# SmartNIC Security Isolation in the Cloud with S-NIC

Yang Zhou
Harvard University

Mark Wilkening*
Harvard University

James Mickens
Harvard University

Minlan Yu
Harvard University

## Abstract

Modern smart NICs provide little isolation between the network functions belonging to different tenants. These NICs also do not protect network functions from the datacenter-provided management OS which runs on the smart NIC. We describe concrete attacks which allow a network function's state to leak to (or be modified by) another network function or the management OS. We then introduce S-NIC, a new hardware design for smart NICs that provides strong isolation guarantees. S-NIC pervasively virtualizes hardware accelerators, enforces single-owner semantics for each line in on-NIC cache and RAM, and provides dedicated bus bandwidth for each network function. Using this design, we eliminate side channels involving shared hardware state, and give each network function the illusion of having a private smart NIC. We show how these virtual NICs can be integrated with preexisting datacenter technologies for virtual LANs and trusted host-level computations like SGX enclaves. The overall result is that S-NIC enables strongly-isolated, NIC-accelerated datacenter applications; in these applications, network functions and host-level code receive hardware-guaranteed isolation from other applications and the datacenter provider.

***CCS Concepts:*** **• Security and privacy → Tamper-proof and tamper-resistant designs**; **• Hardware → Networking hardware**; **• Networks → Middle boxes / network appliances**.

***Keywords:*** Trusted execution environment, Network functions, Smart NICs

**ACM Reference Format:**
Yang Zhou, Mark Wilkening, James Mickens, and Minlan Yu. 2024. SmartNIC Security Isolation in the Cloud with S-NIC. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 19 pages. https://doi.org/10.1145/3627703.3650071

*Now at AMD Research and Advanced Development.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
*EuroSys '24, April 22–25, 2024, Athens, Greece*
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-0437-6/24/04…$15.00
https://doi.org/10.1145/3627703.3650071

## 1 Introduction

In a modern distributed system, network functions (i.e., pieces of code which manipulate packets) are critical [41, 60, 65, 69, 70, 77, 78]. Some network functions are simple, like packet compressors, but others are complex, stateful applications like WAN optimizers, NATs, intrusion detection systems, and split-browser web proxies. When a distributed service is deployed, a developer (or a function-as-a-service company [8, 88, 117, 127]) must push functions in addition to web servers, database engines, and other software components.

Traditionally, datacenter tenants execute functions inside of virtual machines, relying on the hypervisor to isolate a function from other VMs [54, 80, 100, 124]. However, a function's code and data are still accessible to the hypervisor itself; this is unattractive if tenants do not trust the datacenter operator, e.g., because functions manipulate keys for encrypted traffic that should be hidden from the datacenter operator [64, 98, 108]. Running a function as an SGX enclave within a VM [98] offers weak protection against hypervisor snooping due to well-known side channel vulnerabilities in SGX hardware [18, 21, 46, 118].

Recently, "smart" NICs [41, 76, 78, 97] have introduced an alternative deployment model. A smart NIC uses a system-on-a-chip design, aggregating general-purpose cores as well as hardware accelerators that optimize common networking tasks like TCP checksumming. A server can offload a function to a smart NIC, allowing the function to compute on packets directly, via cores and accelerators on the same die as NIC packet buffers. Smart NICs typically contain RISC cores that have lower capital costs and power consumptions than x86 server cores. Thus, offloading functions from server cores to a smart NIC can greatly reduce a datacenter provider's total-cost-of-ownership (TCO) [68]. For many functions, offloading also improves performance [60, 69, 70, 77]. For example, a function might contain a task that runs faster on a NIC accelerator than a host-level x86 core. Offloading can also improve function performance by avoiding PCIe latencies that would otherwise be incurred by transferring packets between on-NIC RAM and host RAM. Because of these cost and performance advantages, major datacenter operators

like Microsoft and Baidu have already started to migrate functions to smart NICs [45, 68]. Application developers have also started to offload a variety of work to smart NICs, including data caching [77], transaction ordering [69], and distributed consensus protocols [60, 70].

Unfortunately, commodity smart NICs provide weak isolation between functions. These NICs also enforce weak isolation between a function and the datacenter operator. The reason is that on-NIC RAM has few access controls, and low-level hardware resources like checksum accelerators are not virtualized. These deficits hurt function robustness and security (§3), making multi-tenant occupation of a single NIC unsafe. Even in single-tenant scenarios (where all functions and privileged software on a NIC belong to the datacenter provider), buggy or subverted code anywhere is a threat to all other software on the NIC. These problems cannot be solved by a trivial application of standard isolation approaches (e.g., extended page tables [14] and SR-IOV [56]); those approaches do not prevent side channels, do not have full (or any) enlightenment about the specific hardware affordances of smart NICs (e.g., on-die IO buffers and workload accelerators), and do not protect tenants from privileged software. Prior work on *performance* isolation between smart NIC functions [47, 73] does not directly address *side channel* isolation, and thus is also insufficient for providing comprehensive non-interference guarantees. The goal of this paper is to show that, with minimal changes to smart NIC hardware, datacenters can provide offloaded functions with strong isolation, while preserving many of the TCO reductions and performance boosts that offloading has traditionally provided. We introduce a new smart NIC design, called S-NIC, that has three important features.

**Aggressive disaggregation of internal resources:** In recent years, datacenter operators have connected compute servers and storage servers via full-bisection networks [49, 85]; these networks enable locality-oblivious assignment of datacenter resources to datacenter tenants [83, 90]. S-NIC uses a similar principle within a NIC, allowing a network function to be assigned to a set of programmable cores, TX/RX queues, hardware accelerators, and DRAM regions. These components are stitched together using trusted bus management hardware that provides reserved memory bandwidth for the function.

**Side-channel-free virtualization of NIC resources:** S-NIC uses the abstraction of a *virtual smart NIC* to expose a collection of physical NIC resources. S-NIC virtualizes each resource in a way that is free of side channels; S-NIC also provides traditional notions of confidentiality and integrity for network function state. For example, consider the physical RAM that belongs to a network function. S-NIC uses per-core, hardware-controlled memory denylists (§4.2) to ensure that other network functions (and even the on-NIC OS) cannot read or write those pages. S-NIC also reserves L1/L2/L3 cache space for the function in a way that eliminates cache-based side channels.

**Integration with preexisting datacenter management technologies:** S-NIC allows a network function to act as a VXLAN endpoint [55]; in this manner, a function can integrate directly with the (virtual) Layer 2 datacenter topology that is owned by a tenant [55]. S-NIC also allows functions to remotely attest their state [95], enabling remote endpoints to trust a function to act a TLS middlebox [86, 105] or perform other sensitive operations. We show that, by stitching together a set of S-NIC functions and SGX enclaves, a tenant can build a high-performance, strongly-isolated distributed system that is resilient to malicious datacenter operators.

In summary, this paper provides three contributions:

- We empirically demonstrate that commodity smart NICs provide weak isolation that allows functions to corrupt each other's packets, steal computational state, and launch denial-of-service attacks on shared hardware resources.
- To prevent these exploits, we introduce new hardware-level isolation mechanisms for smart NICs (§4). A key research challenge was determining how to pervasively virtualize on-NIC resources without invasive microarchitectural changes.
- Using extensive hardware-level simulations, we evaluate S-NIC, showing that our isolation mechanisms decrease function throughput by less than 1.7%. Enforcing isolation requires only modest amounts of new silicon: chip area increases by up to 8.89%, and power draw increases by up to 11.45%. Overall, our analysis demonstrates that S-NIC preserves 91.6% of the TCO benefit that function offloading typically provides.

