Dataplane	Ingest Rate (Mbit/s)	Throughput (Mbit/s)		CPU (%)		Power (mW)	
		IRQ	Poll	IRQ	Poll	IRQ	Poll
Pure XDP	0.1	0.1	0.1	1.8(24)	25.6(13)	1675.7(23)	2230.3(104)
	1	0.9	0.9	3.0(39)	26.6(16)	1664.3(30)	2097.0(104)
	10	9.4	9.4	20.0(61)	41.3(53)	1802.8(75)	2186.3(85)
	50	2.9(3)	3.9(22)	40.0(42)	52.8(221)	1826.5(85)	2227.1(100)
	100	0.1	0.1	2.1(25)	25.4(10)	1695.7(29)	2321.3(81)
<i>AF_XDP</i>	0.1	1.0	1.0	3.9(26)	26.4(17)	1702.5(29)	2197.2(103)
	1	9.5	9.5	35.6(57)	38.6(96)	1852.2(94)	2298.8(106)
	10	3.7(2)	5.2(16)	46.9(46)	62.1(47)	1844.3(94)	2286.3(123)
	50	—	—	—	—	—	—
	100	—	—	—	—	—	—
<i>AF_PACKET</i>	0.1	—	0.1	—	64.4(342)	—	2107.5(45)
	1	—	1.0	—	54.7(75)	—	2068.2(36)
	10	—	9.5	—	75.9(86)	—	2198.6(77)
	50	—	3.4(6)	—	84.4(81)	—	2192.5(70)
	100	—	—	—	—	—	—

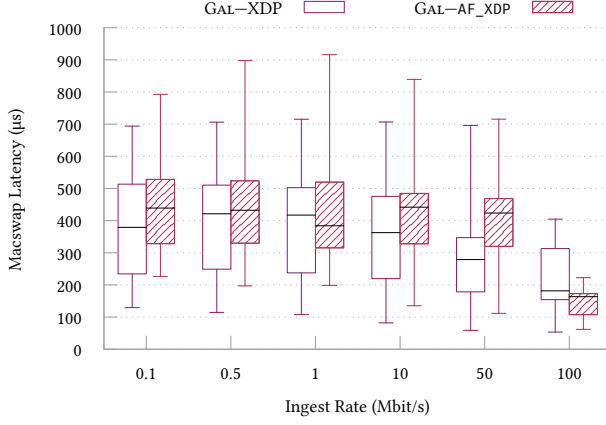
length 1 ms timeout on blocking reads. This offers a useful point of comparison against our baselines which operate by polling, in addition to another way to impact the performance of both our XDP and userspace datapaths.

Scheduling expensive NFs: Returning to the earlier security-focussed case study, SBC hardware limits us to a single XDP thread—a single pipeline is thus vulnerable to the impact of packets which require more costly processing. We investigate how a single expensive NF of 100k operations—analogueous to expensive ML inference or DPI—affects throughput and median/tail latency, and whether moving this traffic to userland can alleviate these issues. We vary the probability that a packet requires an expensive NF, $P \in \{0, 0.01, 0.05, 0.1, 0.25, 0.5, 1.0\}$. This is measured for 64 B packets, giving the highest ingest PPS (thus strictest packet deadlines) at a rate that each device can meet on XDP for this packet size with some slack time (10 Mbit/s for *RPi*, 100 Mbit/s for *NUC*). We then narrow our focus to a challenging scenario for each device—fixing $P = 0.5$ —to investigate how extra processing threads in userland can be used to balance the load imposed by higher proportions of expensive packets.

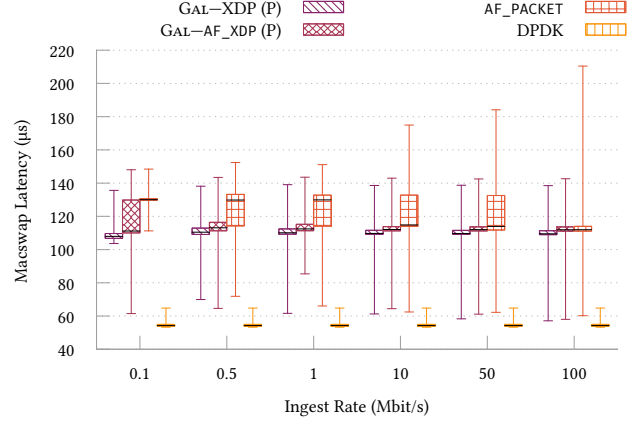
VI. RESULTS AND DISCUSSION

A. Optimal dataplane performance

Figure 4 shows the distribution of packet latencies on *RPi*, up to its peak stable throughput for 64 B packets. We see that GALETTE's XDP fast path significantly outperforms *AF_PACKET*—29–41% lower median latencies for 0.1 Mbit/s



