Hardware-assisted Isolation in a Multi-tenant Function-based Dataplane

Wei Zhang[⋆], Abhigyan Sharma[†], Kaustubh Joshi[†], Timothy Wood[⋆]
[⋆]George Washington University, [†]AT&T Labs Research

ABSTRACT

Existing software dataplanes that run network functions inside VMs or containers can provide either performance (by dedicating CPU cores) or multiplexing (by context switching), but not both at once. Function-based dataplane architectures by replacing VMs and containers with function calls promise to achieve multiplexing and performance at the same time. However, they compromise memory isolation between tenants by forcing them to use a shared memory address space.

In this paper, we show that an operating system-like management layer for modules in a function-based data plane can offer OS-like constructs such as performance and memory isolation. To provide memory isolation, we leverage new Intel CPU extensions (MPX) to create coarse-grained heap and stack protection even for legacy code written in unsafe native languages such as C. In addition, we use programmable NIC offloads to distribute load across cores as well as to prevent batch fragmentation when processing complex service graphs. Our preliminary evaluation shows the limitations of existing techniques that require heavy weight memory isolation or incur cross-core overheads.

CCS CONCEPTS

• **Networks → In-network processing**; **Network management**;

KEYWORDS

Network Function, Memory Isolation, Performance Isolation

1 INTRODUCTION

Large tier-1 network providers are moving the dataplane of their network functions (NFs) to software running on cloud platforms [5]. Despite advances in software dataplanes [4], their performance drops significantly when sharing a single core among multiple NF processes or when processing a packet through a service chain of NFs across multiple cores. Due to these overheads of context switching and cross core operation, NF deployments in VMs and containers fall short of the performance promised by the underlying hardware.

Function-based dataplanes such as NetBricks[15], Fd.io VPP [6] and VMWare NSX [18] signficantly reduce these overheads. These architectures implement NFs as modules and invoke them via function calls. They avoid cross-core overheads by processing a packet through a service chain in a single thread on the same core. Due to nominal overheads of switching among NFs on a core, they can multiplex a core in a fine-grained manner among NFs of multiple tenants.

As a use case for a multi-tenant function-based dataplane, consider a provider edge (PE) router for MPLS services. A physical PE router has ports to connect to a few hundred customers. A recent software-based PE router proposal, Edgeplex [2], runs each customer's router in a separate VM and could consume up to a few hundred cores to replace a physical router. In comparison, a function-based data plane would only consume cores proportional to the aggregate traffic at the router, making it cost-effective for a network provider.

A function-based dataplane, despite its advantages, faces two key challenges in supporting multi-tenancy:

- Memory isolation: The contents of memory for each NF and each tenant must be protected from others. Memory isolation is critical to ensure correctness of NFs and the security of tenants' traffic.
- Performance isolation: Flexible scheduling policies should determine the share of CPU resources a tenant's traffic receives on a multi-core server. These policies are critical to achieving resource fairness among tenants while optimizing global metrics such as aggregate throughput.

Our proposal FastPaas (FastPath-as-a-service) is, to our knowledge, the first software dataplane to adopt a hardware-based technique (Intel MPX) for memory isolation. In principle, MPX can protect modules written in any language by instrumenting the compiler toolchain. We demonstrate its capabilities by protecting modules written in a dominant

SOSR '18, March 28–29, 2018, Los Angeles, CA, USA

ACM ISBN 978-1-4503-5664-0/18/03…$15.00
https://doi.org/10.1145/3185467.3185493

and non-memory safe C language. In our preliminary work towards building FastPaas we:

- Outline the challenges in processing a packet entirely in a run to completion manner on a single core, particularly when the packet must traverse a service chain or when traffic must be split between NFs of many tenants.
- Evaluate the overhead of isolating NF modules using fine-grained MPX instrumentation done by compilers today.
- Propose a coarse-grained hardware-based memory isolation approach for NF modules written in C and show that it significantly lowers the overhead over existing compilers.

Our preliminary FastPaas prototype reveals that the run to completion model can double throughput while using fewer resources, and that our coarse grained approach lowers the cost of memory isolation from over 64% to 24% or less.