S-NIC is the first smart NIC to provide safe, efficient multi-tenancy for network functions and management software that distrust each other.

## 2 Threat Model

We consider eight types of principals. **Tenants** want to run **network functions** atop **S-NIC hardware**. The NIC (and the attached **host machine**) are owned by a **datacenter operator**. The host machine runs a **host OS** and other **unprivileged host software**. The NIC runs a **NIC OS**. Both the host OS and the NIC OS are provided by the datacenter operator.

S-NIC's goal is to isolate a function from all host-level software and all NIC-level software (i.e., the NIC OS and other functions). Isolation prevents external software from directly tampering with function state, or indirectly observing it through side channels induced by NIC-level co-tenancy. By eliminating side channel attacks on the NIC's microarchitecture, we also provide performance isolation between co-located functions. Network-observer side channels caused

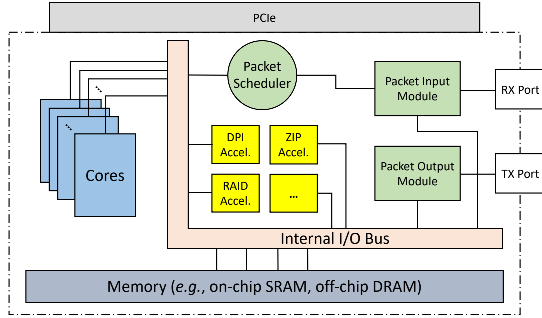



**Figure 1.** High-level architecture of a smart NIC.

by the rate or contents of a function's packet stream are out-of-scope for this paper; however, S-NIC is compatible with software-level mitigations for those side channels [27, 67].

S-NIC's isolation guarantees are enforced by hardware. Thus, we assume that NIC hardware is trusted and bug-free. We also assume that NIC hardware is resistant against physical possession attacks in which, e.g., the datacenter provider tries to tap memory buses. S-NIC leverages preexisting techniques to thwart such attacks [38, 51, 84].

When building a constellation of trusted computations (§4.7), we assume that NIC-level functions and host-level secure computations (e.g., SGX enclaves) attest to each other before exchanging packets. Attestation lets endpoints verify their identity and integrity, and generate cryptographic keys for encrypting subsequent communication. Encryption is necessary because datacenter operators may snoop on or tamper with the bus that connects a NIC to its host. Once S-NIC transfers data to a host-level isolation environment, S-NIC cannot prevent any host-level side channels caused by host-level microarchitectural deficiencies; fixing such problems (e.g., [58, 91, 106]) is orthogonal to S-NIC's design.

# 3 Background

In this paper, we target non-trivial network functions written in high-level languages like C++ or Rust. Thus, we focus on smart NICs that use a system-on-a-chip design. SoC NICs often include custom ASICs to accelerate certain tasks; regardless, SoC NICs always include general-purpose CPUs that can run arbitrary code. FPGA-based smart NICs [23, 126] are well-suited for simple, deterministic code that is highly parallel. However, more complex functions are difficult to map to FPGA implementations [76].

## 3.1 Building Blocks

As shown in Figure 1, a SoC-based smart NIC contains six types of hardware components.

- **Programmable cores** run the tenant-provided code belonging to network functions. A commodity smart NIC contains up to dozens of programmable cores.
- **Management cores** execute the software which orchestrates NF execution. For example, management CPUs assign NFs to programmable cores, and configure on-NIC routing hardware to determine which packets get forwarded to which NFs. On some smart NICs, there is no distinction between a programmable core and a management core, i.e., each core runs network functions side-by-side with management software. Management cores pull a function's initial code and data using DMA transfers from host memory.
- A smart NIC also contains **memory**. A NIC has a few GBs of general-purpose DRAM; this memory is accessible by all cores (programmable or management), with each core having the same access latency. A NIC may supplement general-purpose DRAM with smaller SRAM units that have non-uniform access latencies; in the extreme, a unit may be completely inaccessible to some cores.
- **Hardware accelerators** are special-purpose cores that are optimized for a single task like encrypting data or calculating checksums. Network functions write to memory-mapped accelerator registers to install packet-matching rules or cryptographic keys or other kinds of configuration data. Once a function has installed the necessary information, accelerators communicate with programmable cores via input instruction queues and output data queues that live in general-purpose DRAM. Accelerators use local SRAM to cache the NF data being actively processed.
- A smart NIC also provides circuitry to handle **packet ingress and egress**. The specifics vary across different NIC designs. For example, in the Mellanox BlueField NIC, incoming packets enter an RX buffer. A packet input module copies a packet from the RX buffer to the DRAM region that belongs to a particular function. The packet input module uses switching rules to determine how to forward packets; these rules are configured by management software. Rules are typically expressed as predicates over a packet's "5-tuple", i.e., a packet's source IP, destination IP, protocol, source port, and destination port. A programmable core learns of packet arrival by polling hardware structures or receiving an interrupt from the packet input module. After the core has finished handling a packet, the core notifies the packet output module, e.g., by sending an interrupt, or by adding a work item to a shared queue that lives in DRAM. The packet output module copies the packet from DRAM to the TX buffer. Later, the NIC places the packet on the wire.
- An **IO bus** enables communication between the components described above. As we discuss later, network functions contend for bus bandwidth. Ostensibly fair allocation of other resources like hardware accelerators will be unfair in practice if NFs lack the necessary bus bandwidth to optimally use those resources.

A smart NIC also needs a way to communicate with its host machine, e.g., DMA via a PCIe bus. Our proposed design in Section 4 makes no changes to the NIC/host bus.

### 3.2 Representative Smart NIC Architectures

**Marvell LiquidIO:** A LiquidIO NIC [81] uses an OCTEON processor with 12–48 MIPS64 cores. Each core has a private L1 i-cache and d-cache; however, all cores share an L2 cache and general-purpose DRAM. The NIC provides hardware accelerators for checksumming and cryptographic operations.

In the MIPS64 architecture, a virtual address space is partitioned into regions called segments.

- The xuseg segment is mapped to physical memory using TLB entries configured by privileged software.
- The xkseg segment is also mapped to physical memory using TLB entries, but xkseg is only accessible when the CPU's privilege bit is 1. xkseg stores kernel state.
- The xkphys segment is direct-mapped to physical memory, without being translated via TLBs. For example, the hardware automatically translates the first virtual address in xkphys to the first physical address in DRAM. The kernel configures the MMU to determine whether user-level code may access virtual addresses in xkphys.

Unlike an x86 chip, a MIPS processor uses software-defined page table walks. So, a MIPS processor has no explicit page table pointer register; TLB misses are handled by software.

A LiquidIO NIC supports two execution models. In both models, a MIPS CPU acts as both a management core and a programmable core.

- In SE-S mode, the NIC's bootloader installs each function on a core and then exits. Functions cannot be created or destroyed until the next boot cycle. There is no kernel—instead, all functions run in privileged mode. The bootloader configures each core's TLB entries so that xuseg points to function-specific state. Each function also receives complete access to xkphys.
- In SE-UM mode, the management OS is a multicore Linux kernel which creates and destroys network functions as requested by the host machine. Each function is a standard Linux process that the kernel assigns to a core. The kernel sets the core's TLB entries to map a function's xuseg to the physical location of the function's code and data. Depending on how the NIC is configured, the kernel may also give each function access to xkphys, so that a function can directly manipulate packets and memory-mapped registers for accelerators. Alternatively, the NIC can be configured to force functions to use system calls to manipulate packets.