(a) IRQ



(b) Poll

Figure 5. Macswap latency distributions for 64 B packets on an Intel NUC for GALETTE vs. *AF_PACKET* and DPDK. XDP consistently outperforms *AF_XDP*, with both seeing improvements at higher traffic rates. Polling improves the performance of GALETTE beyond *AF_PACKET*, but cannot beat DPDK.

Table II

RASPBERRY PI MACSWAP NF THROUGHPUT AND RESOURCE USE, 1518 B PACKETS.

Dataplane	Ingest Rate (Mbit/s)	Throughput (Mbit/s)		CPU (%)		Power (mW)	
		IRQ	Poll	IRQ	Poll	IRQ	Poll
Pure XDP	0.1	0.1(1)	0.1	1.8(23)	25.5(15)	1678.6(28)	2259.8(113)
	1	1.0(1)	1.0(1)	1.7(24)	25.5(19)	1684.4(22)	2224.5(106)
	10	9.9(1)	9.9	2.1(38)	26.3(15)	1709.6(31)	2181.6(105)
	50	49.1(1)	49.1	5.3(44)	30.2(46)	1745.1(43)	2185.7(105)
	100	97.6(1)	97.6(1)	10.4(66)	36.9(67)	1824.5(84)	2253.6(98)
<i>AF_XDP</i>	0.1	0.1(1)	0.1(1)	1.8(22)	25.4(13)	1697.2(21)	2330.7(92)
	1	1.1	1.0(1)	2.1(34)	25.7(25)	1686.2(24)	2254.4(100)
	10	9.9	9.9	3.6(28)	26.0(22)	1714.5(30)	2285.5(106)
	50	49.1(1)	49.1(1)	14.0(45)	29.6(37)	1761.3(51)	2258.7(108)
	100	97.6(1)	97.6	32.4(78)	36.5(50)	1885.1(104)	2227.9(100)
<i>AF_PACKET</i>	0.1	—	0.1(1)	—	53.6(63)	—	2069.1(37)
	1	—	1.1	—	54.1(72)	—	2072.4(44)
	10	—	9.9	—	56.9(125)	—	2069.3(37)
	50	—	49.1	—	63.1(53)	—	2152.8(54)
	100	—	97.7(1)	—	74.3(75)	—	2253.9(88)

and 10 Mbit/s (36.1–95.4% reduction at 99th percentile) without polling. At moderate data rates, the userland datapath (*AF_XDP*) falls between these two extremes, tending towards *AF_PACKET*'s median behaviour under stress (with better tail behaviour). Higher ingest rates lead to lower minimum latencies (i.e., a few packets benefit from batching), but have limited impact on median XDP latencies (5.5%) and a slight adverse impact on *AF_XDP*—performance is mainly governed by the USB polling interval. Overloading ingest (e.g., ≥ 50 Mbit/s) causes median latencies to increase by $100 \times$ (plot omitted).

Figure 5a shows that, on Intel NUCs, the XDP fast path offers a 100–150 μ s median improvement over the userland datapath, where latencies improve at higher ingest rates in both cases. However, at 1 Gbit/s XDP's median–99th latencies increase to 1978.2–2008.7 μ s regardless of polling, with *AF_XDP* 1.44 \times worse (plot omitted). When polling for packets (fig. 5a), both of GALETTE's datapaths outperform *AF_PACKET*, but exhibit around 2 \times the overhead of DPDK.

We see that GALETTE's dataplanes can sustain traffic on *RPi* at 10, 50 and 100 Mbit/s for 64, 512 and 1518 B packets respectively (tables I and II). *AF_PACKET* fails to meet 50 Mbit/s for 512 B packets (table omitted). *NUC* is able to meet 1 Gbit/s traffic for all packet sizes and dataplane designs except *AF_PACKET* (table III). We see sub-linear results here as we include the startup and winddown of traffic generation—manual testing with Pktgen-DPDK confirms that

Table III

INTEL NUC MACSWAP NF THROUGHPUT AND RESOURCE USE, 64 B PACKETS.

