The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel

Toke Høiland-Jørgensen
Karlstad University
toke@toke.dk

Jesper Dangaard Brouer
Red Hat
brouer@redhat.com

Daniel Borkmann
Cilium.io
daniel@cilium.io

John Fastabend
Cilium.io
john@cilium.io

Tom Herbert
Quantonium Inc.
tom@herbertland.com

David Ahern
Cumulus Networks
dsahern@gmail.com

David Miller
Red Hat
davem@redhat.com

ABSTRACT

Programmable packet processing is increasingly implemented using kernel bypass techniques, where a userspace application takes complete control of the networking hardware to avoid expensive context switches between kernel and userspace. However, as the operating system is bypassed, so are its application isolation and security mechanisms; and well-tested configuration, deployment and management tools cease to function.

To overcome this limitation, we present the design of a novel approach to programmable packet processing, called the eXpress Data Path (XDP). In XDP, the operating system kernel itself provides a safe execution environment for custom packet processing applications, executed in device driver context. XDP is part of the mainline Linux kernel and provides a fully integrated solution working in concert with the kernel's networking stack. Applications are written in higher level languages such as C and compiled into custom byte code which the kernel statically analyses for safety, and translates into native instructions.

We show that XDP achieves single-core packet processing performance as high as 24 million packets per second, and illustrate the flexibility of the programming model through three example use cases: layer-3 routing, inline DDoS protection and layer-4 load balancing.

CCS CONCEPTS

• **Networks** → **Programming interfaces**; **Programmable networks**; • **Software and its engineering** → *Operating systems*;

KEYWORDS

XDP, BPF, Programmable Networking, DPDK

CoNEXT '18, December 4–7, 2018, Heraklion, Greece

ACM ISBN 978-1-4503-6080-7/18/12.
https://doi.org/10.1145/3281411.3281443

ACM Reference Format:
Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. The eXpress Data Path: Fast Programmable Packet Processing in the Operating System Kernel. In *CoNEXT '18: International Conference on emerging Networking EXperiments and Technologies, December 4–7, 2018, Heraklion, Greece.* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3281411.3281443

1 INTRODUCTION

High-performance packet processing in software requires very tight bounds on the time spent processing each packet. Network stacks in general purpose operating systems are typically optimised for flexibility, which means they perform too many operations per packet to be able to keep up with these high packet rates. This has led to the increased popularity of special-purpose toolkits for software packet processing, such as the Data Plane Development Kit (DPDK) [16]. Such toolkits generally bypass the operating system completely, instead passing control of the network hardware directly to the network application and dedicating one, or several, CPU cores exclusively to packet processing.

The kernel bypass approach can significantly improve performance, but has the drawback that it is more difficult to integrate with the existing system, and applications have to re-implement functionality otherwise provided by the operating system network stack, such as routing tables and higher level protocols. In the worst case, this leads to a scenario where packet processing applications operate in a completely separate environment, where familiar tooling and deployment mechanisms supplied by the operating system cannot be used because of the need for direct hardware access. This results in increased system complexity and blurs security boundaries otherwise enforced by the operating system kernel. The latter is in particular problematic as infrastructure moves towards container-based workloads coupled with orchestration systems such as Docker or Kubernetes, where the kernel plays a dominant role in resource abstraction and isolation.

As an alternative to the kernel bypass design, we present a system that adds programmability directly in the operating system networking stack in a cooperative way. This makes it possible to perform high-speed packet processing that integrates seamlessly with existing systems, while selectively leveraging functionality

in the operating system. This framework, called the eXpress Data Path (XDP), works by defining a limited execution environment in the form of a virtual machine running *eBPF* code, an extended version of original BSD Packet Filter (BPF) [37] byte code format. This environment executes custom programs directly in kernel context, before the kernel itself touches the packet data, which enables custom processing (including redirection) at the earliest possible point after a packet is received from the hardware. The kernel ensures the safety of the custom programs by statically verifying them at load time; and programs are dynamically compiled into native machine instructions to ensure high performance.

XDP has been gradually integrated into the Linux kernel over several releases, but no complete architectural description of the system as a whole has been presented before. In this work we present a high-level design description of XDP and its capabilities, and how it integrates with the rest of the Linux kernel. Our performance evaluation shows raw packet processing performance of up to 24 million packets per second per CPU core. While this does not quite match the highest achievable performance in a DPDK-based application on the same hardware, we argue that the XDP system makes up for this by offering several compelling advantages over DPDK and other kernel bypass solutions. Specifically, XDP:

- Integrates cooperatively with the regular networking stack, retaining full control of the hardware in the kernel. This retains the kernel security boundary, and requires no changes to network configuration and management tools. In addition, any network adapter with a Linux driver can be supported by XDP; no special hardware features are needed, and existing drivers only need to be modified to add the XDP execution hooks.
- Makes it possible to selectively utilise kernel network stack features such as the routing table and TCP stack, keeping the same configuration interface while accelerating critical performance paths.
- Guarantees stability of both the eBPF instruction set and the programming interface (API) exposed along with it.
- Does not require expensive packet re-injection from user space into kernel space when interacting with workloads based on the normal socket layer.
- Is transparent to applications running on the host, enabling new deployment scenarios such as inline protection against denial of service attacks on servers.
- Can be dynamically re-programmed without any service interruption, which means that features can be added on the fly or removed completely when they are not needed without interruption of network traffic, and that processing can react dynamically to conditions in other parts of the system.
- Does not require dedicating full CPU cores to packet processing, which means lower traffic levels translate directly into lower CPU usage. This has important efficiency and power saving implications.

In the rest of this paper we present the design of XDP and our performance analysis. This is structured as follows: Section 2 first outlines related work. Section 3 then presents the design of the XDP system and Section 4 presents our evaluation of its raw packet processing performance. Section 5 supplements this with examples of real-world use cases that can be implemented with XDP. Finally, Section 6 discusses future directions of XDP, and Section 7 concludes.

2 RELATED WORK

XDP is certainly not the first system enabling programmable packet processing. Rather, this field has gained momentum over the last several years, and continues to do so. Several frameworks have been presented to enable this kind of programmability, and they have enabled many novel applications. Examples of such applications include those performing single functions, such as switching [47], routing [19], named-based forwarding [28], classification [48], caching [33] or traffic generation [14]. They also include more general solutions which are highly customisable and can operate on packets from a variety of sources [12, 20, 31, 34, 40, 44].

To achieve high packet processing performance on Common Off The Shelf (COTS) hardware, it is necessary to remove any bottlenecks between the networking interface card (NIC) and the program performing the packet processing. Since one of the main sources of performance bottlenecks is the interface between the operating system kernel and the userspace applications running on top of it (because of the high overhead of a system call and complexity of the underlying feature-rich and generic stack), low-level packet processing frameworks have to manage this overhead in one way or another. The existing frameworks, which have enabled the applications mentioned above, take several approaches to ensuring high performance; and XDP builds on techniques from several of them. In the following we give a brief overview of the similarities and differences between XDP and the most commonly used existing frameworks.

The DataPlane Development Kit (DPDK) [16] is probably the most widely used framework for high-speed packet processing. It started out as an Intel-specific hardware support package, but has since seen a wide uptake under the stewardship of the Linux Foundation. DPDK is a so-called *kernel bypass* framework, which moves the control of the networking hardware out of the kernel into the networking application, completely removing the overhead of the kernel-userspace boundary. Other examples of this approach include the PF_RING ZC module [45] and the hardware-specific Solarflare OpenOnload [24]. Kernel bypass offers the highest performance of the existing frameworks [18]; however, as mentioned in the introduction, it has significant management, maintenance and security drawbacks.

XDP takes an approach that is the opposite of kernel bypass: Instead of moving control of the networking hardware *out* of the kernel, the performance-sensitive packet processing operations are moved *into* the kernel, and executed before the operating system networking stack begins its processing. This retains the advantage of removing the kernel-userspace boundary between networking hardware and packet processing code, while keeping the kernel in control of the hardware, thus preserving the management interface and the security guarantees offered by the operating system. The key innovation that enables this is the use of a virtual execution environment that verifies that loaded programs will not harm or crash the kernel.

Prior to the introduction of XDP, implementing packet processing functionality as a kernel module has been a high-cost approach, since mistakes can crash the whole system, and internal kernel APIs are subject to frequent change. For this reason, it is not surprising that few systems have taken this approach. Of those that have, the most prominent examples are the Open vSwitch [44] virtual switch and the Click [40] and Contrail [41] virtual router frameworks, which are all highly configurable systems with a wide scope, allowing them to amortise the cost over a wide variety of uses. XDP significantly lowers the cost for applications of moving processing into the kernel, by providing a safe execution environment, and by being supported by the kernel community, thus offering the same API stability guarantee as every other interface the kernel exposes to userspace. In addition, XDP programs can completely bypass the networking stack, which offers higher performance than a traditional kernel module that needs to hook into the existing stack.