In SE-S mode (and SE-UM mode with function-level xkphys access enabled), *an NF can read and write arbitrary physical addresses*. Furthermore, *an NF can directly manipulate the packet scheduler and hardware accelerators*. This approach maximizes NF performance, but means that LiquidIO provides no isolation between NFs or any privileged management software. Even if a NIC uses SE-UM mode with function-level xkphys access disabled, functions cannot protect themselves from a buggy or malicious OS. Also, in both modes, *the NIC does not isolate microarchitectural state (e.g., cache lines) belonging to different functions.*

**Netronome Agilio:** An Agilio LX [87] makes an architectural distinction between programmable cores and management cores. Up to 120 Intel IXP cores execute network functions, with management software running on a small number of StrongARM or XScale cores. Programmable cores are grouped into islands. Each island has 256 KB of island-private SRAM. The NIC defines additional banks of memory, including a 6GB bank of DRAM. These additional banks are accessible to all islands (but with non-uniform access latencies). Importantly, *all of the memory units are accessed using raw physical addresses*—programmable cores are not restricted via page tables or TLBs. This approach simplifies the NIC architecture, but unfortunately prevents isolation between NFs.

Management cores are responsible for configuring the NIC's packet scheduler and performing other control plane tasks. *The management OS can also tamper with arbitrary function state.* For example, the management OS can install its own network function which can access all of an island's private memory.

An Agilio NIC has several cryptographic accelerators. Programmable cores use special instructions to encrypt or decrypt data with an accelerator. Accelerators are shared by all cores, so contention may increase the latency of a core's cryptographic tasks. *Contention also creates side channels* that let a core determine whether other cores are doing cryptography.

**Mellanox BlueField:** A BlueField NIC uses ARM cores, with each one acting as both a management core and a programmable core. Each core has the same access latency to on-NIC DRAM. BlueField uses ARM's TrustZone technology [6] to isolate functions. To the best of our knowledge, BlueField provides the best isolation of any commodity smart NIC. We give a brief overview of TrustZone before describing the BlueField architecture in more depth.

TrustZone provides hardware-isolated secure computations. TrustZone adds a new privilege bit indicating whether a CPU is running in the "normal world" or "secure world". Memory is split into a normal region and a secure region; code in the normal world cannot access secure memory, but code in the secure world can access all memory. The memory split is managed by secure code, and can change dynamically.

Secure code can mark specific hardware accelerators as secure-only, meaning that the accelerators are inaccessible to normal code. Secure code can also determine which interrupt types are handled by which world. The TrustZone

DMA controller ensures that normal code cannot use DMA-capable devices to read or write secure memory. However, the two worlds can communicate via shared memory. A CPU in normal mode switches to secure mode when a secure interrupt arrives, or when normal code explicitly invokes the secure world via the `smc` ("secure monitor call") instruction. The secure world switches to the normal one via `smc` too.

Both worlds support the traditional user/kernel privilege levels. The normal world runs a heavyweight OS like Linux. The secure world runs a small, security-focused kernel like OP-TEE [74]; small user-mode applications called trustlets run atop the secure kernel.

BlueField uses TrustZone to implement privilege separation for a network function's code [9]. An untrusted driver in the normal world pulls packets from the wire, and transfers them to the trusted part of the function that lives in the secure world. The trusted code handles the packet and then returns it to the normal world for transmission over the wire.

Note that *BlueField does not isolate a network function from the secure-world management OS.* This means that a network function has no protection from a secure-world OS that is corrupted (or intentionally malicious). *BlueField also does not prevent side channels through shared microarchitectural resources like the memory bus.*

### 3.3 Concrete Attacks

The previous section explained why commodity smart NICs provide weak performance isolation, confidentiality, and integrity to functions. Here, we describe proof-of-concept attacks which leverage these problems.

**Packet corruption (LiquidIO):** In SE-S mode, we ran the MazuNAT [36] network function on one core, and a malicious function on another. The MazuNAT function modified packet headers using translation rules that resided in the function's `xuseg` segment. The malicious function leveraged `xkphys` to scan the metadata structures belonging to the buffer allocator used by all functions. The metadata allowed the malicious function to discover the buffers allocated to MazuNAT's packets; the malicious function then corrupted the packet headers in those buffers, disrupting the intended NAT translations.

**DPI ruleset stealing (LiquidIO):** A LiquidIO NIC has accelerators for deep packet inspection (DPI). A DPI accelerator is basically a regular-expression engine. A network function stores its DPI rules in DRAM, with the accelerator pulling in rules as necessary. We wrote a malicious function which uses `xkphys` to steal the ruleset belonging to another function; to locate the ruleset, the malicious function iterated through the metadata of the buffer allocator. This kind of information leak is damaging because it allows a malicious function to learn which threat signatures a target application is using.

**IO bus denial-of-service (Agilio):** To the best of our knowledge, no commodity smart NIC implements bandwidth reservations for the NIC's internal IO bus. Thus, different network functions contend for bus bandwidth, with no trusted hardware-level arbiter to guarantee fair access. On the Agilio, we ran a function which sat in a tight loop, repeatedly issuing a `test_subsat` instruction to decrement a semaphore in DRAM. The function saturated the bus and caused the NIC to hard-crash, requiring a power cycle to recover.

We do not possess a BlueField NIC. Thus, we could not launch concrete attacks against it. However, as described above, even a BlueField NIC (with its comparatively strong isolation) does not protect functions from the secure world OS or from microarchitectural-level side channels. S-NIC's goal is to prevent all of the attacks described in this section, with only modest impacts on performance and die area.

## 4 Design

S-NIC binds each network function to a *virtual smart NIC*. A virtual smart NIC aggregates a physical collection of programmable cores, hardware accelerators, and RAM areas; a virtual smart NIC also possesses reserved bandwidth in the memory bus and the packet input/output modules of the physical smart NIC. The state of a virtual smart NIC is hidden *at both the ISA level and the microarchitectural level* from other virtual smart NICs, and from the NIC OS that runs on management cores. ISA-level isolation means that external network functions or the NIC OS cannot use ISA-level instructions to read or write the virtual NIC's state. Microarchitectural-level isolation means that each microarchitectural resource is either strictly bound to a single function, with no time-slicing, or is shared, but with hardware-enforced resource reservations that do not leak information about the usage patterns of the reservation's owner. Table 1 summarizes the interfaces that S-NIC exposes to software; we explain those interfaces in more detail below.

To implement its isolation semantics, S-NIC employs the microarchitecture shown in Figure 2. The key research challenge was determining how to *pervasively* virtualize a NIC's resources. Using memory denylists that can only be controlled by trusted hardware, S-NIC implements single-owner semantics for RAM areas (§4.2). To bind hardware accelerators to specific network functions, S-NIC groups individual hardware threads into clusters and then places each cluster behind a single TLB bank; the cluster acts as a virtual accelerator, with S-NIC configuring the cluster's TLB bank so that the cluster's hardware threads can only access the RAM belonging to a single network function (§4.3). To enforce fair-sharing of the IO bus, S-NIC adds a trusted bus arbiter; the arbiter, in combination with several other components, provides guaranteed memory bandwidth and packet throughput to a virtual NIC (§4.4 and §4.5). Finally, S-NIC uses remote attestation [3, 95, 115, 116] to allow a network

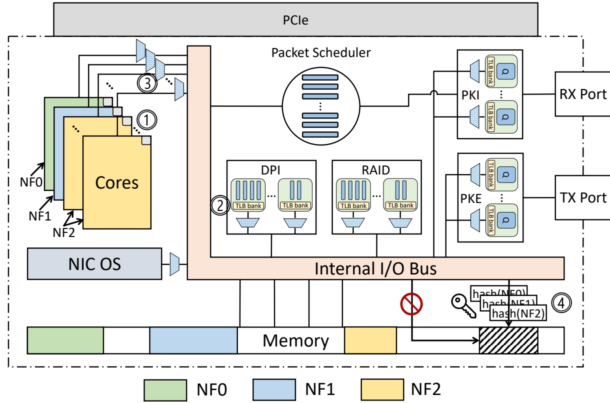



**Figure 2.** S-NIC's high-level architecture. Locked-down TLB entries (①) restrict the memory that each network function can access. Virtualized hardware accelerators are also restricted using TLB locks (②). Bus arbiters provide reserved bus bandwidth to programmable cores and other disaggregated components (③). Using a hardware root of trust, network functions can remotely attest the confidentiality and integrity of their state (④).

function to vouch for the confidentiality and integrity of its state. Remote attestation allows a datacenter tenant to stitch together a constellation of S-NIC network functions and secure host-level computations, e.g., SGX enclaves or TrustZone worlds (§4.7).