Dataplane	Ingest Rate (Mbit/s)	Throughput (Mbit/s)		CPU (%)	
		IRQ	Poll	IRQ	Poll
Pure XDP	0.1	0.1	0.1	0.5(4)	12.5(3)
	1	1.0	1.0	0.5(4)	12.5(2)
	10	9.6	9.6	0.6(4)	12.5(3)
	50	47.8	48.1	0.5(4)	12.6(3)
	100	95.9	96.2	0.4(5)	12.7(3)
<i>AF_XDP</i>	1000	744.8(1)	745.0(1)	10.5(5)	23.0(4)
	0.1	0.1	0.1	0.2(4)	12.5(2)
	1	1.0	1.0	0.5(4)	12.5(2)
	10	9.5	9.6	1.5(4)	12.5(2)
	50	47.8	48.1	5.1(9)	12.6(3)
<i>AF_PACKET</i>	100	95.9	96.2	1.2(5)	12.8(3)
	1000	744.8(1)	744.5(22)	21.6(7)	22.7(9)
DPDK	0.1	—	0.1	—	12.5(1)
	1	—	1.0	—	12.5(1)
	10	—	9.6	—	12.8(4)
	50	—	47.8	—	14.0(3)
	100	—	95.6(1)	—	15.5(4)
DPDK	1000	—	433.7(92)	—	25.0(2)
	0.1	—	0.1	—	12.5(1)
	1	—	1.0	—	12.5(1)
	10	—	9.6	—	12.5(1)
	50	—	48.1	—	12.5(1)
DPDK	100	—	96.2	—	12.5(1)
	1000	—	745.1	—	12.5(1)

our dataplanes and DPDK sustain 1 Gbit/s at peak.

On *RPi* (IRQ), we see lower like-for-like CPU and power use as compared with *AF_PACKET* (table I). For 1518 B packets in the same case, we see that the pure XDP datapath offers up to $3.1 \times$ CPU reduction versus userland, requiring 7.4% less power (table II). We interpret this effect as arising from copying packet contents into UMEM frames, however this remains lower than *AF_PACKET*. However, while GALETTE CPU use is lower than *AF_PACKET* when polling, we observe that XDP dataplanes have around 5% higher power use. In *NUC*, we see that the fastpath is always more CPU-efficient for rates beyond 1 Mbit/s (table III).

Takeaways: GALETTE is well-suited to non-DPDK capable devices such as Raspberry Pis, offering better overall performance in both its datapaths (and lower power/CPU use) than frameworks like *AF_PACKET*. Polling offers limited benefits on the Raspberry Pi, and cannot beat best-of-breed solutions

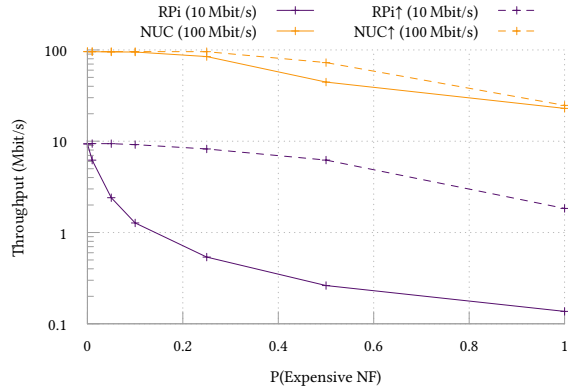


Figure 6. Throughput degradation as compute-intensive NFs are run in the XDP path. ‘↑’ denotes passing such packets to userland. Expensive NFs cause significant throughput loss—strongest for *RPi*—but are alleviated using GALETTE’s two-tier datapath strategy.

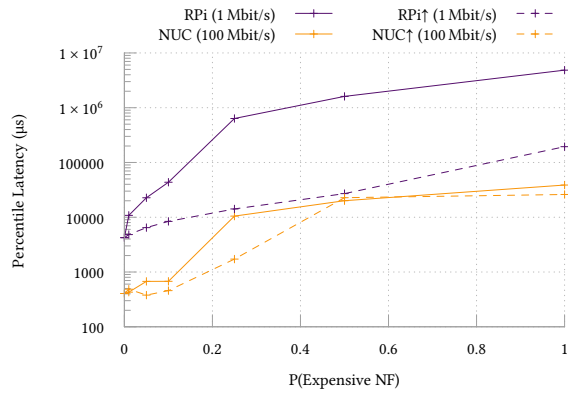


Figure 7. 99th percentile latency degradation, as in fig. 6. For *RPi*, offloading costly NFs to userland offers 1–2 orders of magnitude improvement.

like DPDK when they are supported—however, it is already understood that XDP underperforms in this comparison, but is far more CPU-efficient at lower traffic rates [10, 13] (i.e., those of interest in IoT/sensor networks). GALETTE benefits most from the CPU and power reductions of blocking I/O to support portable, *efficient*, in-situ traffic processing.