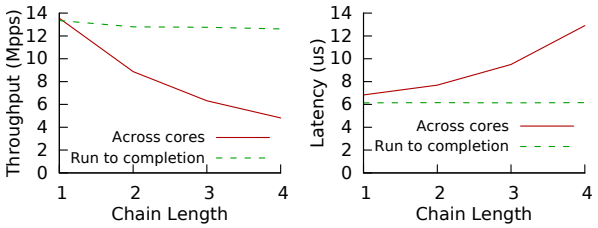
2 WHY RUN-TO-COMPLETION IS HARD

We refer to a run-to-completion model in which a packet is processed entirely in a single thread on the same CPU core. Here we highlight the overheads of context-switching among processes and cross core communication, while also recognizing the challenge of achieving good performance when run-to-completion is used in a multi-tenant environment.

Cross-core processing costs: We first compare the single process, run-to-completion model to a more modular approach that deploys NFs as separate processes across cores. We evaluate chains of one to four NFs, all running a Longest Prefix Match (LPM) function. We consider two setups: in the run-to-completion case, the NIC uses RSS to assign packets to a core, where a single thread processes the full service chain. In the across-cores case, packets first arrive at a management thread which acts as a software switch that delivers the packets to the first NF in the chain running as a separate process on a different core; that NF gives packets to the next process on a different core, etc. In both cases, each CPU core runs a single thread to prevent context switch overheads and shared memory is used for zero copy packet movement.

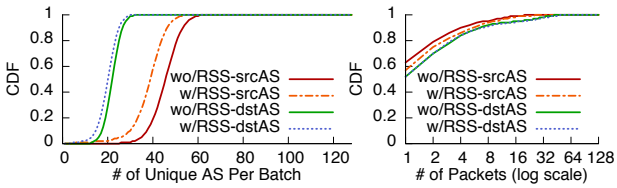
Figures 1(a) and 1(b) show that as the chain length rises, the performance gap between the run-to-completion and across cores rises. This is due to the high cost of moving packets between NFs on different cores—even with zero-copy packet transfer, a performance gap still arises because the management thread moving packets between NFs becomes a bottleneck. Resolving this issue would require dedicating more CPU cores than the run-to-completion approach.

A run-to-completion, function-based NF architecture eliminates context switches and allows more NFs to be put on the same core, avoiding cross-core processing overheads. However, function-based NFs are in the same process space and lack memory isolation, meaning one NF can easily access another NF's data. In multi-tenant environments this will



(a) Throughput (b) Latency

Figure 1: Run to completion eliminates the need to move packet pointers across cores.



(a) AS per Batch (b) Packets per AS

Figure 2: Trace analysis illustrates the difficulty of batching packets when each AS has a different chain.

bring serious security concerns. In FastPaas, we are designing coarse-grained isolation techniques that leverage new CPU hardware extensions to provide fast memory isolation.

Multi-Tenant Batching: Run-to-completion frameworks such as Fd.io's Vector Packet Processing (VPP) tool achieve extremely high performance by maximizing data and instruction cache hit rates through careful batching. These approaches assume that all packets in a batch will be processed with the same function pipeline, allowing various optimizations such as vector operations. However, in multi-tenant environments an incoming packet stream may need to be demultiplexed by tenant, producing inefficient, fragmented batches that each require a unique set of functions.

To evaluate how much fragmentation might occur in a realistic scenario, we consider a packet trace from the CAIDA Equinix-Chicago collection monitor [1]. We analyze a one hour trace from 01/21/16 and determine the source and destination Autonomous System (AS) for each packet. We use each AS number to represent a different tenant in the NFV platform, mimicking the case where a network operator might have different NFV service chains being applied based on either the source or destination ISP of each traffic flow. Figure 2(a) shows a CDF of the number of unique source or destination AS observed in each 128 packet batch in the trace. There is a greater diversity of source AS. However, in either case there is a relatively large number of unique AS, meaning that each chain only receives a small number of packets per batch read from the NIC as shown in Figure 2(b). One simple approach commonly used to help group packets

from the same flow is Receive Side Scaling. But as shown in the "w/RSS" lines, which mimics dividing packets into 8 RSS queues by 5-tuple hash, this has minimal impact on the number of packets per AS in each batch, since splitting by flow does not directly correlate to splitting by AS.