### 4.1 Launching a Function

To launch a network function, a remote developer first uploads the function's initial code and data to the RAM of a datacenter host. The developer also uploads the function's configuration state. Some of that state describes reservation requests for hardware resources. For example, a function may request a virtual smart NIC with three cores, 40 MB of RAM, two cryptographic accelerators, and a compression accelerator. A function's configuration state also specifies the 5-tuple packet matching rules that the NIC should use to determine whether to forward a packet to the function.

The on-NIC OS uses DMA to transfer the initial function state from host RAM to on-NIC RAM. S-NIC runs the NIC OS on a dedicated core; all other cores are used to run functions. Once the DMA transfers complete, the NIC OS executes the privileged nf_launch instruction (Table 1). This instruction atomically installs a new function. The instruction takes six arguments that are specified via CPU registers. Each argument describes a specific set of physical resources that should be bound to a virtual NIC. We discuss the first two arguments now, and the rest in later sections.

The first argument to nf_launch is a bitmask that lists the cores which the trusted hardware should bind to the new function. The second argument is a pointer to a page table; the referenced physical pages contain the initial code, data, and configuration settings for the new function. The untrusted NIC OS controls these arguments, and may specify incorrect values. Fortunately, such problems are detectable later, during function attestation (§4.7).

The trusted hardware maintains a bitmap which tracks which cores have been allocated to a network function. The hardware maintains another bitmap which tracks which physical RAM pages have been allocated to a network function.[1] When the NIC OS invokes nf_launch, the instruction checks whether the requested cores are unassigned, failing if some of the cores are currently bound to live functions. Otherwise, nf_launch walks the page table referenced by the second argument. If any of the physical pages mapped by the page table already belong to a function, nf_launch fails.

### 4.2 Single-owner RAM Semantics

RAM is perhaps the most important ISA-visible resource on a smart NIC. When packets arrive, the packet input module copies the packets to RAM; the switching rules that determine exactly where to copy each incoming packet also live in RAM. Hardware accelerators pull instructions from RAM, and output results to RAM. A network function also uses RAM to store general-purpose code and data. Thus, a major security goal of S-NIC is to implement *single-owner semantics* for each region of RAM: a region exclusively belongs to either a running network function or the management OS.

**Isolating the new function from the management core:** If nf_launch validates both arguments, the instruction installs a memory page denylist on the management core. The denylist prevents the management core from accessing any physical pages that belong to the new function. To implement the denylist, S-NIC associates a denylist page table register with the management core. The denylist page table, which resides in private hardware memory, contains a mapping for a physical address if that address should not be accessed by the management core. When the management core suffers a TLB miss, the management core walks its normal page table (if using an x86-style approach) or uses software to update a TLB register (if using a MIPS-style approach). When the management core tries to install a virtual-to-physical mapping, the trusted hardware uses the physical address in the new mapping to walk the denylist page table. If the denylist table has an entry for the physical address, the trusted hardware rejects the management core's attempt to update the TLB. S-NIC's use of dual page tables is somewhat reminiscent of the EPT mechanism used to implement shadow paging [15]; thus, S-NIC's denylist implementation can use the same hardware optimizations that efficient EPT implementations use.

**Isolating old functions and the management core from the new function:** Once nf_launch has installed the TLB

[1] At the microarchitectural level, this "bitmap" could be implemented in a variety of ways. For example, the bitmap could literally be a bitmap, or its logical functionality could be implemented by traversing the page tables of programmable cores. The former option is faster but requires more die area.

| Management APIs | Trusted instructions | Descriptions |
|---|---|---|
| **NF_create**(net_config, core_config, dpi_config, ...) → nf_id or failure | **nf_launch**: core_mask, page_table, pkt_pipeline_config, accel_mask → nf_id or failure | Atomically install a network function on a virtual smart NIC: reserve the necessary physical resources, configure the isolation hardware, calculate a hash of the initial function state, and then launch the function. |
| N/A | **nf_attest**: pointer to <$g$, $p$, $n$, $g^x$ mod $p$> → $S_{AK_{priv}}$<Hash<nf's initial state>, $g$, $p$, $n$, $g^x$ mod $p$> | Using the attestation key $AK_{priv}$, sign the hash of the function's initial state plus the Diffie-Hellman parameters. |
| **NF_destroy**(nf_id) → success or failure | **nf_teardown**: nf_id → success or failure | Atomically destroy an NF and release its resources. |

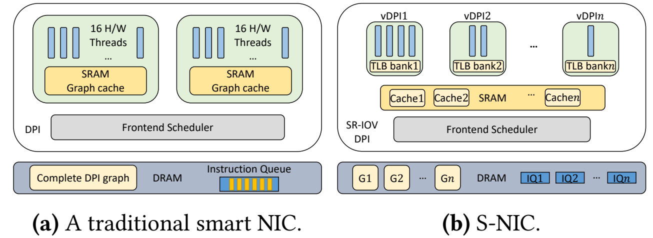
**Table 1.** The first column describes the host-visible management APIs exposed by an S-NIC OS. The second and third columns describe the underlying S-NIC hardware instructions.

denylist for the management core, nf_launch installs the memory mappings for a function's programmable cores. At a high level, each core receives a mapping that allows the core to access its own DRAM pages, but not pages belonging to other functions or the NIC OS. The details of the mapping depend on how the core accesses memory. For example, common network functions require less than 70 MB of virtual address space to hold the state for active flows; the largest function that we tested required 360 MB (Appendix B). Address spaces of this size can be covered by a handful of TLB entries, with variable-sized pages (e.g., 2 MB, 32 MB, and 128 MB) minimizing internal fragmentation. Thus, a typical hardware implementation for S-NIC will not associate a page table pointer with a programmable core. Instead, each core will get a small number of TLB entries which are configured by nf_launch to cover all valid mappings for the function. Once nf_launch completes, the hardware sets the TLBs to read-only; any subsequent TLB misses represent a bug in the network function, and cause S-NIC to destroy the function. In an alternate design that does associate a page table with each programmable core, the page table pointer register (and the pointed-to memory pages) would become read-only after nf_launch completes.

A network function may want to transfer data to or from host-level memory. S-NIC's DMA controller must provide isolation for both transfer directions. In other words, the host should only be able to transfer data to a specific on-NIC RAM location that is owned by the function; the function should only be able to transfer data to a host-sanctioned region in host RAM. S-NIC achieves these properties using a multi-bank DMA controller, with one bank per programmable core. Each bank has TLB entries for the upstream and downstream transfer directions. This approach is similar to the one used by SR-IOV DMA engines [56].