While XDP allows packet processing to move into the operating system for maximum performance, it also allows the programs loaded into the kernel to selectively redirect packets to a special user-space socket type, which bypasses the normal networking stack, and can even operate in a zero-copy mode to further lower the overhead. This operating mode is quite similar to the approach used by frameworks such as Netmap [46] and PF_RING [11], which offer high packet processing performance by lowering the overhead of transporting packet data from the network device to a userspace application, without bypassing the kernel completely. The Packet I/O engine that is part of PacketShader [19] is another example of this approach, and it has some similarities with special-purpose operating systems such as Arrakis [43] and ClickOS [36].

Finally, programmable hardware devices are another way to achieve high-performance packet processing. One example is the NetFPGA [32], which exposes an API that makes it possible to run arbitrary packet processing tasks on the FPGA-based dedicated hardware. The P4 language [7] seeks to extend this programmability to a wider variety of packet processing hardware (including, incidently, an XDP backend [51]). In a sense, XDP can be thought of as a "software offload", where performance-sensitive processing is offloaded to increase performance, while applications otherwise interact with the regular networking stack. In addition, XDP programs that don't need to access kernel helper functions can be offloaded entirely to supported networking hardware (currently supported with Netronome smart-NICs [27]).

In summary, XDP represents an approach to high-performance packet processing that, while it builds on previous approaches, offers a new tradeoff between performance, integration into the system and general flexibility. The next section explains in more detail how XDP achieves this.

3 THE DESIGN OF XDP

The driving rationale behind the design of XDP has been to allow high-performance packet processing that can integrate cooperatively with the operating system kernel, while ensuring the safety and integrity of the rest of the system. This deep integration with the kernel obviously imposes some design constraints, and the components of XDP have been gradually introduced into the Linux kernel over a number of releases, during which the design has evolved through continuous feedback and testing from the community.

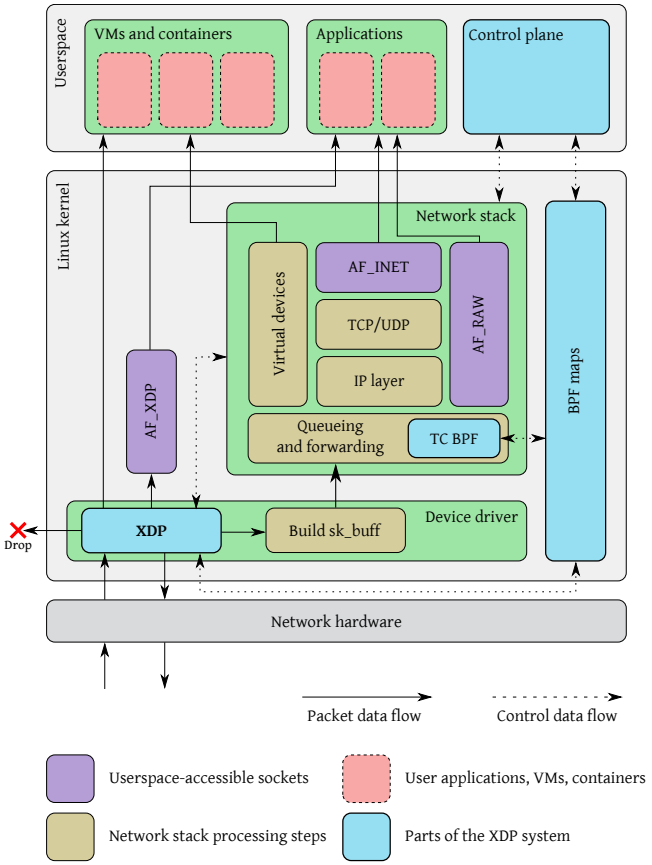

Figure 1: XDP's integration with the Linux network stack. On packet arrival, before touching the packet data, the device driver executes an eBPF program in the main XDP hook. This program can choose to drop packets; to send them back out the same interface it was received on; to redirect them, either to another interface (including vNICs of virtual machines) or to userspace through special AF_XDP sockets; or to allow them to proceed to the regular networking stack, where a separate TC BPF hook can perform further processing before packets are queued for transmission. The different eBPF programs can communicate with each other and with userspace through the use of BPF maps. To simplify the diagram, only the ingress path is shown.

Unfortunately, recounting the process and lessons learned is not possible in the scope of this paper. Instead, this section describes the complete system, by explaining how the major components of XDP work, and how they fit together to create the full system. This is illustrated by Figure 1, which shows a diagram of how XDP integrates into the Linux kernel, and Figure 2, which shows the execution flow of a typical XDP program. There are four major components of the XDP system:

- **The XDP driver hook** is the main entry point for an XDP program, and is executed when a packet is received from the hardware.

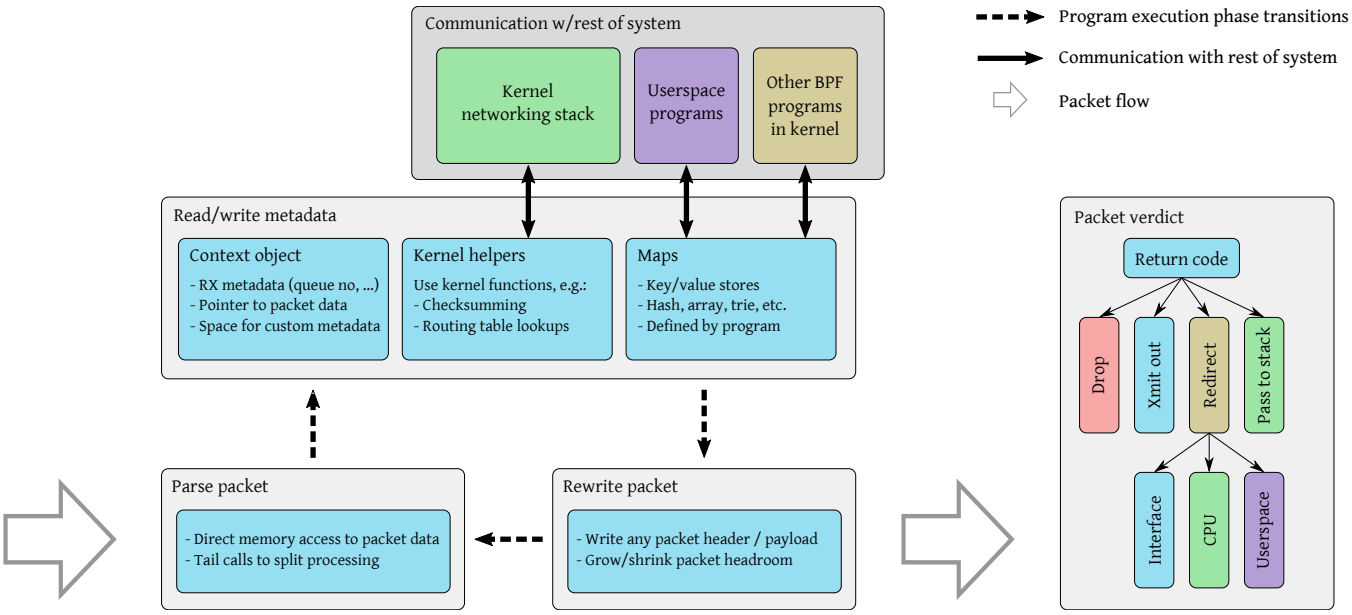

Figure 2: Execution flow of a typical XDP program. When a packet arrives, the program starts by parsing packet headers to extract the information it will react on. It then reads or updates metadata from one of several sources. Finally, a packet can be rewritten and a final verdict for the packet is determined. The program can alternate between packet parsing, metadata lookup and rewriting, all of which are optional. The final verdict is given in the form of a program return code.

- **The eBPF virtual machine** executes the byte code of the XDP program, and just-in-time-compiles it for increased performance.
- **BPF maps** are key/value stores that serve as the primary communication channel to the rest of the system.
- **The eBPF verifier** statically verifies programs before they are loaded to make sure they do not crash or corrupt the running kernel.

3.1 The XDP Driver Hook

An XDP program is run by a hook in the network device driver each time a packet arrives. The infrastructure to execute the program is contained in the kernel as a library function, which means that the program is executed directly in the device driver, without context switching to userspace. As shown in Figure 1, the program is executed at the earliest possible moment after a packet is received from the hardware, before the kernel allocates its per-packet *sk_buff* data structure or performs any parsing of the packet.

Figure 2 shows the various processing steps typically performed by an XDP program. The program starts its execution with access to a context object. This object contains pointers to the raw packet data, along with metadata fields describing which interface and receive queue the packet was received on.