B. Scheduling expensive NFs

Compute-intensive NFs, as hypothesised, significantly harm median/99th percentile latencies and overall throughputs if applied in the XDP fast path to even 1 % of packets. Figures 6 and 7 show this property for throughput and 99th percentile latencies respectively on peak traffic for each SBC—*RPi* is more adversely affected (having a slower CPU clock), but unlike *NUC* even lower traffic rates are impacted (0.5 and 1 Mbit/s, plot omitted). Crucially, pushing costly NFs to userland alleviates this problem, particularly on weaker SBCs (i.e., a $25.7 \times$ increase in throughput for *RPi*, $P = 0.5$).

We find the value of additional cores for load balancing depends on NF complexity. In earlier experiments, we found for *RPi* that the split-datapath design could not improve throughputs beyond pure XDP for cheaper NF chains (i.e., bottlenecked by kernel packet handling), while on *NUC* the XDP dataplane could already handle all rates with ease in

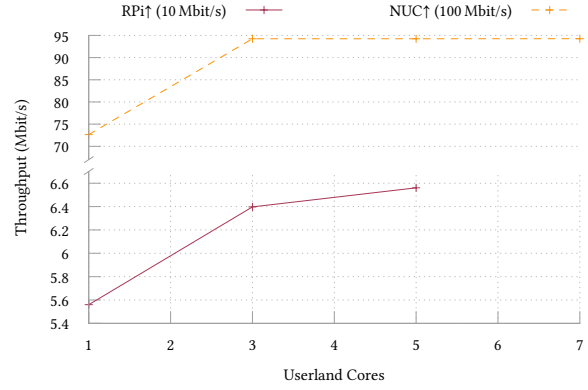


Figure 8. Extra userland cores lead to improved throughput as more packets ($P = 0.5$) require expensive NFs. 64B packets.

spite of having a single queue. In addition, we found there to be some contention when reading from the separate *AF_XDP* sockets in userland which limits the applicability of load-balancing cheaper NFs in this way. However, more expensive NFs are a prime candidate for placement over the unused device cores—we see from fig. 8 that additional threads are serving as many packets as possible.

Takeaways: The two-tier split datapath design of GALETTE is key in ensuring that SBCs can provide different quality of service to packets which require different levels of processing, and for protecting ‘normal’ traffic from latency spikes and drops caused by expensive analyses on adversarial flows.

VII. RELATED WORK

SBC dataplanes: *P4Pi* [11] is an educational platform to run P4 dataplanes on Raspberry Pi devices. Its dataplane uses DPDK on tap devices bridged from the onboard NICs (similar to our *AF_PACKET* baseline), and is limited to the simpler semantics of P4 programs versus our eBPF-plus-native code strategy. P4Pi and GALETTE have altogether different aims and are complementary to one another; given our results, P4Pi might reduce its power costs and latency using (*AF_XDP*).

eBPF/XDP dataplanes: *Polycube* [13] uses a similar hybrid XDP-userland model to provide SFC in datacentres. While both GALETTE and Polycube rely on chains of tail-calls between XDP programs, Miano *et al.* designate an explicit userland component per NF. If userland processing is required, Polycube upcalls packets using per-CPU ring buffers, and then *recirculates packets back to the XDP datapath* using tap devices. GALETTE is instead tailored towards SBC devices. We use *AF_XDP* for upcalling due to its ubiquity and robustness—and never return packets to XDP due to the single XDP thread offered by SBC hardware. Both frameworks are specialised toward their target environment (datacentres vs. SBCs).

Morpheus [14] examines how runtime profiling can improve the performance of eBPF-based dataplanes via *profile guided optimisation*. While the improvements it offers would complement our work, this relies on routinely sending instrumentation data back to GALETTE’s compile server—and so would require more in-depth cost/benefit analysis of network overheads.