**Eliminating side channels:** After configuring the MMUs for the management core and the function's programmable cores, nf_launch reserves dedicated L1/L2/L3 cache space for the function. Since S-NIC must prevent cache-based side channels, soft partitioning schemes like Intel CAT [57] provide insufficient isolation.[2] S-NIC has two options. The first

[2]The partitioning is "soft" because a core can only write to a specific cache region, but can read (i.e., satisfy a cache hit) from any region.




**(a)** A traditional smart NIC. **(b)** S-NIC.

**Figure 3.** Virtualizing hardware accelerators

is to use a hard partitioning of the cache; this suffices to eliminate side channels [93], but prevents S-NIC from dynamically resizing cache allocations as a function's workload changes. Alternatively, if S-NIC is willing to allow side channels from the NIC OS to functions (but not vice versa), S-NIC can use SecDCP cache partitioning [120]. In this approach, each function receives a minimum cache allocation. Trusted cache hardware [99] examines utilization by functions and the NIC OS, and only resizes allocations in response to the cache behavior of the NIC OS.

### 4.3 Virtualizing Hardware Accelerators

In commodity smart NICs, a single hardware accelerator consists of one or more "hardware threads." A scheduler inside the accelerator pulls requests from a queue in RAM, and assigns each request to a thread. To complete the request, a thread may need to pull additional data from RAM. For example, Figure 3a shows a DPI accelerator (§3.3) in a traditional smart NIC. A network function uses the accelerator by (1) writing a finite automata graph to RAM, (2) registering the graph with the DPI, and then (3) registering the instruction queue with the DPI. The function performs steps (2) and (3) by writing to the DPI's memory-mapped IO registers. Internally, the DPI caches parts of the graph in private SRAM.

Commodity smart NICs often provide hardware threads with unfettered access to physical RAM. Many NICs also provide network functions with unrestricted RAM access (§3.2), meaning that accelerator state has no confidentiality or integrity. To fix this problem, S-NIC statically assigns each thread to a cluster, and places a TLB bank in front of each cluster. When nf_launch installs a network function, the hardware checks whether the requested number and

types of clusters are available. If so, the hardware marks the clusters as allocated and then configures the associated TLB banks so that hardware threads can only access the physical memory that belongs to the new function. The hardware also ensures that each thread's memory-mapped registers are privately and directly mapped to a well-known location in the function's virtual address space. This prevents other functions or the NIC OS from configuring the threads.

Figure 3b shows an example of a virtual DPI (vDPI). Using memory denylisting, each function's DPI graph and instruction queue receive confidentiality and integrity. The front-end hardware scheduler also reserves guaranteed DRAM bandwidth for each vDPI (§4.5), preventing side channels via DRAM contention.

A cluster's TLB bank is read-only after nf_launch finishes. A cluster does not require a page table, because a cluster only needs access to a small, contiguous range of virtual memory. Thus, S-NIC treats any cluster TLB misses as fatal errors.

### 4.4 Virtualizing Packet IO

To isolate the packet handling workflow for each network function, we take inspiration from SR-IOV [56], a preexisting technology for virtualizing a (non-smart) NIC. SR-IOV assigns a MAC address to each virtual NIC. When a packet arrives, an internal Layer 2 switch inspects the MAC address and adds the packet to the queue of the relevant virtual NIC. An IOMMU restricts the physical pages that a virtual NIC can access via DMA.

In S-NIC, a virtual packet pipeline (VPP) is a bundled group of hardware resources that move a function's packets between the wire and the function's private RAM. A VPP consists of:

- buffer space in the physical RX and TX ports,
- a packet scheduler (which copies incoming packets from the RX queue to RAM, and outgoing packets from the TX queue to the wire), and
- switch configuration rules that live in RAM and determine which incoming packets are forwarded to the VPP.

The pkt_pipeline_config argument to nf_launch configures a function's VPP. The argument points to a buffer in memory that specifies the requested amount of RX/TX buffer space, the desired packet scheduling algorithm [107, 110], and the switch configuration rules. If nf_launch can find the requested amount of buffer space in the physical ports, nf_launch marks the associated regions as allocated. Then, nf_launch installs the desired scheduler, and adds the switch configuration rules to the set of denylisted physical memory pages. S-NIC assigns one scheduler unit to each programmable core, and locks the scheduler's TLB entries to ensure that the scheduler can only perform DMA operations on memory regions that are owned by the associated network function.

VXLAN [55] is a popular datacenter technology for giving a tenant the illusion of a private Layer 2 network. When tenant-owned software tries to send a Layer 2 frame, the frame is actually encapsulated within a VXLAN frame. The VXLAN header contains the Virtual Network Identifiers (VNIs) for the tenant-visible Layer 2 source and destination; however, the VXLAN frame traverses switches as determined by the datacenter's mapping from the tenant's virtual L2 topology to the physical L2 topology. The ultimate receiver strips the VXLAN header before injecting the frame into the receiver's network stack. S-NIC supports VXLAN by allowing developer-specified switching rules to mention VNIs in addition to MAC addresses and 5-tuple data. This allows a NIC to direct specific VXLAN flows to specific functions.

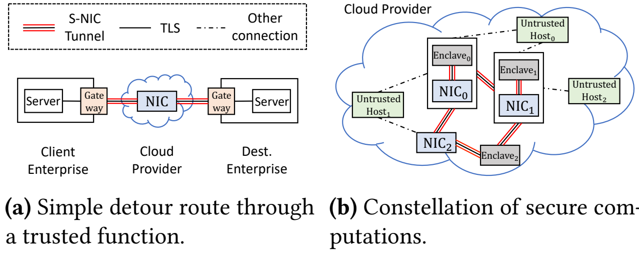
### 4.5 Bus Arbitration

To prevent side channels via IO bus contention, S-NIC must prevent interference between the bus requests that are issued by different tenants. S-NIC is compatible with various approaches for doing so [33, 103, 119]. The S-NIC prototype that we evaluate in Section 5 uses temporal partitioning [119], a particular approach that has simple hardware requirements and works well for applications like network functions that generally create a lot of bus traffic. As shown in Figure 2, S-NIC places bus arbiters in front of hardware components that access the IO bus. S-NIC divides time into epochs; during each epoch, only a single bus client may initiate memory operations. When the epoch expires, the bus switches to another client. To ensure that any in-flight memory operations complete before the end of the epoch, the arbiter must only allow new memory operations to issue during the first part of the epoch. Despite the "dead time" at the end of each epoch, temporal partitioning decreases computational speeds by less than 5% [119] when using four security domains (which in our setting translates to four concurrently-executing, potentially multi-core network functions). In concert with VPP hardware reservations, temporal partitioning eliminates watermark attacks that leverage packet flow interference [11].

### 4.6 Function Execution and Teardown

As nf_launch installs various parts of a function, nf_launch updates a cumulative hash. When nf_launch completes, the hash will represent a fingerprint of the state used to create the function. For example, the hash will cover the initial code and data pages of the function, as well as the switching rules which select the packets that are forwarded to the function. This hash will later be used during remote attestation (§4.7).

If nf_launch succeeds, it stores the arguments to nf_launch in hardware-private memory. The instruction then returns an opaque integer representing the function's id. The function is now running. At this point, the NIC OS is no longer involved in the management of the hardware resources that are bound to a function. Indeed, the NIC OS cannot even access those resources due to memory denylisting and S-NIC's techniques for microarchitectural-level isolation.