This observation further reinforces the efficiency benefits gained from running a large number of NFs within a single process—since batch sizes will typically be small, paying a high context switch cost (if using multiple processes) or a data copying cost (if spread over multiple cores) on each batch will result in unacceptable overhead. However, it also suggests that a multi-tenant platform would benefit from carefully demultiplexing and batching packets prior to sending them through a service chain. We will explore how new features in programmable NICs can allow for efficient, customizable demultiplexing in hardware [11]. We believe this will help increase cache hit rates without requiring a software demultiplexer before each tenant's pipeline.

3 FASTPAAS ARCHITECTURE

Our goal is to provide two types of isolation to tenants.

First is **performance isolation**. The function-based dataplane must be able to control the scheduling of NFs to meet the performance goals of each tenant. Fortunately, the run-to-completion model can help with this problem since the management framework in the process can carefully select which NF function to call next based on tenant defined SLAs. In contrast, approaches that employ multiple processes running on each core (e.g., running one process per tenant), are more difficult to manage since scheduling is done by the OS kernel, which is not NF-aware. In this section, we describe the FastPaas architecture and how its structure facilitates scheduling NFs to meet performance goals.

Second is **memory isolation**. Run-to-completion implies that processing occurs as a series of function calls within one process' address space, so if done naively, different NFs can trivially observe or corrupt the memory state of other NFs. This applies to both packets, which may hold sensitive data, and to each NF's internal data structures. In Section 4, we describe a memory protection scheme leveraging Intel MPX hardware instructions to provide data integrity (protecting against rogue NF writes) or confidentiality (protecting against both reads and writes). This allows trade-offs in performance and isolation requirements.

3.1 Multi-tenant performance isolation

Figure 3 presents the FastPaas architecture. The FastPaas system being developed consists of a single process running on a physical machine, with multiple worker threads on dedicated cores. Network functions are composed as directed acyclic graphs (DAGs). The FastPaas *controller* configures

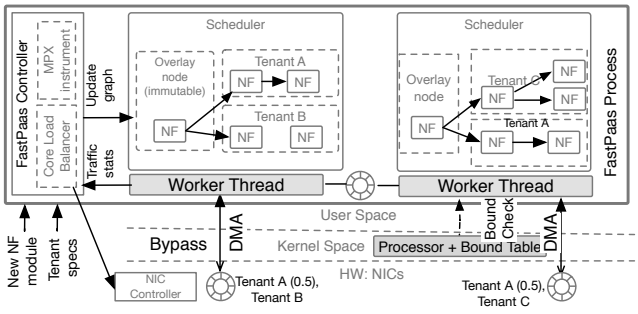


Figure 3: FastPaas Architecture

the NICs on the machine to filter incoming traffic to separate hardware queues based on the service graph they belong to. Each thread polls one or more NIC queues for packets, and the thread's *batch scheduler* decides which queue to process next. The batch of packets read from the selected queue is processed in a run-to-completion manner.

FastPaas's controller makes coarse grained resource management decisions, e.g., how many cores to allocate for processing traffic from different tenants, and its batch scheduler makes fine grained scheduling decisions, e.g., which batch of packets should be processed next on a given core. Most prior work in this area has focused on simple packet schedulers, where there may be different packet queues to decide from, but the processing cost of any packet is roughly identical [17]. In FastPaas, processing costs may vary depending on the nature and length of the processing graph a packet needs to traverse. Thus, FastPaas must provide fairness for tenants (weighted by priority) accounting for both packet arrival rates and the computation cost of tenants' service chains. FastPaas also needs to implement signaling mechanisms to detect and pre-empt modules to deal with buggy/unfriendly modules hogging CPU cycles.

Our design will alleviate the problem of batch fragmentation by leveraging NIC support for mapping incoming packets to queues. The branches in the FastPaas DAG are annotated with the filtering criteria that determine which path will be taken for a given packet. While sometimes this criteria requires complex module-specific processing to decide, it is often a simple rule based on header information like the destination IP or port. Current generation NICs support filtering packets into queues based on this type of information, and future NICs are likely to have growing support for programmable filtering criteria [11]. Thus FastPaas will parse the DAG to determine a set of filters that can be installed in HW to preemptively separate packets into queues that correspond to the paths they are likely to take through the graph, reducing the need to split batches at branch points.