Shahinfar *et al.* [23] have seen some performance benefits

in splitting packet processing between XDP and userland via *AF_XDP*. We believe that this is due to the same ‘pipeline parallelism’ that *GALETTE* takes advantage of.

eBPF in industry: eBPF’s safety, performance, and kernel integration make it useful in many applications. *Cilium* [4] uses XDP to insert security and load balancing functions into container networks. *flowtrackd* [34] uses *AF_XDP* to provide DDoS attack scrubbing capabilities for Cloudflare CDNs. *Open vSwitch* [29] has been redesigned to make use of *AF_XDP* for its agility over kernel modules, while load balancers like *Katran* [7] are in widespread deployment in Meta.

Rust-based dataplanes: *NetBricks* [19] installs SFC graphs of Rust NFs over the DPDK dataplane. It offers effective abstractions for aggregating and processing traffic—however, it does not provide any mechanisms or support for an eBPF/XDP dataplane. *SafeBricks* [21] protects these NFs from a compromised kernel using TEEs, however these are absent in Raspberry Pis and deprecated in consumer Intel CPUs [20].

VIII. CONCLUSION

We have presented *GALETTE*, an SFC framework designed for the cheap defence of IoT networks. By carefully considering the limitations of the XDP framework on SBC hardware, we have designed and implemented an SFC framework tailored to making the best use of SBC parallelism while protecting ‘normal’ traffic. We have empirically shown that our design achieves lower and more consistent latencies on weaker SBC devices like Raspberry Pis, enabling inexpensive NF installation in IoT and sensor networks.

Acknowledgements: This work was supported in part by the PETRAS National Centre of Excellence for IoT Systems Cybersecurity, via the UK Engineering and Physical Sciences Research Council [grant EP/S035362/1].

