Towards an OS for the Network Data Plane

Wei Zhang[⋆], Abhigyan Sharma[†], Kaustubh Joshi[†], Timothy Wood[⋆]

[⋆]George Washington University, [†]AT&T Labs Research

Network Function Virtualization (NFV) promises a cloud-computing-like shared platform for packet processing network functions (NFs). Realizing this vision requires a carefully managed packet processing architecture that ensures multiple tenants can safely and efficiently utilize resources. Recent advancements such as user space I/O have significantly improved the throughput (packets/sec) of x86-based packet processing. However, current approaches either rely on VM or container-based isolation between NFs, which incurs high context switch overheads, or run NFs in a shared address space without protection or proper performance guarantees. Our position is that the data plane architecture must play the role of an operating system (OS) for modular NFs run by different tenants, and hence it should provide a number of *OS-like capabilities*, including:

Memory protection: Similar to the abstraction of an OS process, the contents of memory for an NF and a tenant must be protected from others.

Resource allocation: Similar to an OS process scheduler, resource allocation should balance the twin goals of high throughput and fairness among tenants.

State management: Similar to a file system, state management should enable modules to store processing state, e.g., TCP connection state in a stateful NF.

Access control: Similar to the concepts of OS users & permissions, access control should determine the privilege level of tenants and NFs to read or modify architectural components, e.g, processing graph and per-flow state.

FastPaas Architecture

We are developing these capabilities in our data plane architecture called FastPath-as-a-service (FastPaas). FastPaas is the first data plane architecture to protect modules written in a native language. FastPaas implements a multi-threaded modular data plane inside a single process, with one thread dedicated to processing packets on each core (Fig 1). FastPaas maintains a unified packet processing graph, as a DAG, across all tenants, whose nodes correspond to NF modules. The branches in the graph encode packet processing rules that are implemented by FastPaas, e.g., send port 80 traffic to Cache-Node, and remaining traffic to Firewall-Node. The cloud administrator can specify some parts of the graph as unmodifiable by tenants. For example, a node that checks if the incoming packets follow the datacenter's security policy for source IP addresses could be specified as mandatory.

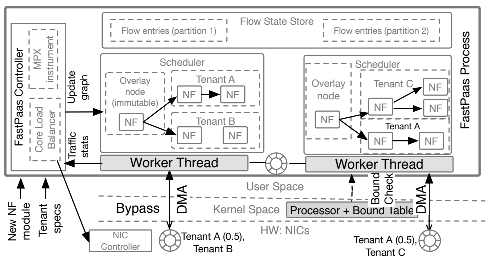


Figure 1: FastPaas Architecture

Resource Allocation: FastPaas must make both coarse grained resource management decisions, e.g., how many cores to allocate for processing traffic from different tenants, and fine grained scheduling decisions, e.g., which batch of packets should be processed next on a given core. In making these decisions, FastPaas scheduler must provide fairness for tenants (weighted by priority) accounting for both packet arrival rates and the computation cost of tenants' service chains. Further, FastPaas must also ensure high utilization of all the cores towards increasing the total throughput.

FastPaas scheduler must reduce fragmentation of a batch of packets due to branches in the processing graph. Fragmentation can diminish benefits of batching by reducing instruction cache hit rates when packets in batch are processed by separate sets of NF modules. To improve performance, FastPaas seeks to leverage advance NIC features (e.g., programmable filtering criteria [1]) for mapping incoming packets to hardware queues that are likely to take the same path in the graph.

State Management & Access Control: FastPaas provides a common state store that NF modules can use. The store is indexed by tenant ID and flow ID. The flow ID is, for example, a hash of the five tuple. This store enables modules to keep per-flow statistics in fields defined by the module it-

DOI: http://dx.doi.org/10.1145/3050220.3060608

self. The data store supports read/write access control lists on flow entries as well as on each field to protect the tenant's and NF's sensitive information. We will explore ways in which a common state store simplifies data management issues, e.g., amortizing flow lookup costs across NFs, and state replication for elastic scaling (e.g., OpenNF [2]) or for fault tolerance (e.g., FTMB [3]).