FastPaas's controller assigns to each thread one or more NIC queues to poll for packets; the next NIC queue to read packets from is determined by a thread's batch scheduler. To

ensure fair scheduling, FastPaas tracks the CPU consumption of each tenant. Function-based data planes such as BESS [7] provide primitives to instrument the processing pipeline for collecting the processing cost of individual modules as well as for a service chain. Our current implementation uses a simple weighted-round robin scheduling among queues. The weight of a queue is statically defined based on the priority of a tenant. Further study of scheduling strategies (including strict priority classes) is ongoing.

4 MEMORY PROTECTION

FastPaas is the first dataplane architecture to enable support for memory protection of modules written in a native language. It achieves this through a coarse-grained, hardware-assisted approach that achieves low overhead. In this section, we describe the threat model, explain the limitations of existing protection approaches, describe our memory protection approach, and present some preliminary evaluation based on a manual instrumentation of network function modules.

Threat Model: We assume an honest but curious environment where tenants running NFs want the overall system to operate correctly (i.e., will not attack its availability), but may attempt to snoop on or manipulate the traffic or state of other tenants. Thus we focus on providing both secrecy and integrity to a tenant's NFs, including NF code and functions, packets, NF internal data, and flow state.

4.1 Protection techniques & limitations

We classify existing approaches to memory protection into three categories as described below.

Writing modules in a **memory safe language** such as Go or Rust, which disallows pointer arithmetic and automatically manages memory, is a known protection approach. Recently, NetBricks [15] has proposed the use of such languages in implementing a network dataplane. However, many NFs have been developed in C/C++ and have a large code base (e.g., Bro: 398,796 and Snort: 409,065 lines of C code). Rewriting them from scratch with Rust will take lots of effort. If we can provide memory isolation that is transparent to legacy NFs, it will benefit both NFV providers and NF developers.

Memory protection for native languages (C/C++) has seen a lot of work in the programming languages community [3, 13]. These techniques commonly use static analysis and code instrumentation to detect unsafe memory access at compile time or at runtime. These techniques either do not provide complete protection [3] and/or incur a high computation and memory overhead. For example, the SoftBound approach slows SPEC benchmark applications by 67% on average, and by over 150% in the worst case [13].

Hardware-based memory protection has recently been introduced in Intel Skylake processors [9]. In essence, MPX implements an optimized form of SoftBound [13], that takes advantage of additional registers and bound checking instructions to improve performance [14]. Each process has a bound directory and a bound table that stores the lower and upper bound of valid memory addresses for each pointer. A process registers a signal handler with a configuration register at startup. Upon a memory access by a pointer, the address being accessed is compared against the lower and upper bounds retrieved from the bound table. If the address falls outside these bounds, the CPU traps into kernel mode and sends a SIGSEGV signal to trigger the signal handler function, to notify of the invalid memory access.

	Default MPX	Remove load/store	Remove comparison
Execution Time	1.59	1.06	1.54
Memory Usage	1.54	1.01	1.53

Table 1: Performance and memory overhead of Intel MPX relative to no memory protection.

In typical usage, an MPX-enabled compiler automatically sets narrow bounds around each pointer access. However, this fine-grained protection causes both performance and correctness issues. First, frequently loading and checking bounds can add high overhead as we discuss in the next section. Second, some common C idioms leave compiler ambiguity that prevent correct MPX bounds from being set. For example, the compiler cannot correctly set bounds for dynamically sized arrays, nor does it properly handle internal pointers in structs [14]. Both of these are common in NFV software, since packet buffer arrays may be dynamically sized and packet meta data structs may contain internal pointers. This motivates us to adapt MPX so it uses coarser grained bounds which will be both quicker to validate any will avoid bounds being incorrectly set in idiomatic code.

We evaluate the overhead of protection using Intel MPX using compilers that support the necessary code instrumentation. We measure the overhead of MPX by running a representative application, HashSet, that we compile with GCC. We find that the use of MPX inflates execution time by 2.6× and the memory use by 3.13× over the baseline, non-MPX version of the application. To further understand this cost we consider a simple memory allocation microbenchmark and analyze the impact of the different MPX operations. Table 1 shows that removing the MPX instructions that load/store pointer address bounds in registers from/to the bounds table stored in DRAM eliminates almost all overheads. In contrast, the instructions for comparing memory addresses against bounds stored in registers are inexpensive since they involve simple arithmetic operations.