The program typically begins by parsing packet data, and can pass control to a different XDP program through tail calls, thus splitting processing into logical sub-units (based on, say, IP header version).

After parsing the packet data, the XDP program can use the context object to read metadata fields associated with the packet, describing the interface and receive queue the packet came from. The context object also gives access to a special memory area, located adjacent in memory to the packet data. The XDP program can use this memory to attach its own metadata to the packet, which will be carried with it as it traverses the system.

In addition to the per-packet metadata, an XDP program can define and access its own persistent data structures (through BPF maps, described in Section 3.3 below), and it can access kernel facilities through various helper functions. Maps allow the program to communicate with the rest of the system, and the helpers allow it to selectively make use of existing kernel functionality (such as the routing table), without having to go through the full kernel networking stack. New helper functions are actively added by the kernel development community in response to requests from the community, thus continuously expanding the functionality that XDP programs can make use of.

Finally, the program can write any part of the packet data, including expanding or shrinking the packet buffer to add or remove headers. This allows it to perform encapsulation or decapsulation, as well as, for instance, rewrite address fields for forwarding. Various kernel helper functions are available to assist with things like checksum calculation for a modified packet.

These three steps (reading, metadata processing, and writing packet data) correspond to the light grey boxes on the left side of Figure 2. Since XDP programs can contain arbitrary instructions, the different steps can alternate and repeat in arbitrary ways. However, to achieve high performance, it is often necessary to structure the execution order as described here.

At the end of processing, the XDP program issues a final verdict for the packet. This is done by setting one of the four available return codes, shown on the right-hand side of Figure 2. There are

three simple return codes (with no parameters), which can drop the packet, immediately re-transmit it out the same network interface, or allow it to be processed by the kernel networking stack. The fourth return code allows the XDP program to *redirect* the packet, offering additional control over its further processing.

Unlike the other three return codes, the redirect packet verdict requires an additional parameter that specifies the redirection target, which is set through a helper function before the program exits. The redirect functionality can be used (1) to transmit the raw packet out a different network interface (including virtual interfaces connected to virtual machines), (2) to pass it to a different CPU for further processing, or (3) to pass it directly to a special userspace socket address family (`AF_XDP`). These different packet paths are shown as solid lines in Figure 1. The decoupling of the return code and the target parameter makes redirection a flexible forwarding mechanism, which can be extended with additional target types without requiring any special support from either the XDP programs themselves, or the device drivers implementing the XDP hooks. In addition, because the redirect parameter is implemented as a map lookup (where the XDP program provides the lookup key), redirect targets can be changed dynamically without modifying the program.

3.2 The eBPF Virtual Machine

XDP programs run in the Extended BPF (eBPF) virtual machine. eBPF is an evolution of the original BSD packet filter (BPF) [37] which has seen extensive use in various packet filtering applications over the last decades. BPF uses a register-based virtual machine to describe filtering actions. The original BPF virtual machine has two 32-bit registers and understands 22 different instructions. eBPF extends the number of registers to eleven, and increases register widths to 64 bits. The 64-bit registers map one-to-one to hardware registers on the 64-bit architectures supported by the kernel, enabling efficient just-in-time (JIT) compilation into native machine code. Support for compiling (restricted) C code into eBPF is included in the LLVM compiler infrastructure [29].

eBPF also adds new instructions to the eBPF instruction set. These include arithmetic and logic instructions for the larger register sizes, as well as a *call* instruction for function calls. eBPF adopts the same calling convention as the C language conventions used on the architectures supported by the kernel. Along with the register mapping mentioned above, this makes it possible to map a BPF call instruction to a single native call instruction, enabling function calls with close to zero additional overhead. This facility is used by eBPF to support helper functions that eBPF programs can call to interact with the kernel while processing, as well as for function calls within the same eBPF program.

While the eBPF instruction set itself can express any general purpose computation, the verifier (described in Section 3.4 below) places limitations on the programs loaded into the kernel to ensure that the user-supplied programs cannot harm the running kernel. With this in place, it is safe to execute the code directly in the kernel address space, which makes eBPF useful for a wide variety of tasks in the Linux kernel, not just for XDP. Because all eBPF programs can share the same set of maps, this makes it possible for programs to react to arbitrary events in other parts of the kernel. For instance, a separate eBPF program could monitor CPU load and instruct an XDP program to drop packets if load increases above a certain threshold.

The eBPF virtual machine supports dynamically loading and re-loading programs, and the kernel manages the life cycle of all programs. This makes it possible to extend or limit the amount of processing performed for a given situation, by adding or completely removing parts of the program that are not needed, and re-loading it atomically as requirements change. The dynamic loading of programs also makes it possible to express processing rules directly in program code, which for some applications can increase performance by replacing lookups into general purpose data structures with simple conditional jumps.

3.3 BPF Maps

eBPF programs are executed in response to an event in the kernel (a packet arrival, in the case of XDP). Each time they are executed they start in the same initial state, and they do not have access to persistent memory storage in their program context. Instead, the kernel exposes helper functions giving programs access to *BPF maps*.

BPF maps are key/value stores that are defined upon loading an eBPF program, and can be referred to from within the eBPF code. Maps exist in both global and per-CPU variants, and can be shared, both between different eBPF programs running at various places in the kernel, as well as between eBPF and userspace. The map types include generic hash maps, arrays and radix trees, as well as specialised types containing pointers to eBPF programs (used for tail calls), or redirect targets, or even pointers to other maps.

Maps serve several purposes: they are a persistent data store between invocations of the same eBPF program; a global coordination tool, where eBPF programs in one part of the kernel can update state that changes the behaviour in another; and a communication mechanism between userspace programs and the kernel eBPF programs, similar to the communication between control plane and data plane in other programmable packet processing systems.

3.4 The eBPF Verifier

Since eBPF code runs directly in the kernel address space, it can directly access, and potentially corrupt, arbitrary kernel memory. To prevent this from happening, the kernel enforces a single entry point for loading all eBPF programs (through the `bpf()` system call). When loading an eBPF program it is first analysed by the in-kernel eBPF verifier. The verifier performs a static analysis of the program byte code to ensure that the program performs no actions that are unsafe (such as accessing arbitrary memory), and that the program will terminate. The latter is ensured by disallowing loops and limiting the maximum program size. The verifier works by first building a directed acyclic graph (DAG) of the control flow of the program. This DAG is then verified as follows:

First, the verifier performs a depth-first search on the DAG to ensure it is in fact acyclic, i.e., that it contains no loops, and also that it contains no unsupported or unreachable instructions. Then, in a second pass, the verifier walks all possible paths of the DAG. The purpose of this second pass is to ensure that the program performs only safe memory accesses, and that any helper functions are

called with the right argument types. This is ensured by rejecting programs that perform load or call instructions with invalid arguments. Argument validity is determined by tracking the state of all registers and stack variables through the execution of the program.

The purpose of this register state tracking mechanism is to ensure that the program performs no out of bounds memory accesses *without knowing in advance what the valid bounds are*. The bounds cannot be known because programs must process data packets which vary in size; and similarly, the contents of maps are not known in advance, so it is not known whether a given lookup will succeed. To deal with this, the verifier checks that the program being loaded does its own bounds checking before dereferencing pointers to packet data, and that map lookups are checked for `NULL` values before being dereferenced. This approach leaves the program writer in control of how checks are integrated into the processing logic, and what to do in the error path.

To track data access, the verifier tracks data types, pointer offsets and possible value ranges of all registers. At the beginning of the program, `R1` contains a pointer to the *context* metadata object, `R10` is a stack pointer, and all other registers are marked as not initialised. At each execution step, register states are updated based on the operations performed by the program. When a new value is stored in a register, that register inherits the state variables from the source of the value. Arithmetic operations will affect the possible value ranges of scalar types, and the offsets of pointer types. The widest possible range is stored in the state variables, e.g., if a one-byte load is performed into a register, that register's possible value range is set to 0-255. Branches in the instruction graph will update the register state according to the logical operation contained in the branch. For example, given a comparison such as `"R1 > 10"`, the verifier will set the maximum value of `R1` to 10 in one branch, and the minimum value to 11 in the other.

Using this range information stored in the state variables, it is possible for the verifier to predict the ranges of memory that each load instruction can potentially access, and ensure that only safe memory accesses are performed. For packet data access this is done by looking for comparisons with the special `data_end` pointer that is available in the context object; for values retrieved from a BPF map the data size in the map definition is used; and for values stored on the stack, accesses are checked against the data ranges that have previously been written to. Furthermore, restrictions are placed on pointer arithmetic, and pointers cannot generally be converted to integer values. Any eBPF program that performs operations that the verifier cannot prove are safe, are simply rejected at load time. In addition to this, the verifier also uses the range information to enforce aligned memory accesses.