REFERENCES

- [1] Manos Antonakakis *et al.* ‘Understanding the Mirai Botnet’. In: *26th USENIX Security Symposium, Vancouver, Canada, August 16-18, 2017*. USENIX Association, 2017, pp. 1093–1110.
- [2] Tom Barbette *et al.* ‘Fast Userspace Packet Processing’. In: *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for networking and communications systems, ANCS 2015, Oakland, CA, USA, May 7-8, 2015*. IEEE Computer Society, 2015, pp. 5–16.
- [3] Marco Spaziani Brunella *et al.* ‘hXDP: Efficient Software Packet Processing on FPGA NICs’. In: *14th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2020, Virtual Event, November 4-6, 2020*. USENIX Association, 2020, pp. 973–990.
- [4] Cilium Authors. *Cilium. Linux Native, API-Aware Networking and Security for Containers*. 2022. URL: <https://cilium.io/>.
- [5] Jonathan Corbet. *Accelerating networking with AF_XDP*. 9th Apr. 2018. URL: <https://lwn.net/Articles/750845/>.
- [6] DPDK Project. *DPDK*. 2022. URL: <https://www.dpdk.org/>.
- [7] Facebook Incubator. *Katran. A high performance layer 4 load balancer*. 2020. URL: <https://github.com/facebookincubator/katran>.
- [8] Matt Fleming. *A thorough introduction to eBPF*. 2nd Dec. 2017. URL: <https://lwn.net/Articles/740157/>.
- [9] Yansong Gao *et al.* ‘Physical unclonable functions’. In: *Nature Electronics* 3.2 (Feb. 2020), pp. 81–91. issn: 2520-1131.
- [10] Toke Høiland-Jørgensen *et al.* ‘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, CoNEXT 2018, Heraklion, Greece, December 04-07, 2018*. ACM, 2018, pp. 54–66.
- [11] Sándor Laki *et al.* ‘P4Pi: P4 on Raspberry Pi for networking education’. In: *Comput. Commun. Rev.* 51.3 (2021), pp. 17–21.
- [12] Keegan McAllister. *Attacking hardened Linux systems with kernel JIT spraying*. 17th Nov. 2012. URL: <http://mainisusuallyafunction.blogspot.com/2012/11/attacking-hardened-linux-systems-with.html>.
- [13] Sebastiano Miano *et al.* ‘A Framework for eBPF-Based Network Functions in an Era of Microservices’. In: *IEEE Trans. Netw. Serv. Manag.* 18.1 (2021), pp. 133–151.
- [14] Sebastiano Miano *et al.* ‘Domain specific run time optimization for software data planes’. In: *ASPLOS ’22: 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Lausanne, Switzerland, 28 February 2022 - 4 March 2022*. ACM, 2022, pp. 1148–1164.
- [15] Microchip Technology Inc. *LAN9514/LAN9514i USB 2.0 Hub and 10/100 Ethernet Controller Data Sheet*. 2nd Nov. 2016. URL: <http://www1.microchip.com/downloads/en/devicedoc/00002306a.pdf>.
- [16] Robert Tappan Morris *et al.* ‘The Cluck modular router’. In: *Proceedings of the 17th ACM Symposium on Operating System Principles, SOSOP 1999, Kiawah Island Resort, near Charleston, South Carolina, USA, December 12-15, 1999*. ACM, 1999, pp. 217–231.
- [17] Andrii Nakryiko. *BPF CO-RE reference guide*. 24th Oct. 2021. URL: <https://nakryiko.com/posts/bpf-core-reference-guide/>.
- [18] Netronome. *SmartNIC Overview*. 2021. URL: <https://www.netronome.com/products/smartnic/overview/>.
- [19] Aurojit Panda *et al.* ‘NetBricks: Taking the V out of NFV’. In: *12th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2016, Savannah, GA, USA, November 2-4, 2016*. USENIX Association, 2016, pp. 203–216.
- [20] Jimmy Pezzone. *Intel’s SGX deprecation impacts DRM and Ultra HD Blu-ray support*. 15th Jan. 2022. URL: <https://www.techspot.com/news/93006-intel-sgx-deprecation-impacts-drm-ultra-hd-blu.html>.
- [21] Rishabh Poddar *et al.* ‘SafeBricks: Shielding Network Functions in the Cloud’. In: *15th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2018, Renton, WA, USA, April 9-11, 2018*. USENIX Association, 2018, pp. 201–216.
- [22] Luigi Rizzo. ‘netmap: A Novel Framework for Fast Packet I/O’. In: *2012 USENIX Annual Technical Conference, Boston, MA, USA, June 13-15, 2012*. USENIX Association, 2012, pp. 101–112.
- [23] Farbod Shahinfar *et al.* ‘The case for network functions decomposition’. In: *CoNEXT ’21: The 17th International Conference on emerging Networking EXperiments and Technologies, Virtual Event, Munich, Germany, December 7 - 10, 2021*. ACM, 2021, pp. 475–476.
- [24] Kyle A. Simpson and Dimitrios P. Pazaros. ‘Revisiting the Classics: Online RL in the Programmable Dataplane’. In: *2022 IEEE/IFIP Network Operations and Management Symposium, NOMS 2022, Budapest, Hungary, April 25-29, 2022*. IEEE, 2022, pp. 1–10.
- [25] Kyle A. Simpson *et al.* *Galette*. URL: <https://github.com/FelixMcFelix/galette>.
- [26] Kyle A. Simpson *et al.* ‘Seiðr: Dataplane Assisted Flow Classification Using ML’. In: *IEEE Global Communications Conference, GLOBECOM 2020, Virtual Event, December 7-11, 2020*. IEEE, 2020, pp. 1–6.
- [27] Giuseppe Siracusano *et al.* ‘Re-architecting Traffic Analysis with Neural Network Interface Cards’. In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. USENIX Association, 2022, pp. 513–533.
- [28] The Rust Team. *Rust Programming Language*. 2022. URL: <https://www.rust-lang.org/>.
- [29] William Tu *et al.* ‘Revisiting the Open vSwitch Dataplane Ten Years Later’. In: *ACM SIGCOMM 2021 Conference, Virtual Event, USA, August 23-27, 2021*. ACM, 2021, pp. 245–257.
- [30] Twilio. *Electric Imp Secure IoT Connectivity Platform*. 2022. URL: <https://www.electricimp.com/>.
- [31] UK Parliament. *Product Security and Telecommunications Infrastructure Act 2022*. 16th Dec. 2022. URL: <https://bills.parliament.uk/bills/3069>.
- [32] Keith Wiles. *Pktgen-DPDK – DPDK-based packet generator*. 2022. URL: <https://github.com/pktgen/Pktgen-DPDK>.
- [33] Jiarong Xing *et al.* ‘Runtime Programmable Switches’. In: *19th USENIX Symposium on Networked Systems Design and Implementation, NSDI 2022, Renton, WA, USA, April 4-6, 2022*. USENIX Association, 2022, pp. 651–665.
- [34] Omer Yoachimik. *flowtrackd: DDoS Protection with Unidirectional TCP Flow Tracking*. 14th July 2020. URL: <https://blog.cloudflare.com/announcing-flowtrackd/>.