**(a)** Simple detour route through a trusted function. **(b)** Constellation of secure computations.

**Figure 4.** Sample use cases for S-NIC. In use case (a), two enterprises use S-NIC to outsource packet processing (e.g., intrusion detection) for a cross-enterprise flow; an S-NIC tunnel connects the gateways and the function to hide packet headers from the untrusted cloud. In use case (b), a tenant creates a constellation of trusted computations within an untrusted cloud.

The NIC OS does require the ability to destroy a function in a way that leaks no information about the function. S-NIC provides this capability via nf_teardown. This instruction, which executes atomically, is the dual of nf_launch. For example, nf_teardown marks $F$'s cores as unallocated, unbinds $F$'s accelerators, and zeroes out $F$'s physical pages before removing them from the memory denylist. The instruction also zeroes out the registers and cache lines used by $F$.

### 4.7 Attestation and Secure Constellations

Attestation [3, 95, 115, 116] allows a network function to prove to external peers that the function (1) is running atop an authentic S-NIC, and (2) had a specific initial state, where "state" corresponds to the information covered by the cumulative hash that trusted hardware built during function initialization (§4.6). In Appendix A, we describe the low-level cryptographic details of how S-NIC attestation works. At a high level, a function $F$ attests to a peer $P$ by engaging in a Diffie-Hellman exchange [34]. $F$'s contributions to the exchange include the cumulative hash of $F$'s initial state; $F$ retrieves that value from the trusted S-NIC hardware, who signs the value with a NIC-specific private key whose public key is endorsed by a certificate from the NIC's hardware vendor. At the end of the attestation protocol, both $F$ and $P$ have established a symmetric key that only they know. Furthermore, $P$ is convinced that $F$ had a known initial state and is running atop a legitimate S-NIC.

If $P$ runs atop trusted hardware as well (e.g., because $P$ resides within an SGX enclave [82] or a TrustZone secure world [6]), $F$ can now ask $P$ to attest to $F$. If the pairwise attestations succeed, the endpoints now share an encrypted network channel and possess mutual confidence in the integrity and confidentiality of both computational environments. Pairwise attestations allow a developer to build a constellation of trusted computations spanning multiple S-NIC functions and host-level hardware enclaves. Figure 4 provides a visual overview.

### 4.8 Discussion

**Implementing nf_launch:** nf_launch is a complex instruction. Thus, we expect it to be implemented in microcode, similar to how complex SGX instructions are implemented [31]. A single nf_launch instruction will require tens of thousands of cycles to complete, but S-NIC targets datacenter environments in which network functions live for minutes or hours. Thus, the cost of an nf_launch instruction is amortized over minutes or hours.

**Underutilization:** S-NIC provides a virtual NIC with strong isolation of both its ISA-visible state and its microarchitectural state. However, this strong isolation may lead to underutilization of physical resources. For example, suppose that the NIC OS allocates $P$ pages of physical memory to function $F$, and then calls nf_launch. After the call to nf_launch, $F$ cannot return pages to the OS, e.g., if $F$'s workload decreases. S-NIC intentionally prohibits such interactions to prevent side channels via the status of OS-managed resources [44, 121]. Similarly, if a function goes idle, the function cannot temporarily relinquish one of the programmable cores in its virtual NIC. The tension between strong isolation and underutilization is fundamental, given the lack of trust between the different code on the NIC. When using an S-NIC, physical utilization should be kept high by creating or destroying functions in response to time-varying load.

We note that, in practice, commodity clouds allocate cores to non-burstable VMs at the granularity of a full core, binding each VM to a unique set of physical cores [125]. Thus, S-NIC's core-granular allocations to NFs should not be surprising to developers of cloud applications.

**Denial of service attacks:** The NIC OS may refuse to launch functions, or tear them down prematurely. Denial of service attacks are out of scope for this paper. A buggy or malicious NIC OS may also improperly setup a function, e.g., by omitting a code page from the registration process. Remote clients can detect improper function setups by requiring the function to attest (§4.7).

**Chaining functions:** In prior sections of the paper, we assumed that S-NIC runs a single function in each virtual NIC. S-NIC's strict isolation semantics prohibit the sharing of memory data between functions in different virtual NICs. However, commodity smart NICs often support function chains in which a single packet is passed through a series of functions. S-NIC is compatible with function chaining via compiler-enforced isolation [13, 63, 98]. For example, using the strong types and language-level isolation primitives provided by Rust, S-NIC could place multiple distrusting functions in the memory region belonging to a single virtual NIC. However, this approach would enable cross-function side channels via core-local microarchitectural state (§4.2).

An extended version of S-NIC could have NFs exchange data via localhost networking, such that S-NIC hardware would transfer messages directly between the side-channel-isolated VPPs (§4.4) owned by different NFs. Assuming that the cross-VPP management hardware exposed no side channels itself, this approach would restrict the information leakage between two communicating VPPs to just the information that is revealed via overt traffic timings and packet content. We leave the design of such management hardware to future work.

# 5 Evaluation

In this section, we demonstrate that S-NIC's hardware costs are modest: chip area increases by only 8.89% and power consumption increases by only 11.45%. As a result, S-NIC reduces the TCO advantage of a smart NIC by only 8.37%. Furthermore, S-NIC's TLB denylisting, cache partitioning, and bus arbitration reduce function throughput by less than 1.7% in the worst case. Thus, S-NIC preserves the traditional performance benefits of function offloading.

## 5.1 Workloads

We used six different network functions to evaluate S-NIC.

- **Firewall (FW):** A stateful firewall that drops packets by scanning a list of rules. Recently-accessed rules are cached in a HashMap implemented by Rust's standard library. We limit the cache size to 200,000 entries, which is the cached flow limit in Open vSwitch [96]). The function uses rules from the Emerging Threats site [40]. We configure the function with 643 rules, as in the SafeBricks paper [98].
- **DPI:** A pattern-matching application that uses the Aho-Corasick algorithm [1]. We use an efficient SIMD-accelerated implementation provided by the aho_corasick Rust crate. We use 33,471 patterns extracted from six open source rulesets [29, 39].
- **NAT:** A network address translator derived from MazuNAT [36]. The NAT uses a HashMap to cache frequently-used translations. The cache only records the translation results of the first 65,535 flows that can be successfully assigned a distinct port number.
- **Load Balancer (LB):** Google's software load balancer called Maglev [37]. This function uses consistent hashing to distribute flows.
- **LPM:** Longest prefix matching using the DIR-24-8 algorithm [52] for IP packet routing. Like NetBricks [94], we generate 16,000 random rules to construct the lookup table.
- **Monitor (Mon):** Uses a HashMap to record the number of packets for each 5-tuple flow.

We implemented the DPI and Monitor functions ourselves. The other four implementations were derived from versions in the NetBricks repository [30].