It should be noted that the purpose of the verifier is to protect the internals of the kernel from being exposed to malicious or buggy eBPF programs, not to ensure that the programs perform their designated function in the most efficient way possible. That is, an XDP program can slow down the machine by performing excessive processing (up to the maximum program size), and it can corrupt network packets if written incorrectly. Loading programs requires administrative (root) privileges for this reason, and it is up to the eBPF programmer to prevent these types of bugs, and to the system administrator to decide which programs to load on the system.

3.5 Example XDP program

To showcase the features described above, Listing 1 shows an example of a simple XDP program. The program will parse packet headers, and reflect all UDP packets by swapping the source and destination MAC addresses and sending the packet back out the interface it came in on. While this is obviously a very simple example, the program does feature most of the components of an XDP program that is useful in the real world. Specifically:

- A BPF map is defined (lines 1–7) for keeping statistics of the number of processed packets. The map is keyed on IP protocol number and each value is simply a packet count (updated in lines 60–62). A userspace program can poll this map to output statistics while the XDP program is running.
- Pointers to the start and end of the packet data is read from the context object (lines 30–31), to be used for direct packet data access.
- Checking against the data_end pointer ensures that no data is read out of bounds (lines 22, 36 and 47). The verifier ensures correctness even across pointer copies (as in lines 21–22).
- The program must handle any packet parsing itself, including things such as VLAN headers (lines 41–50).
- Direct packet data access is used to modify the packet headers (lines 14–16).
- The map lookup helper function exposed by the kernel (called on line 60). This is the only real function call in the program; all other functions are inlined on compilation, including helpers like `htons()`.
- The final packet verdict is communicated by the program return code (line 69).

When the program is installed on an interface, it is first compiled into eBPF byte code, then checked by the verifier. The notable things checked by the verifier in this case are (a) the absence of loops, and the total size of the program, (b) that all direct packet data accesses are preceded by appropriate bounds checking (c) that the sizes of parameters passed to the map lookup function matches the map definition, and (d) that the return value from the map lookup is checked against NULL before it is accessed.

3.6 Summary

The XDP system consists of four major components: (1) The XDP device driver hook which is run directly after a packet is received from the hardware. (2) The eBPF virtual machine which is responsible for the actual program execution (and is also used for executing programs in other parts of the kernel). (3) BPF maps, which allow programs running in various parts of the kernel to communicate with each other and with userspace. And (4) The eBPF verifier, which ensures programs do not perform any operations that can harm the running kernel.

These four components combine to create a powerful environment for writing custom packet processing applications, that can accelerate packet processing in essential paths, while integrating with the kernel and making full use of its existing facilities. The performance achievable by these applications is the subject of the next section.

```c
/* map used to count packets; key is IP protocol, value is pkt count */
struct bpf_map_def SEC("maps") rxcnt = {
    .type = BPF_MAP_TYPE_PERCPU_ARRAY,
    .key_size = sizeof(u32),
    .value_size = sizeof(long),
    .max_entries = 256,
};

/* swaps MAC addresses using direct packet data access */
static void swap_src_dst_mac(void *data)
{
    unsigned short *p = data;
    unsigned short dst[3];
    dst[0] = p[0];   dst[1] = p[1];   dst[2] = p[2];
    p[0]   = p[3];   p[1]   = p[4];   p[2]   = p[5];
    p[3]   = dst[0]; p[4]   = dst[1]; p[5]   = dst[2];
}

static int parse_ipv4(void *data, u64 nh_off, void *data_end)
{
    struct iphdr *iph = data + nh_off;
    if (iph + 1 > data_end)
        return 0;
    return iph->protocol;
}

SEC("xdp1") /* marks main eBPF program entry point */
int xdp_prog1(struct xdp_md *ctx)
{
    void *data_end = (void *)(long)ctx->data_end;
    void *data = (void *)(long)ctx->data;
    struct ethhdr *eth = data; int rc = XDP_DROP;
    long *value; u16 h_proto; u64 nh_off; u32 ipproto;

    nh_off = sizeof(*eth);
    if (data + nh_off > data_end)
        return rc;

    h_proto = eth->h_proto;

    /* check VLAN tag; could be repeated to support double-tagged VLAN */
    if (h_proto == htons(ETH_P_8021Q) || h_proto == htons(ETH_P_8021AD)) {
        struct vlan_hdr *vhdr;

        vhdr = data + nh_off;
        nh_off += sizeof(struct vlan_hdr);
        if (data + nh_off > data_end)
            return rc;
        h_proto = vhdr->h_vlan_encapsulated_proto;
    }

    if (h_proto == htons(ETH_P_IP))
        ipproto = parse_ipv4(data, nh_off, data_end);
    else if (h_proto == htons(ETH_P_IPV6))
        ipproto = parse_ipv6(data, nh_off, data_end);
    else
        ipproto = 0;

    /* lookup map element for ip protocol, used for packet counter */
    value = bpf_map_lookup_elem(&rxcnt, &ipproto);
    if (value)
        *value += 1;

    /* swap MAC addrs for UDP packets, transmit out this interface */
    if (ipproto == IPPROTO_UDP) {
        swap_src_dst_mac(data);
        rc = XDP_TX;
    }
    return rc;
}
```

Listing 1: Example XDP program. The program parses packet headers, swaps source and destination MAC addresses for all UDP packets, and sends them back out the same interface. A packet counter is kept per IP protocol number. Adapted from xdp2_kern.c, which is distributed with the kernel source code.

4 PERFORMANCE EVALUATION

In this section we present our performance evaluation of XDP. As mentioned in Section 2, there are quite a few existing systems for high-performance packet processing, and benchmarking all of them is not feasible in the scope of this paper. Instead, we note that DPDK is the existing solution that achieves the highest performance [18], and compare against that as a baseline for the current state of the art in high-speed software packet processing (using the `testpmd` example application shipped with the 18.05 release of DPDK). We focus on the raw packet processing performance, using synthetic benchmarks, and also compare against the performance of the Linux kernel network stack, to show the performance improvements offered by XDP in the same system. In the next section, we supplement these raw performance benchmarks with some examples of real-world applications implemented on top of XDP, to demonstrate their feasibility within the programming model.

For all benchmarks, we use a machine equipped with a hexa-core Intel Xeon E5-1650 v4 CPU running at 3.60GHz, which supports Intel's Data Direct I/O (DDIO) technology allowing the networking hardware Direct Memory Access (DMA) system to place packet data directly in the CPU cache. The test machine is equipped with two Mellanox ConnectX-5 Ex VPI dual-port 100Gbps network adapters, which are supported by the *mlx5* driver. We use the TRex packet generator [9] to produce the test traffic. The test machine runs a pre-release of version 4.18 of the Linux kernel. To help others reproduce our results, we make available the full details of our setup, along with links to source code and the raw test data, in an online repository [22].

In our evaluation, we focus on three metrics:

- Packet drop performance. To show the maximum packet processing performance, we measure the performance of the simplest possible operation of dropping the incoming packet. This effectively measures the overhead of the system as a whole, and serves as an upper bound on the expected performance of a real packet processing application.
- CPU usage. As mentioned in the introduction, one of the benefits of XDP is that it scales the CPU usage with the packet load, instead of dedicating CPU cores exclusively to packet processing. We quantify this by measuring how CPU usage scales with the offered network load.
- Packet forwarding performance. A packet processing system that cannot forward packets has limited utility. Since forwarding introduces an additional complexity compared to the simple processing case (e.g., interacting with more than one network adapter, rewriting link-layer headers, etc.), a separate evaluation of forwarding performance is useful. We include both throughput and latency in the forwarding evaluation.

We have verified that with full-sized (1500 bytes) packets, our system can process packets at line-speed (100 Gbps) on a single core that is half-idle. This makes it clear that the challenge is processing many *packets* per second, as others have also noted [46]. For this reason, we perform all tests using minimum-sized (64 bytes) packets, and measure the maximum number of packets per second the system can process. To measure how performance scales with the number of CPU cores, we repeat the tests with an increasing

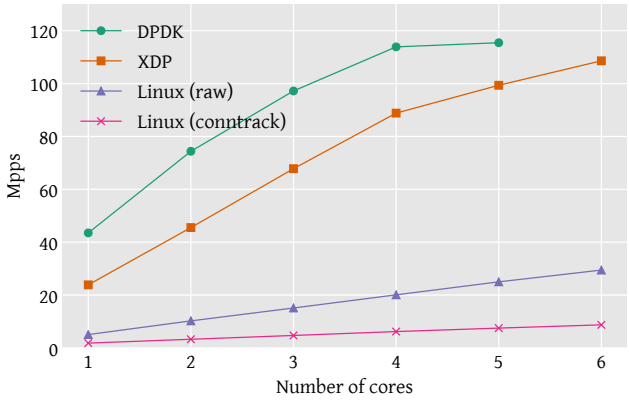

Figure 3: Packet drop performance. DPDK uses one core for control tasks, so only 5 are available for packet processing.