This result gives us a key insight: most of the overhead of MPX comes from loading/storing bounds for every pointer, so we propose a new approach that achieves low overhead by applying memory protection at coarser granularity.

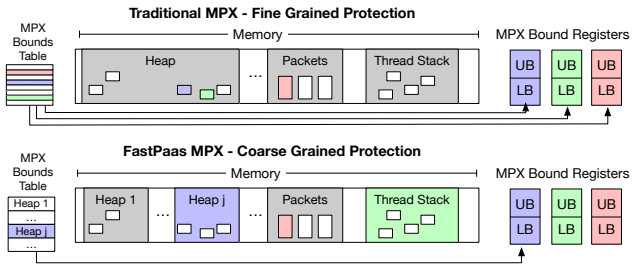


Figure 4: FastPaas results in a significantly smaller bounds table and fewer memory accesses.

4.2 Coarse-grained hardware protection

The key idea behind coarse-grained protection is to define a small number of contiguous memory regions that a module is allowed to access during its execution as shown in Figure 4. These regions include a per-module heap, a per-thread stack, and the boundary for the packet 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, thereby reducing memory and computation overhead. Our approach ensures the goal of protecting the memory of a 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.

Usage: NF modules are submitted to FastPaas in source code form written in C, which are compiled and instrumented with Intel MPX instructions for implementing our coarse-grained protection. FastPaas assumes that only the NF module is untrusted and hence instruments only its code. It can protect both read and writes or just memory writes depending on the protection needed for a module. As expected, read/write protection has a higher overhead than write-only protection. The compiled module is linked using a modified version of glibc which disables some system calls, e.g., fork and exit, that a module is not allowed to make.

Instrumentation technique: We provide a sketch of the technique for instrumenting modules for coarse-grained protection. The instrumentation requires a compiler to make three main decisions: (1) classifying a pointer into one of three types: heap, stack, or packet, (2) calculating address bounds for each pointer type and (3) determining locations for inserting instructions for storing and checking bounds.

Pointer type classification: A stack pointer is created by taking the address of a local variable. A heap pointer is created as a result of a malloc call. Global and static variable declarations are re-written as malloc operations so that they are allocated within the per-module heap. A packet pointer is created by taking the address of any packet (or its fields) in the current batch being processed. Pointers created by pointer arithmetic or pointer assignment are treated as the same type as the original pointer, e.g., array accesses and structure field accesses. Pointer type information is propagated with function calls made during a module's execution to infer pointer types inside the called function [13].

Pointer bounds calculation: Bounds must be calculated for each type of pointer, not each object that is pointed at. A heap pointer's lower and upper bounds are obtained from the current memory slab allocated to this module by the per-module memory allocator. A packet's lower bound is its starting address in the memory and its upper bound is the end address of the packet's slot in the memory pool for packets, e.g., DPDK creates a slot of constant size 2KB per packet. Stack upper bound (assuming decreasing stack addresses) is the top of the stack prior to calling an NF module. To compute a lower bound, we assume that a stack of constant size is allocated to a module. The stack lower bound is obtained by subtracting the constant from the stack upper bound.

Instrumentation location: Bounds checking instructions are inserted before a pointer dereference of any type. The instrumentation for storing bounds in bounds registers depends on the type of the pointer. Before a module is scheduled for execution, its bounds for stack and heap are saved in two of the bounds registers. Modules typically process packets one by one, so a packet's bounds are computed and stored before the first memory access. A straightforward optimization is to not store the bounds of a pointer type at all if no memory accesses by pointers of that type are found. If a given pointer cannot be fully disambiguated, i.e., it could point to either the stack or heap depending on run-time behavior, then multiple bounds checks can be inserted to ensure the pointer is within one of those regions. Our current implementation requires manual addition of the above code, but we are investigating ways to automate this analysis in LLVM.

4.3 Performance comparison

Our experiments evaluate FastPaas's protection by manually instrumenting the NF module code as described above.