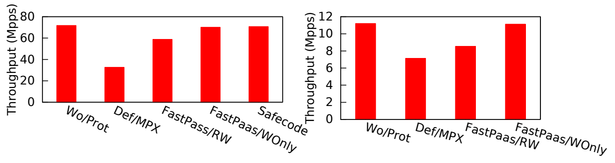
Memory Protection: There are three main approaches for protecting a module's memory contents within a shared process: (1) using a memory-safe language such as Rust, Go, or Java; (2) using a purely software-based memory protection for native languages (C/C++) based on static analysis and code instrumentation; and (3) using hardware-based protection to reduce the overhead of a purely software-based approach. The latter two approaches can protect modules in native languages and hence can make use of the large base of existing NFs written in those languages, and developers that are familiar with those languages.

We have evaluated the overhead of a state-of-the-art hardware protection called MPX introduced in Intel Skylake processors. MPX introduces new registers for storing bounds and instructions for checking the a memory address against stored bounds. Bounds are maintained on a per-pointer basis in an in-memory data structure using compiler instrumentation and are referred to prior to a memory access by a pointer. We found that that MPX incurred a 1.6× overhead in execution time and 1.5× overhead in memory use for a micro benchmark of the malloc module. Interestingly, we find that the primary cause of the overhead is the loading and storing of bounds in registers from the in memory bounds table. The bounds checking operations result in less than 6% overhead for both computation and memory.

The above insight motivates us to design a **coarse-grained hardware memory protection system**. Its key idea is to define a small number of contiguous memory regions that a module is allowed to access during its execution. These regions include a per-module heap, a per-thread stack, and boundaries for packets in the batch being processed. Using a small number of protection boundaries significantly reduces the size of the bounds table and the number of memory accesses to the bounds table. Thus, it reduces memory and computation overhead, but still protects protecting the memory of an NF and the traffic of a tenant. It achieves efficiency by ignoring pointer bounds violations as long as the memory accessed by a pointer is within the coarse-grained boundary that the pointer belongs to.

Evaluation: We present a preliminary evaluation of FastPaas's memory protection by manually instrumenting the NF module code. Our work on automatic source code instrumentation using the LLVM toolchain is ongoing. We test the performance first with a pool of preallocated 64-byte packets on the test machine and then by sending traffic from a separate client machine to the test machine to include the cost of NIC processing. Fig 2 shows the results for a traffic policer (rate limiter) module. We find qualitatively similar findings for other modules (macswap and longest-prefix-match) also.

We observe that the default MPX protection (**Def/MPX**) significantly reduces throughput by up to 46% compared to



(a) Preallocated Pkts (b) Real Pkts

Figure 2: Memory protection schemes.

no protection (**Wo/Prot**). FastPaas achieves better throughput since it eliminates most of the instructions for loading/storing bounds. **FastPaas/RW**, which protects both read and write accesses, achieves a throughput within 24% of the original in both experiments. Since most of the pointer accesses are read-only, **FastPaas/WOnly**, which protects just the writes, achieves an even higher throughput of 96% and 98% of the unprotected module.

FastPaas' overheads appear similar to other protection approaches including a memory safe language (**Rust**) and **Safecode**, which uses static analysis and runtime checks to protect C programs [4]. We found that FastPaas/RW has throughput similar to Rust's version of macswap. Evaluation of Rust for other modules is a topic of our ongoing work. Although Safecode has comparable performance to FastPaas/RW, but it does not protect packet data as it declared by an external memory allocator (DPDK). Unfortunately, we could not test Safecode's performance with real traffic, since the DPDK library reported exceptions when running modules compiled with Safecode. Our initial results suggest that FastPaas provides memory protection while giving additional flexibility of using native language modules and well known data plane libraries such as DPDK.

Conclusions: A multi-tenant NFV platform must strike a careful balance between isolation and efficiency. While prior approaches have required heavy-weight techniques like virtualization to separate tenants, FastPaas leverages new memory protection CPU instructions to provide security at an appropriate granularity with minimal impact on performance. FastPaas allows multiple tenants to be safely deployed within a shared address space, with isolation, scheduling, state management, and access controls all provided by the framework instead of the underlying OS. We are continuing to develop FastPaas and explore how it can provide prioritized resource allocations, simplify state management, and enforce appropriate security policies. This work was supported in part by NSF Grant CNS-177651.

References

[1] Antoine Kaufmann and et al. High Performance Packet Processing with FlexNIC. In *ASPLOS*, 2016.

[2] A Gember-Jacobson and et al. OpenNF: Enabling innovation in network function control. *SIGCOMM*, 2014.

[3] J Sherry and et al. Rollback-recovery for middleboxes. In *SIGCOMM*, 2015.

[4] Dinakar Dhurjati and et al. SAFECode: Enforcing alias analysis for weakly typed languages. In *PLDI*, 2006.