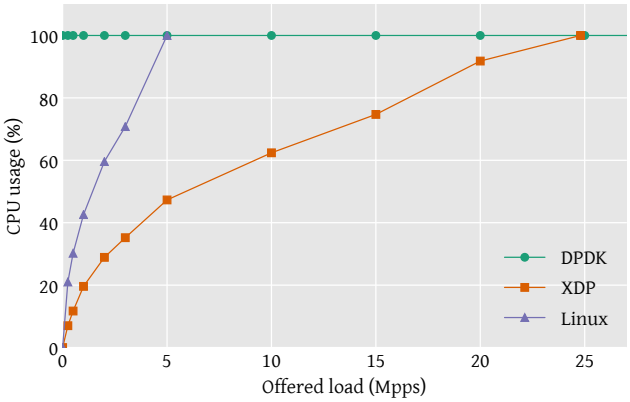

Figure 4: CPU usage in the drop scenario. Each line stops at the method's maximum processing capacity. The DPDK line continues at 100% up to the maximum performance shown in Figure 3.

number of cores dedicated to packet processing.[1] For XDP and the Linux network stack (which do not offer an explicit way to dedicate cores to packet processing) we achieve this by configuring the hardware Receive Side Scaling (RSS) feature to steer traffic to the desired number of cores for each test.

As we will see in the results below, our tests push the hardware to its very limits. As such, tuning the performance of the system as a whole is important to realise optimal performance. This includes the physical hardware configuration, configuration of the network adapter features such as Ethernet flow control and receive queue size, and configuration parameters of the Linux kernel, where we for instance disable full preemption and the "retpoline" mitigation for the recent Meltdown and Spectre vulnerabilities. The full details of these configuration issues are omitted here due to space constraints, but are available in the online repository.

The following subsections present the evaluation results for each of the metrics outlined above, followed by a general discussion of the performance of XDP compared to the other systems. As all our results are highly repeatable, we show results from a single test run (with no error bars) to make the graphs more readable.

4.1 Packet Drop Performance

Figure 3 shows the packet drop performance as a function of the number of cores. The baseline performance of XDP for a single core is 24 Mpps, while for DPDK it is 43.5 Mpps. Both scale their performance linearly until they approach the global performance limit of the PCI bus, which is reached at 115 Mpps after enabling PCI descriptor compression support in the hardware (trading CPU cycles for PCI bus bandwidth).

The figure also shows the performance of the Linux networking stack in two configurations: one where we use the "raw" table of the *iptables* firewall module to drop packets, which ensures the earliest possible drop in the network stack processing; and another where we use the connection tracking (*conntrack*) module, which carries a high overhead, but is enabled by default on many Linux distributions. These two modes illustrate the performance span of the Linux networking stack, from 1.8 Mpps of single-core performance with conntrack, up to 4.8 Mpps in raw mode. It also shows that in the absence of hardware bottlenecks, Linux performance scales linearly with the number of cores. And finally, it shows that with its 24 Mpps on a single core, XDP offers a five-fold improvement over the fastest processing mode of the regular networking stack.

As part of this Linux raw mode test, we also measured the overhead of XDP by installing an XDP program that does no operation other than updating packets counters and passing the packet on to the stack. We measured a drop in performance to 4.5 Mpps on a single core, corresponding to 13.3 ns of processing overhead. This is not shown on the figure, asthe difference is too small to be legible.

4.2 CPU Usage

We measure the CPU usage of the different tested systems when running the packet drop application on a single CPU core, by recording the percentage of CPU busy time using the `mpstat` system utility. The results of this is shown in Figure 4. The test varies the offered packet load up to the maximum that each system can handle on a single core.

Since DPDK by design dedicates a full core to packet processing, and uses busy polling to process the packets, its CPU usage is always pegged at 100%, which is the green line at the top of the figure. In contrast, both XDP and Linux smoothly scale CPU usage with the offered load, with a slightly larger relative increase in CPU usage at a small offered load level.

The non-linearity of the graph in the bottom-left corner is due to the fixed overhead of interrupt processing. At lower packet rates, the number of packets processed during each interrupt is smaller, leading to higher CPU usage per packet.

4.3 Packet Forwarding Performance

Figure 5 shows packet forwarding performance. The forwarding applications perform a simple Ethernet address rewrite, where the source and destination address of the incoming packet are swapped before the packet is forwarded. This is the minimum rewriting that is needed for packet forwarding to function, so the results

[1] The Hyperthreading feature of the CPU is disabled for our experiments, so whenever we refer to the number of active CPU cores, this means the number of physical cores.

Table 1: Packet forwarding latency. Measurement machine connected to two ports on the same NIC, measuring end-to-end latency for 50 seconds with high and low packet rates (100 pps and 1 Mpps).

	Average		Maximum		$< 10\mu s$	
	100 pps	1 Mpps	100 pps	1 Mpps	100 pps	1 Mpps
XDP	$82\mu s$	$7\mu s$	$272\mu s$	$202\mu s$	0%	98.1%
DPDK	$2\mu s$	$3\mu s$	$161\mu s$	$189\mu s$	99.5%	99.0%

represent an upper bound on forwarding performance. Since XDP can forward packets out the same NIC as well as out a different NIC (using two different program return codes), we include both modes in the graph. The DPDK example program only supports forwarding packets through a different interface, so we only include this operating mode in the test. Finally, the Linux networking stack does not support this minimal forwarding mode, but requires a full bridging or routing lookup to forward packets; this lookup is expensive, and since the other applications do not perform it, the results are not directly comparable. For this reason, we omit the Linux networking stack from these results, and instead include the Linux routing performance in our routing use case presented in Section 5.1.

As Figure 5 shows, we again see linear scaling with the number of cores up to a global performance bottleneck. The absolute performance is somewhat lower than for the packet drop case, which shows the overhead of packet forwarding. We also see that the XDP performance improves significantly when packets are sent out on the same interface that they were received on, surpassing the DPDK forwarding performance at two cores and above. The performance difference is primarily due to differences in memory handling: packet buffers are allocated by the device driver and associated with the receiving interface. And so, when the packet is forwarded out a different interface, the memory buffer needs to be returned to the interface that it is associated with.

Looking at forwarding latency, as seen in Table 1, the relative performance of XDP and DPDK for different-NIC forwarding are reflected for the high packet rate test (with DPDK showing slightly lower variance as well). However, for low packet rates, the latency of XDP is dominated by the interrupt processing time, which leads to much higher end-to-end latency than DPDK achieves with constant polling.

4.4 Discussion

As we have seen in the previous subsections, XDP achieves significantly higher performance than the regular Linux networking stack. Even so, for most use cases XDP does not quite match the performance of DPDK. We believe this is primarily because DPDK has incorporated more performance optimisations at the lowest level of the code. To illustrate this, consider the packet drop example: XDP achieves 24 Mpps on a single core, which corresponds to 41.6 nanoseconds per packet, while DPDK achieves 43.5 Mpps, or 22.9 nanoseconds per packet. The difference of 18.7 nanoseconds corresponds to 67 clock cycles on the 3.6 GHz processor in our test machine. Thus, it is clear that every micro-optimisation counts; for example, we measure an overhead of 1.3 nanoseconds for a

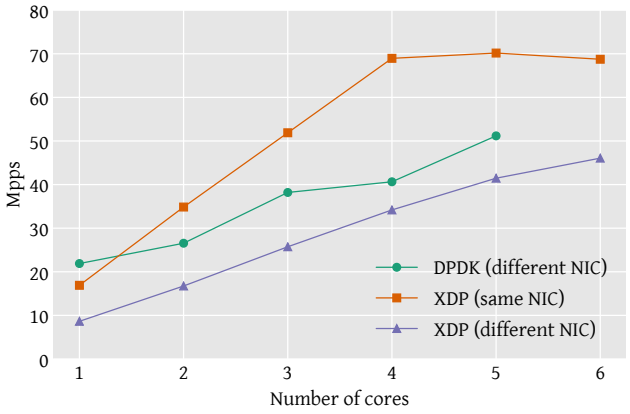

Figure 5: Packet forwarding throughput. Sending and receiving on the same interface takes up more bandwidth on the same PCI port, which means we hit the PCI bus limit at 70 Mpps.

single function call on our test system. The mlx5 driver performs 10 function calls when processing a single packet, corresponding to 13 of the 18.7 nanoseconds of performance difference between XDP and DPDK.