Schemes: We evaluate **FastPaas/RW**, which protects both read and write memory accesses and **FastPaas/WOnly**, which protects just the writes. **Wo/Prot** does not protect invalid memory access. **Def/MPX** protects memory using fine-grained MPX instrumentation done by GCC (v5.4). **Rust** implements modules in the memory safe language Rust. We also evaluated **Safecode**, which uses static analysis and runtime checks to protect C programs [3].

Experiment setup: Our test server is a desktop with Intel Skylake processor (Intel(R) Core(TM) i7-6700) that supports MPX instructions, an Intel X710 10GbE SFP+ NIC, and Ubuntu 16.04. We use MoonGen as the packet generator.

We first test the performance with a pool of preallocated packets on the test machine, which ensures that processing at the NIC does not become a bottleneck. We evaluate two NFs

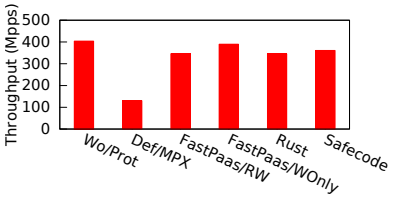


Figure 5: Macswap Preallocated Pkts

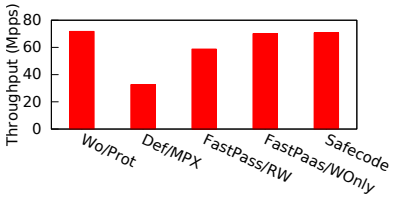


Figure 6: Policer Preallocated Pkts

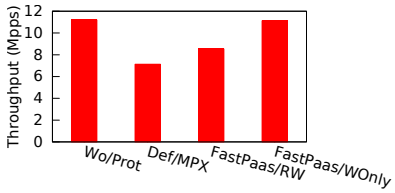


Figure 7: Policer Real Pkts

with different processing costs per packet: macswap (less costly) and traffic policer (more costly). Figure 5 shows that default MPX protection reduces macswap's throughput to only 32% of the throughput achieved by the unprotected module. For the policer module (Figure 6), the throughput drops to 46% of the original. FastPaas's coarse grained protection achieves better throughput since it eliminates most of the instructions for loading/storing bounds. FastPaas/RW achieve a throughput of 86% for macswap and 82% for the traffic policer compared to the unprotected module. FastPaas/WOnly acheves an even higher throughput of 96% and 98% of the unprotected module. FastPaas/RW has similar throughput with Rust's version of macswap, suggesting that protecting modules using a memory safe language does not provide additional performance benefits over protecting modules in a native language with FastPaas. Comparing Rust with FastPaas for policer and other modules is a topic of our ongoing work. Although Safecode [3] has comparable performance to FastPaas/RW, it does not protect packet data as it declared by an external memory allocator (DPDK).

Next, we send traffic (64 byte packets) from a separate client machine to the test machine to also include the cost of processing at the NIC. For macswap, every scheme (even the MPX/def) is able to saturate the 10 Gbps link. The implication is that if packet processing is inexpensive, memory protection may be achieved with almost no overhead. For traffic policer (Figure 7), the performance follows the same trend as the above experiment. FastPaas/RW achieves 76% of the throughput of an unprotected module. FastPaas/WOnly gets almost the same performance as an unprotected module. This is because most of the pointer accesses in this module are read-only. Unfortunately, we could not test Safecode's performance with real traffic, since the DPDK library reported exceptions in running modules compiled with Safecode. While these experiments run only a single NF at a time, we expect similar performance when running NF chains or NFs from different tenants. Our implementation is not optimized for the single NF case, e.g., we still reload the MPX bound registers at every iteration even though we only consider one tenant. Overall, our preliminary results show that coarse-grained protection reduces overhead compared to the standard fine-grained MPX protection.

5 RELATED WORK

Memory isolation: AddressSanitizer [16] only seeks to detect invalid addresses for a process, and does not isolate valid addresses on a per-tenant or a per-module basis inside a process. SoftFlow [10] is a OpenFlow-based dataplane that supports a run-to-completion model, however it does not provide any memory isolation for multi tenancy. NetBricks [15] leverages memory safe language for isolation, for which it needs to rewrite an existing NF developed in C language. Light-Weight Contexts [12] take microseconds for context switches, while a core needs to process a packet in as few as tens of nanoseconds to support line-rate performance.