**Traces:** Our evaluation used two traces. The first was a one-hour, anonymous CAIDA trace from 2016 [24]. This trace had 26.7 million TCP flows and 1.34 billion packets. The second trace was an ICTF trace from 2010 [113]. This trace, from which we randomly sampled 100,000 flows (§5.3), was collected during a wide-area "capture-the-flag" competition, and was the same trace used by the SafeBricks paper [98]. We ran each experiment 10 times, and we report the median for memory usage and throughput.

| | | 4-core A9 Total | 4-core NIC | 8-core NIC | 16-core NIC | 48-core NIC |
|---|---|---|---|---|---|---|
| 366MB per core (183 TLB entries) | Area (mm$^2$) | 4.984 | 0.045 (0.90%) | 0.090 | 0.179 | 0.538 |
| | Power (W) | 1.909 | 0.026 (1.36%) | 0.052 | 0.104 | 0.311 |
| 512MB per core (256 TLB entries) | Area (mm$^2$) | 4.999 | 0.060 (1.20%) | 0.120 | 0.239 | 0.718 |
| | Power (W) | 1.913 | 0.035 (1.81%) | 0.069 | 0.139 | 0.416 |
| 1024MB per core (512 TLB entries) | Area (mm$^2$) | 5.102 | 0.163 (3.19%) | 0.326 | 0.652 | 1.956 |
| | Power (W) | 1.971 | 0.088 (4.45%) | 0.175 | 0.351 | 1.052 |

**Table 2.** Estimated hardware costs for TLBs on programmable cores. We assume a 2MB page size, and use our memory profiling results (Table 6 in Appendix B) to calculate the minimum TLB size (i.e., 183 entries) that can map each function's memory. We examine a variety of core counts, based on the counts of existing smart NICs: a Marvell NIC has 12–48 cores, a Mellanox NIC has 4–16 cores, and a Broadcom NIC has 8 cores. In the 4-core S-NIC configuration, we provide (in parentheses) relative hardware overheads compared to the total cost of a 4-core A9 processor.

| | | DPI | ZIP | RAID |
|---|---|---|---|---|
| TLB size per cluster | | 54 | 70 | 5 |
| 16 clusters (4 threads per cluster) | Area (mm$^2$) | 0.074 | 0.091 | 0.050 |
| | Power (W) | 0.037 | 0.044 | 0.023 |
| 8 clusters (8 threads per cluster) | Area (mm$^2$) | 0.037 | 0.046 | 0.025 |
| | Power (W) | 0.019 | 0.022 | 0.012 |
| 4 clusters (16 threads per cluster) | Area (mm$^2$) | 0.019 | 0.023 | 0.012 |
| | Power (W) | 0.009 | 0.011 | 0.006 |

**Table 3.** Estimated hardware costs for TLB banks on virtualized accelerators. For each accelerator, we estimate its per-cluster TLB size based on memory profiling (see Table 7 in Appendix B). We assume there are 64 hardware threads for each accelerator.

## 5.2 Die Area and Power Consumption

Existing smart NIC vendors do not publicly share the chip area and power consumption of their products. So, we evaluated the additional hardware cost of S-NIC by extending an ARM Cortex-A9 multicore processor [7]. Although the overall size and complexity of the A9 might be different than that of current smart NICs, the process technology (28nm) and frequency target (2.0GHz) match current smart NIC designs. So, the absolute hardware sizes for components like TLBs should be comparable. Given that the A9 is a relatively small processor, we expect our estimates of *relative* cost to be high compared to actual smart NIC designs.

To generate our cost estimates, we used the McPAT modeling framework [71]. This framework, widely used in the architecture community, provides area, power, and timing estimates for multi-core processors.

| | | Virtual packet pipeline | DMA |
|---|---|---|---|
| TLB size per VPP/vDMA | | 3 | 2 |
| 12 VPP/vDMA (4 cores per NF) | Area ($mm^2$) | 0.037 | 0.037 |
| | Power (W) | 0.017 | 0.017 |
| 6 VPP/vDMA (8 cores per NF) | Area ($mm^2$) | 0.019 | 0.019 |
| | Power (W) | 0.009 | 0.009 |
| 3 VPP/vDMA (16 cores per NF) | Area ($mm^2$) | 0.009 | 0.009 |
| | Power (W) | 0.004 | 0.004 |

**Table 4.** Estimated hardware costs for TLB banks on the virtual packet pipeline and the DMA controller that mediate NIC/host data transfers. We assume there are 48 programmable cores on the smart NIC. Note that, in McPAT, 2 TLB entries have the same cost estimation as 3 TLB entries because the size difference is so small.

**Overall cost:** The majority of S-NIC's hardware costs derive from three sources: (1) TLBs for programmable cores, (2) TLBs for virtualized accelerators, and (3) TLBs for the virtual packet pipeline and the DMA controller that manages NIC/host data transfers. Smaller costs arise from denylisting logic and bus arbiters; we discuss those costs later.

Table 2 shows how cost changes with the number of programmable cores and the amount of memory that each core must be able to access. Our analysis assumes a 2 MB page size, since many NFs leverage huge pages to minimize TLB miss costs. Note that a TLB's size does not need to be a power of two, as TLBs use fully-associative logic. Table 2 shows that, for a smart NIC with 4 processors, S-NIC only requires an additional 3.19% die area and 4.45% energy, relative to a 4-core A9. With additional cores, the relative costs of the TLB will decrease. The reason is that TLB area scales roughly linearly with core count, but total processor area will scale superlinearly due to larger cache directories and core interconnects.

Table 3 shows how cost changes with the accelerator type and cluster size. Aggregating over the total TLB costs for the DPI, ZIP, and RAID accelerators, S-NIC adds up to 4.2% more die area and 5.3% more power consumption, given a baseline 4-core A9 with a TLB size of 512 entries.

Table 4 shows how cost changes with the number of cores per network function. The total cost is a 1.5% increase in chip area, and 1.7% additional power draw, given a baseline 4-core A9 with a TLB size of 512 entries. To summarize all three tables, *S-NIC's additional TLB entries add 8.89% more chip area and 11.45% more power consumption compared to a baseline 4-core A9.*

**TCO impact:** Here, we give a rough estimate of the total-cost-of-ownership (TCO) for an S-NIC and a traditional smart NIC (namely, a 12-core Marvell LiquidIO NIC). According to analysis from Liu et al. [78], the LiquidIO NIC has a peak power draw of 24.7W, and a purchase cost of $420. A 12-core Intel E5-2680 v3 processor (as used by a host server) has a peak power draw of 113W, and a purchase cost of $1745 [59].

| Page size settings | TLB size | Area ($mm^2$) | Power (W) |
|---|---|---|---|
| Equal (2MB) | 183×16 | 0.538 | 0.311 |
| Flex-high (128KB,2MB,64MB) | 51×16 | 0.214 | 0.106 |
| Flex-low (2MB,32MB,128MB) | 13×16 | 0.150 | 0.069 |

**Table 5.** TLB hardware costs as a function of the supported page sizes and the number of programmable cores. For each page size configuration, we profiled six popular network functions (Table 6 in Appendix B), and used the maximum number of TLB entries that any function would require. We assume that there are 48 programmable cores on the smart NIC.

Similar to the analysis of Liu et al., we use the average electricity price in a U.S. datacenter [109]—$0.0733/kWh—to calculate the three-year TCO per core in a LiquidIO NIC and a host server: the LiquidIO NIC has a three-year TCO of $38.97 per core, whereas a host server has a three-year TCO of $163.56 per core. Given that S-NIC increases SoC chip area by up to 8.89% and power consumption by 11.45%, the three-year TCO of an S-NIC-extended LiquidIO NIC would increase to $42.53 per core in the worst case. Thus, *S-NIC decreases a smart NIC's TCO advantage over a host-based solution by up to 8.37%.* The 8.37% number is a rough estimate, and is sensitive to various factors, like whether a datacenter operator can purchase NICs more cheaply using bulk discounts. However, the high-level observation is that S-NIC's extra security still preserves most of the TCO advantages conferred by smart NICs.

**Sizing a programmable core's TLB:** Table 5 estimates the number of TLB entries required by a programmable core. To estimate the appropriate TLB sizes, we profiled the maximum memory usage of our six evaluation functions (§5.1). For the Monitor function, we measured flows in the CAIDA trace every five minutes (as in the UnivMon paper [79]) and recorded the maximum memory usage. The other functions had bounded memory usage, even with increasing numbers of flows.