Some of this overhead is inevitable on a general-purpose operating system such as Linux, as device drivers and subsystems are structured in a way that makes it possible to support a wide variety of systems and configurations. However, we believe that some optimisations are viable. For instance, we have performed an experiment that removed DMA-related function calls that were not needed on our specific hardware from the driver, removing four of the 10 per-packet function calls. This improved the packet drop performance to 29 Mpps. Extrapolating this, removing all function calls would increase performance to 37.6 Mpps. While this is not possible in practice, it is possible to remove some of them, and combining this with other performance optimisations, we believe it is reasonable to expect the performance gap between DPDK and XDP to lessen over time. We see similar effects with other drivers, such as the *i40e* driver for 40 Gbps Intel cards, which achieves full performance up to the NIC hardware performance limit with both XDP and DPDK.[2]

Given the above points, we believe it is feasible for XDP to further decrease the performance delta to DPDK. However, given the benefits of XDP in terms of flexibility and integration with the rest of the system, XDP is already a compelling choice for many use cases; we show some examples of this in the next section.

5 REAL-WORLD USE CASES

To show how the various aspects of XDP can be used to implement useful real-world applications, this section describes three example use cases. These use cases have all seen deployment in one form or another, although we use simplified versions in our evaluation to be able to make the code available. We also refer the reader to [38]

[2]While DPDK uses the drivers in the operating system to assume control of the hardware, it contains its own drivers that are used for the actual packet processing.

for an independent look at some of the challenges of implementing real-world network services in XDP.

The purpose of this section is to demonstrate the feasibility of implementing each use case in XDP, so we do not perform exhaustive performance evaluations against state of the art implementations. Instead, we use the regular Linux kernel stack as a simple performance baseline and benchmark the XDP applications against that. The three use cases are a software router, an inline Denial of Service (DoS) mitigation application and a layer-4 load balancer.

5.1 Software Routing

The Linux kernel contains a full-featured routing table, which includes support for policy routing, source-specific routing, multipath load balancing, and more. For the control plane, routing daemons such as Bird [10] or FRR [17] implement a variety of routing control plane protocols. Because of this rich ecosystem supporting routing on Linux, re-implementing the routing stack in another packet processing framework carries a high cost, and improving performance of the kernel data plane is desirable.

XDP is a natural fit for this task, especially as it includes a helper function which performs full routing table lookups directly from XDP. The result of the lookup is an egress interface and a next-hop MAC address, which makes it possible for the XDP program to immediately forward the packet if the lookup succeeds. If no next-hop MAC is known (because neighbour lookup has not been performed yet), the XDP program can pass the packet to the networking stack, which will resolve the neighbour, allowing subsequent packets to be forwarded by XDP.

To show the performance of this use case, we use the XDP routing example that is included in the Linux kernel source [1] and compare its performance to routing in the regular Linux network stack. We perform two tests: one with a single route installed in the routing table, and another where we use a full dump of the global BGP routing table from *routeviews.org*. In both cases, all next-hop addresses are set to the address of the test system connected to our egress interface. The full table contains 752,138 distinct routes, and for our tests we generate 4000 random destination IP addresses to make sure we exercise the full table.[3].

The performance of this use case is seen in Figure 6. Using XDP for the forwarding plane improves performance with a factor of 2.5 for a full table lookup, and a factor of 3 for the smaller routing table example. This makes it feasible to run a software router with a full BGP table at line rate on a 10 Gbps link using a single core (using a conservative estimate of an average packet size of 300 bytes).

5.2 Inline DoS Mitigation

DoS attacks continue to plague the internet, typically in the form of distributed attacks (DDoS attacks) from compromised devices. With XDP, it is possible to deploy packet filtering to mitigate such attacks directly at the application servers, without needing to change applications. In the case of a virtual machine deployment, the filter can even be installed in the hypervisor, and thus protect all virtual machines running on the host.

[3]Using fewer than 4000 destination IPs, the part of the routing table that is actually used is small enough to be kept in the CPU cache, which gives misleading (better) results. Increasing the number of IPs above 4000 had no additional effects on forwarding performance.

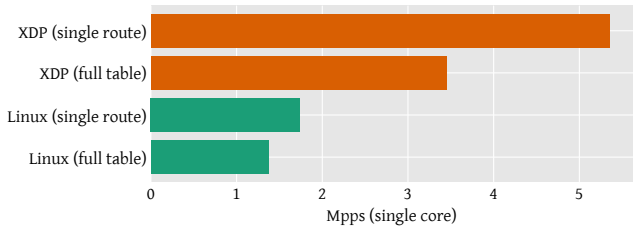

Figure 6: Software routing performance. Since the performance scales linearly with the number of cores, only the results for a single core are shown.

To show how this could work, we perform a test modelled on the DDoS mitigation architecture used by Cloudflare, which uses XDP as the filtering mechanism [6]. Their Gatebot architecture works by sampling traffic at servers located in distributed Points of Presence (PoPs), collecting it centrally for analysis, and formulating mitigation rules based on the analysis. The mitigation rules take the form of a series of simple checks on the packet payload, which are compiled directly into eBPF code and distributed to the edge servers in the PoPs. Here the code is executed as an XDP program that will drop packets matching the rules, while also updating match counters stored in BPF maps.

To test the performance of such a solution, we use an XDP program that parses the packet headers and performs a small number of tests[4] to identify attack traffic and drop it, and uses the CPU redirect feature to pass all other packets to a different CPU core for processing. To simulate a baseline application load we use the Netperf benchmarking tool [26]. Netperf contains a TCP-based round-trip benchmark, which opens a TCP connection and sends a small payload that is echoed back from the server, repeating as soon as a reply is received. The output is the number of transactions per second, which represents performance of an interactive use case, such as small remote procedure calls.

We run our experiment on a single core, to illustrate the situation where legitimate traffic has to compete for the same hardware resources as attack traffic. We apply a baseline load of 35.000 TCP transactions per second, then simulate the DoS attack by offering an increasing load of small UDP packets matching our packet filter. We measure the TCP transactions performance as the attack traffic volume increases, reporting the average of four test repetitions per data point.

The results of this is shown in Figure 7. Without the XDP filter, performance drops rapidly, being halved at 3 Mpps and effectively zero at just below 3.5 Mpps of attack traffic. However, with the XDP filter in place, the TCP transaction performance is stable at around 28.500 transactions per second until 19.5 Mpps of attack traffic, after which it again drops rapidly. This shows that effective DDoS filtering is feasible to perform in XDP, which comfortably handles 10 Gbps of minimum-packet DoS traffic on a single CPU core. Deploying DDoS mitigation this way leads to increased flexibility, since no special hardware or application changes are needed.

[4]Our example program performs four packet data reads per packet, to parse the outer packet headers and drop packets with a pre-defined UDP destination port number.

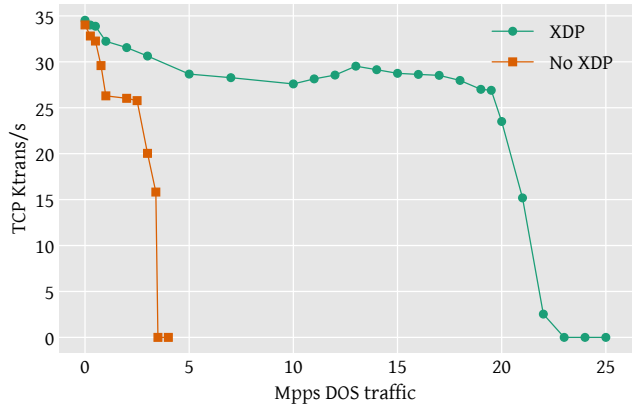

Figure 7: DDoS performance. Number of TCP transactions per second as the level of attack traffic directed at the server increases.

Table 2: Load balancer performance (Mpps).

CPU Cores	1	2	3	4	5	6
XDP (Katran)	5.2	10.1	14.6	19.5	23.4	29.3
Linux (IPVS)	1.2	2.4	3.7	4.8	6.0	7.3

5.3 Load Balancing

For the load balancer use case, we use the XDP component of the Katran load balancer [15] released as open source by Facebook. This works by announcing an IP address for the service, which is routed to the load balancer. The load balancer hashes the source packet header to select a destination application server. The packet is then encapsulated and sent to the application server, which is responsible for decapsulating it, processing the request, and replying directly to the originator. The XDP program performs the hashing and encapsulation, and returns the packet out the same interface on which it was received. It keeps configuration data in BPF maps and implements the encapsulation entirely in the eBPF program.

To test this use case, we configure the Katran XDP program with a fixed number of destination hosts[5], and run it on our test machine. We compare it with the IPVS load balancer that is part of the Linux kernel, which can be configured in the same way. The performance of both is shown in Table 2, which shows linear scaling with the number of CPUs, and that XDP offers a performance gain of 4.3x over IPVS.