Scheduling: Deficit round-robin (DRR) [17] only equalizes bytes per flow, it does not balance the CPU usage in processing those flows. While FlexNIC provides mechanisms to perform application-specific load balancing across cores [11], FastPaas seeks to automatically define load balancing policies based on packet processing graph specifications and traffic measurements. DPDK QoS [4] and BESS [8] rely on hierarchical schedulers to limit the resource usage but do require manual configuration of these schedulers. Our ongoing work will study automated scheduling and packet batching strategies to improve throughput.

6 CONCLUSION

A multi-tenant function-based data plane must strike a careful balance between isolation and efficiency. Instead of heavyweight 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. It proposes the use of advanced packet filtering in NICs to support batching and core load balancing. FastPaas allows multiple tenants to be safely deployed within a shared address space, with performance and memory isolation provided by the framework instead of the underlying OS. We are continuing to develop FastPaas and explore how it can provide prioritized resource allocations, and enforce appropriate security policies.

Acknowledgments: This work was supported in part by NSF grants CNS-1422362 and CNS-1525992. We would like to thank the anonymous reviewers and our shepherd, Michio Honda, for their comments and feedback.

REFERENCES

[1] 2017. CAIDA Passive Monitor: equinix-chicago. (2017). http://www.caida.org/data/monitors/ http://www.caida.org/data/monitors/.

[2] Angela Chiu, Vijay Gopalakrishnan, Bo Han, Murad Kablan, Oliver Spatscheck, Chengwei Wang, and Yang Xu. 2015. EdgePlex: Decomposing the Provider Edge for Flexibilty and Reliability. In *SOSR*.

[3] D. Dhurjati, S. Kowshik, and V. Adve. 2006. SAFECode: enforcing alias analysis for weakly typed languages. In *PLDI*.

[4] DPDK. 2018. DPDK: Data Plane Development Kit. (2018).

[5] ETSI. 2016. NFV. (2016). http://www.etsi.org/.

[6] FD.io. 2016. VPP. https://fd.io/technology. (2016).

[7] S. Han and et al. 2017. BESS: Berkeley Extensible Software Switch. https://github.com/NetSys/bess. (2017).

[8] S. Han, K. Jang, A. Panda, S. Palkarand D. Han, and S. Ratnasamy. 2015. *SoftNIC: A Software NIC to Augment Hardware.* Technical Report. http://www.eecs.berkeley.edu/Pubs/TechRpts/2015/EECS-2015-155.html

[9] Dave Hansen. 2016. Intel MPX for Linux. https://01.org/blogs/2016/intel-mpx-linux. (2016).

[10] Ethan J. Jackson, Melvin Walls, Aurojit Panda, Justin Pettit, Ben Pfaff, Jarno Rajahalme, Teemu Koponen, and Scott Shenker. 2016. SoftFlow: A Middlebox Architecture for Open vSwitch. In *USENIX ATC*.

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

[12] James Litton, Anjo Vahldiek-Oberwagner, Eslam Elnikety, Deepak Garg, Bobby Bhattacharjee, and Peter Druschel. 2016. Light-Weight Contexts: An OS Abstraction for Safety and Performance.. In *OSDI*.

[13] S. Nagarakatte, J. Zhao, M. Martin, and S. Zdancewic. 2009. SoftBound: Highly compatible and complete spatial memory safety for C. *ACM Sigplan Notices* (2009).

[14] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. 2017. Intel MPX Explained: An Empirical Study of Intel MPX and Software-based Bounds Checking Approaches. *CoRR* abs/1702.00719 (2017). arXiv:1702.00719 http://arxiv.org/abs/1702.00719

[15] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker. 2016. NetBricks: Taking the V out of NFV. In *OSDI*.

[16] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. 2012. AddressSanitizer: a fast address sanity checker. In *USENIX ATC*.

[17] M. Shreedhar and G. Varghese. 1996. Efficient fair queuing using deficit round-robin. *IEEE/ACM TON* (1996).

[18] VMWare. 2017. VMware NSX. https://code.vmware.com/nsx-for-vsphere/nsx-components. (2017).