6 FUTURE DIRECTIONS OF XDP

As we have shown above, XDP offers high performance and can be used to implement a variety of real-world use cases. However, this does not mean that XDP is a finished system. On the contrary, as part of the Linux kernel, XDP undergoes continuous improvement. Some of this development effort goes into softening the rough edges that are the inevitable result of XDP being incrementally incorporated into a general purpose operating system kernel. Other efforts continue to push the boundaries of XDP's capabilities. In this section we discuss some of these efforts.

6.1 Limitations on eBPF programs

As mentioned previously, the programs loaded into the eBPF virtual machine are analysed by the eBPF verifier, which places certain limitations on the programs to ensure they do not harm the running kernel. These limitations fall in two categories: (a) Ensuring the program will terminate, which is implemented by disallowing loops and limiting the maximum size of the program. And (b) ensuring the safety of memory accesses, which is done by the register state tracking explained in Section 3.4.

Since the primary function of the verifier is to ensure the safety of the kernel, a conservative approach is taken, and the verifier will reject any program that it cannot prove is safe. This can lead to false negatives, where safe programs are needlessly rejected; reducing such cases is an ongoing effort. The error messages reported by the verifier have also been made friendlier, to make it easier for developers to change their code to fix verification errors when they do occur. Support for function calls in eBPF has recently been added, support for bounded loops is planned, and efficiency improvements of the verifier itself are being worked on, which will allow it to operate on larger programs.

Another limitation of eBPF programs compared to user-space C programs is the lack of a standard library, including things like memory allocation, threading, locking, etc. This is partly alleviated by the life cycle and execution context management of the kernel (i.e., an XDP program is automatically run for each arriving packet), and partly by the helper functions exposed by the kernel.

Finally, only one XDP program can be attached to each networking interface. This can be worked around by cooperation between programs, where the tail call functionality can be used to either dynamically dispatch to different programs depending on packet content, or to chain several programs together.

6.2 User Experience and Debugging

Since an XDP program runs in the kernel, the debugging tools available to a regular userspace program are not generally applicable. Instead, the debugging and introspection features included in the kernel can be applied to XDP (and other eBPF programs). These tools include tracepoints and kprobes [13] as well as the performance counters that are part of the *perf* subsystem [42]. However, developers who are not familiar with the kernel ecosystem may find this ecosystem of kernel-specific tools a limitation. To ease the transition, a variety of tools exist, including the BPF Compiler Collection [50], the *bpftool* introspection program [8] and the *libbpf* library of utility functions [30]. These have already seen significant improvements, but more work is needed in this area.

6.3 Driver Support

Each device driver needs to add support for running XDP programs, by supporting an API exposed by the core networking stack, and support is continuously being added to more and more drivers.[6]

[5] We use one virtual IP per CPU core, and 100 destinations per virtual IP.

[6] At the time of writing Linux 4.18 has XDP support in 12 different drivers, including most high-speed network adapters. For an updated list, see [2].

However, since features are usually implemented in smaller increments, some care is still needed when selecting hardware to use with XDP, to ensure full support for the features needed for a particular use case. However, since XDP is integrated into the kernel device driver model, it imposes no particular capability constraints on the hardware, which means that full support in all drivers is possible.

As the XDP system has evolved, the need to keep the changes required in drivers to support XDP to a minimum has become increasingly clear, and some steps have been taken in this direction. For instance, support for new targets can be added to the redirection action without any changes needed from the drivers. Finally, the Generic XDP feature [39] allows running XDP programs (at reduced performance) even if the networking driver lacks the proper support, by moving execution into the core networking stack.

6.4 Performance Improvements

As we discussed in Section 4.4, there is still a performance gap between XDP and DPDK in some use cases. Efforts to improve this are ongoing, which includes micro-optimisations of driver code as well as changes to the core XDP code to remove unnecessary operations, and amortise processing costs through improved batching.

6.5 QoS and Rate Transitions

Currently, XDP does not implement any mechanism for supporting different Quality of Service (QoS) levels. Specifically, an XDP program receives no back-pressure when attempting to forward packets to a destination that has exhausted its capacity, such as when joining networks with different speeds or other mismatched network characteristics.

While QoS is lacking from XDP, the Linux kernel networking stack features best-in-class Active Queue Management (AQM) and packet scheduling algorithms [23]. Not all of these features are a good fit for the XDP architecture, but we believe that selectively integrating features from the networking stack into XDP is an opportunity to provide excellent support for QoS and AQM in XDP, in a way that can be completely transparent to the packet processing applications themselves. We are planning to explore this further.

6.6 Accelerating Transport Protocols

With XDP we have shown how high-speed packet processing can be integrated cooperatively into the operating system to accelerate processing while making use of existing features of the operating system where it makes sense. XDP focuses on stateless packet processing, but extending the same model to stateful transport protocols such as TCP would provide many of the same performance benefits to applications that require reliable (and thus stateful) transports. Indeed, others have shown that accelerated transport protocols can significantly improve performance relative to the regular operating system stack [5, 25, 35, 52].

One of these previous solutions [52] shows that there is significant potential in improving the raw packet processing performance while keeping the in-kernel TCP stack itself. XDP is a natural fit for this, and there has been some initial discussion of how this could be achieved [21]; while far from trivial, this presents an exciting opportunity for expanding the scope of the XDP system.

6.7 Zero-copy to userspace

As mentioned in Section 3.1, an XDP program can redirect data packets to a special socket opened by a user space application. This can be used to improve performance of network-heavy applications running on the local machine. However, in its initial implementation, this mechanism still involves copying the packet data, which negatively affects performance. There is ongoing work to enable true zero-copy data transfer to user space applications through `AF_XDP` sockets. This places some constrains on the memory handling of the network device, and so requires explicit driver support. The first such support was merged into the kernel in the 4.19 release cycle, and work is ongoing to add it to more drivers. The initial performance numbers look promising, showing transfers of upwards of 20 Mpps to userspace on a single core.

6.8 XDP as a building block

Just as DPDK is used as a low-level building block for higher level packet processing frameworks (e.g., [31]), XDP has the potential to serve as a runtime environment for higher-level applications. In fact, we have already started to see examples of applications and frameworks leveraging XDP. Prominent examples include the Cilium security middle-ware [3], the Suricata network monitor [4], Open vSwitch [49] and the P4-to-XDP compiler project [51]. There is even an effort to add XDP as a low-level driver for DPDK [53].

7 CONCLUSION

We have presented XDP, a system for safely integrating fast programmable packet processing into the operating system kernel. Our evaluation has shown that XDP achieves raw packet processing performance of up to 24 Mpps on a single CPU core.

While this is not quite on par with state of the art kernel bypass-based solutions, we argue that XDP offers other compelling features that more than make up for the performance delta. These features include retaining kernel security and management compatibility; selectively utilising existing kernel stack features as needed; providing a stable programming interface; and complete transparency to applications. In addition, XDP can be dynamically re-programmed without service interruption, and requires neither specialised hardware nor dedicating resources exclusively to packet processing.

We believe that these features make XDP a compelling alternative to all-or-nothing kernel bypass solutions. This belief is supported by XDP's adoption in a variety of real-world applications, some of which we have shown examples of. Furthermore, the XDP system is still evolving, and we have outlined a number of interesting developments which will continue to improve it in the future.

ACKNOWLEDGEMENTS

XDP has been developed by the Linux networking community for a number of years, and the authors would like to thank everyone who has been involved. In particular, Alexei Starovoitov has been instrumental in the development of the eBPF VM and verifier; Jakub Kicinski has been a driving force behind XDP hardware offloading and the bpftool utility; and Björn Töpel and Magnus Karlsson have been leading the AF_XDP and userspace zero-copy efforts.

We also wish to extend our thanks to the anonymous reviewers, and to our shepherd Srinivas Narayana, for their helpful comments.

REFERENCES

[1] David Ahern. 2018. XDP forwarding example. https://elixir.bootlin.com/linux/v4.18-rc1/source/samples/bpf/xdp_fwd_kern.c

[2] Cilium Authors. 2018. BPF and XDP Reference Guide. https://cilium.readthedocs.io/en/latest/bpf/

[3] Cilium Authors. 2018. Cilium software. https://github.com/cilium/cilium

[4] Suricata authors. 2018. Suricata - eBPF and XDP. https://suricata.readthedocs.io/en/latest/capture-hardware/ebpf-xdp.html

[5] Adam Belay, George Prekas, Ana Klimovic, Samuel Grossman, Christos Kozyrakis, and Edouard Bugnion. 2014. IX: A protected dataplane operating system for high throughput and low latency. In *Proceedings of the 11th USENIX Symposium on Operating System Design and Implementation (OSDI '14)*. USENIX.

[6] Gilberto Bertin. 2017. XDP in practice: integrating XDP in our DDoS mitigation pipeline. In *NetDev 2.1 - The Technical Conference on Linux Networking*.

[7] Pat Bosshart, Dan Daly, Glen Gibb, Martin Izzard, Nick McKeown, Jennifer Rexford, Cole Schlesinger, Dan Talayco, Amin Vahdat, George Varghese, et al. 2014. P4: Programming protocol-independent packet processors. *ACM SIGCOMM Computer Communication Review* 44, 3 (2014).

[8] bpftool authors. 2018. bpftool manual. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/bpf/bpftool/Documentation/bpftool.rst

[9] Cisco. 2018. TRex Traffic Generator. https://trex-tgn.cisco.com/

[10] CZ.nic. 2018. BIRD Internet Routing Daemon. https://bird.network.cz/

[11] Luca Deri. 2009. Modern packet capture and analysis: Multi-core, multi-gigabit, and beyond. In *the 11th IFIP/IEEE International Symposium on Integrated Network Management (IM)*.

[12] Mihai Dobrescu, Norbert Egi, Katerina Argyraki, Byung-Gon Chun, Kevin Fall, Gianluca Iannaccone, Allan Knies, Maziar Manesh, and Sylvia Ratnasamy. 2009. RouteBricks: exploiting parallelism to scale software routers. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM.

[13] Linux documentation authors. 2018. Linux Tracing Technologies. https://www.kernel.org/doc/html/latest/trace/index.html

[14] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. Moongen: A scriptable high-speed packet generator. In *Proceedings of the 2015 Internet Measurement Conference*. ACM.

[15] Facebook. 2018. Katran source code repository. https://github.com/facebookincubator/katran

[16] Linux Foundation. 2018. Data Plane Development Kit. https://www.dpdk.org/

[17] The Linux Foundation. 2018. FRRouting. https://frrouting.org/

[18] Sebastian Gallenmüller, Paul Emmerich, Florian Wohlfart, Daniel Raumer, and Georg Carle. 2015. Comparison of Frameworks for High-Performance Packet IO. In *Proceedings of the Eleventh ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '15)*. IEEE Computer Society, 29–38.

[19] Sangjin Han, Keon Jang, KyoungSoo Park, and Sue Moon. 2010. PacketShader: a GPU-accelerated software router. In *ACM SIGCOMM Computer Communication Review*, Vol. 40. ACM.

[20] Sangjin Han, Scott Marshall, Byung-Gon Chun, and Sylvia Ratnasamy. 2012. MegaPipe: A New Programming Interface for Scalable Network I/O. In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '12)*.

[21] Tom Herbert. 2016. Initial thoughts on TXDP. https://www.spinics.net/lists/netdev/msg407537.html

[22] Toke Høiland-Jørgensen, Jesper Dangaard Brouer, Daniel Borkmann, John Fastabend, Tom Herbert, David Ahern, and David Miller. 2018. XDP-paper online appendix. https://github.com/tohojo/xdp-paper

[23] Toke Høiland-Jørgensen, Per Hurtig, and Anna Brunstrom. 2015. The Good, the Bad and the WiFi: Modern AQMs in a residential setting. *Computer Networks* 89 (Oct. 2015).

[24] Solarflare Communications Inc. 2018. OpenOnload. https://www.openonload.org/

[25] EunYoung Jeong, Shinae Woo, Muhammad Asim Jamshed, Haewon Jeong, Sunghwan Ihm, Dongsu Han, and KyoungSoo Park. 2014. mTCP: a Highly Scalable User-level TCP Stack for Multicore Systems.. In *Proceedings of the 11th USENIX Symposium on Networked Systems Design and Implementation (NSDI '14)*, Vol. 14. 489–502.

[26] Rick Jones. 2018. Netperf. Open source benchmarking software. http://www.netperf.org/

[27] Jakub Kicinski and Nic Viljoen. 2016. eBPF/XDP hardware offload to SmartNICs. In *NetDev 1.2 - The Technical Conference on Linux Networking*.

[28] Davide Kirchner, Raihana Ferdous, Renato Lo Cigno, Leonardo Maccari, Massimo Gallo, Diego Perino, and Lorenzo Saino. 2016. Augustus: a CCN router for programmable networks. In *Proceedings of the 3rd ACM Conference on Information-Centric Networking*. ACM.

[29] Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*. IEEE Computer Society.

[30] libbpf authors. 2018. libbpf source code. https://elixir.bootlin.com/linux/v4.18-rc1/source/tools/lib/bpf

[31] Leonardo Linguaglossa, Dario Rossi, Salvatore Pontarelli, Dave Barach, Damjan Marjon, and Pierre Pfister. 2017. *High-speed software data plane via vectorized packet processing*. Technical Report. Telecom ParisTech.

[32] John W Lockwood, Nick McKeown, Greg Watson, Glen Gibb, Paul Hartke, Jad Naous, Ramanan Raghuraman, and Jianying Luo. 2007. NetFPGA–an open platform for gigabit-rate network switching and routing. In *IEEE International Conference on Microelectronic Systems Education*. IEEE.

[33] Rodrigo B Mansilha, Lorenzo Saino, Marinho P Barcellos, Massimo Gallo, Emilio Leonardi, Diego Perino, and Dario Rossi. 2015. Hierarchical content stores in high-speed ICN routers: Emulation and prototype implementation. In *Proceedings of the 2nd ACM Conference on Information-Centric Networking*. ACM.

[34] Tudor Marian, Ki Suh Lee, and Hakim Weatherspoon. 2012. NetSlices: scalable multi-core packet processing in user-space. In *Proceedings of the eighth ACM/IEEE symposium on Architectures for networking and communications systems*. ACM.

[35] Ilias Marinos, Robert NM Watson, and Mark Handley. 2014. Network stack specialization for performance. In *ACM SIGCOMM Computer Communication Review*, Vol. 44. ACM, 175–186.

[36] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. 2014. ClickOS and the art of network function virtualization. In *Proceedings of the 11th USENIX Conference on Networked Systems Design and Implementation*. USENIX Association.

[37] Steven McCanne and Van Jacobson. 1993. The BSD Packet Filter: A New Architecture for User-level Packet Capture. In *USENIX winter*, Vol. 93.

[38] Sebastiano Miano, Matteo Bertrone, Fulvio Risso, Massimo Tumolo, and Mauricio Vásquez Bernal. 2018. Creating Complex Network Service with eBPF: Experience and Lessons Learned. In *IEEE International Conference on High Performance Switching and Routing*.

[39] David S. Miller. 2017. Generic XDP. https://git.kernel.org/torvalds/c/b5cdae3291f7

[40] Robert Morris, Eddie Kohler, John Jannotti, and M Frans Kaashoek. 1999. The Click modular router. *ACM SIGOPS Operating Systems Review* 33, 5 (1999).

[41] Juniper Networks. 2018. Juniper Contrail Virtual Router. https://github.com/Juniper/contrail-vrouter

[42] perf authors. 2018. perf: Linux profiling with performance counters. https://perf.wiki.kernel.org/index.php/Main_Page

[43] Simon Peter, Jialin Li, Irene Zhang, Dan RK Ports, Doug Woos, Arvind Krishnamurthy, Thomas Anderson, and Timothy Roscoe. 2016. Arrakis: The operating system is the control plane. *ACM Transactions on Computer Systems (TOCS)* 33, 4 (2016).

[44] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan J Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. 2015. The Design and Implementation of Open vSwitch. In *Proceedings of the 12th USENIX Symposium on Networked Systems Design and Implementation (NSDI '15)*.

[45] Ntop project. 2018. PF_RING ZC (Zero Copy). https://www.ntop.org/products/packet-capture/pf_ring/pf_ring-zc-zero-copy/

[46] Luigi Rizzo. 2012. Netmap: a novel framework for fast packet I/O. In *21st USENIX Security Symposium (USENIX Security 12)*.

[47] Luigi Rizzo and Giuseppe Lettieri. 2012. Vale, a switched ethernet for virtual machines. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*. ACM.

[48] Pedro M Santiago del Rio, Dario Rossi, Francesco Gringoli, Lorenzo Nava, Luca Salgarelli, and Javier Aracil. 2012. Wire-speed statistical classification of network traffic on commodity hardware. In *Proceedings of the 2012 Internet Measurement Conference*. ACM.

[49] William Tu. 2018. [ovs-dev] AF_XDP support for OVS. https://mail.openvswitch.org/pipermail/ovs-dev/2018-August/351295.html

[50] IO Visor. 2018. BCC BPF Compiler Collection. https://www.iovisor.org/technology/bcc

[51] VMWare. 2018. p4c-xdp. https://github.com/vmware/p4c-xdp

[52] Kenichi Yasukata, Michio Honda, Douglas Santry, and Lars Eggert. 2016. StackMap: Low-Latency Networking with the OS Stack and Dedicated NICs. In *2016 USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 43–56.

[53] Qi Zhang. 2018. [dpdk-dev] PMD driver for AF_XDP. http://mails.dpdk.org/archives/dev/2018-February/091502.html