Table 5 shows the required number of TLB entries under three page size settings: the *Equal* setting only allows 2 MB pages; the *Flex-low* setting allows page sizes of 128 KB, 2 MB, and 64 MB; and the *Flex-high* setting allows page sizes of 2 MB, 32 MB, and 128 MB. Our evaluated NFs typically only required 13–69 MB memory; thus, to handle most network functions, a programmable core only needs dozens of TLB entries. The Monitor function required the most memory (361 MB) and the most TLB entries (183).

**Sizing a virtualized accelerator's TLB:** We profiled the memory usage of three popular accelerators: a deep packet inspection (DPI) engine, a data compressor (ZIP), and a storage accelerator (RAID). For DPI, we used a matching graph with 33K rules, consuming 97 MB of RAM. For ZIP, the compression dictionary was 32KB. For RAID, the SGP buffer size was 128 MB. Table 7 in Appendix B shows detailed memory

profiles for each accelerator. Here, we simply note that, if TLBs only support a 2 MB page size, then a virtualized DPI engine requires a 54-entry TLB. A virtualized ZIP accelerator requires 70 entries, but a virtualized RAID unit only needs 5.

**Sizing the TLB for a virtual packet pipeline and DMA controller:** A virtual packet pipeline (§4.4) accesses three buffers:

- a packet buffer (PB) for storing packet data;
- a packet descriptor buffer (PDB) for storing packet metadata like the address and length of a packet buffer; and
- an output descriptor buffer (ODB) for storing metadata about outgoing packets.

On a LiquidIO NIC, these buffer sizes are 2 MB, 128 KB, and 1 MB respectively. Thus, if S-NIC uses the same sizes, a virtual packet pipeline will require 3 TLB entries.

To manage DMA transfers between a network function and its host, a DMA engine requires access to the function's PB (2 MB), and a DMA instruction queue (which is 256 KB per SR-IOV function on a LiquidIO NIC). Thus, an S-NIC DMA engine needs 2 TLB entries per function.

**Cost of management-core denylisting and bus arbitration:** As mentioned in Section 4.2, S-NIC implements memory denylisting via dual page table walks. The hardware-level costs will be similar to those of EPT [15], which are small. Section 4.5 explains why the cost for bus arbitration is also small.

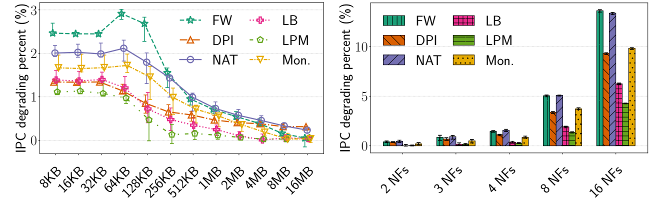
### 5.3 Performance Costs of Strong Isolation

As mentioned in Section 1, offloading a function to a smart NIC can improve performance. S-NIC might decrease those benefits in three ways:

- S-NIC eliminates cache-based side channels by cache partitioning (§4.2). Restricting a function's access to cache lines might prevent the function's working set from completely fitting in the cache.
- S-NIC eliminates bus-based side channels using bus arbitration (§4.5). Arbitration might cause a memory-bound function to stall more frequently.
- To enforce ownership semantics for RAM (§4.2), S-NIC puts TLBs in front of accelerators and programmable cores.

On modern architectures, TLB translations impose negligible overhead (assuming that there are no TLB misses, which S-NIC ensures (§4.2 & 4.3)). Thus, S-NIC's performance costs arise solely from cache partitioning and bus arbitration.

To evaluate these costs, we used gem5 [16] to simulate an S-NIC that used static partitioning for the cache and temporal partitioning [119, 122] for the bus. Static partitioning allocated $1/N$ of the cache to each of the $N$ functions. Our simulated NIC had multiple out-of-order, 1.2 GHz ARM cores that used a two-level cache and 16 GB of 1,600 MHz DDR3 RAM. We configured the core frequency, cache line size, L1




**(a)** Varying L2 cache size (2 co-located NFs). **(b)** Varying co-tenancy (4MB L2 cache).

**Figure 5.** IPC degradation induced by cache partitioning and bus arbitration. For each experimental setting, we calculate the median IPC degradation of a function by running every possible colocation with other functions, and determining the median IPC decrease; we also plot 1st and 99th percentile error bars across all possible colocations.

cache size, and cache associativity and latency to match those of the Marvell smart NIC described in the iPipe paper [76]. Other microarchitectural settings used the gem5 defaults.

Since gem5 does not currently simulate high-speed NICs, we fed packets directly into RAM and rewrote functions to directly access packets in memory. Packet streams came from a pool of 100,000 flows that were uniformly sampled from the ICTF trace; those traces had a Zipf distribution with a skewness of 1.1. Before collecting experimental results, we ran 1 billion instructions to warm microarchitectural structures like caches and branch predictors. We then collected experimental data for the next 100 million instructions. We measured the impact of cache partitioning and bus arbitration on instructions-per-cycle (IPC), since, for a function that always has work to do, IPC is directly correlated with function throughput. IPC degradation was calculated with respect to the IPC of baseline hardware with the same cache size and the same degree of function cotenancy, but no cache partitioning and no bus arbitration.

As shown in Figures 5a and 5b, S-NIC's performance overheads increase as the L2 cache size decreases and the degree of cotenancy increases. However, network functions that only examine packet headers are not memory-intensive [61], so IPC is mostly insensitive to cache size. For example, when running 2 NFs and using a 4MB L2 cache (equivalent in size to a Marvell NIC's L2 cache), the average (median) IPC degradation is 0.24%. When colocating 4 NFs and using 4MB L2 cache, the average (median) IPC degradation is 0.93%, with a worst (99th percentile) IPC degradation of 1.66%. With 8 NFs, the average of the median IPC degradation is 3.41%, with a 99th-percentile degradtion of 5.12%. With 16 NFs, the average is 9.44% with a 99th-percentile degradation of 13.71%. The firewall, DPI, and NAT functions suffered the worst degradations due to their larger working sets. Regardless, we believe that these IPC reductions are acceptable, given the strong isolation that S-NIC provides.

## 6 Related Work

**Trusted execution environments:** A variety of prior research has investigated hardware support for secure computation [17, 25, 32, 43, 66, 72, 92, 111, 112]. Commercially, Intel SGX [82] and ARM TrustZone [89] are the most popular approaches. SGX and TrustZone provide helpful isolation guarantees. Unfortunately, both platforms are vulnerable to a variety of side channel attacks due to microarchitectural co-tenancy of trusted and untrusted code [2, 12, 20, 26, 42, 48, 75, 104, 123]. For example, if a single hyperthreaded physical core runs secure code on one logical core, and insecure code on another, side channels arise from contention on the physical core's functional units. S-NIC removes such problems by eliminating microarchitectural co-tenancy.

Unlike SGX and TrustZone, S-NIC does not force a secure computation to rely on an untrusted OS to perform IO. In S-NIC, a network function has direct control over a virtual packet pipeline. The datacenter-provided NIC OS cannot tamper with the VPP once a function has been launched.

S-NIC also enables fine-grained virtualization of heterogeneous components like DPI units. In contrast, SGX focuses on isolating CPUs and memory. TrustZone allows devices to be attached to the normal world or the secure world, but does not provide fine-grained partitioning and side-channel avoidance at the microarchitectural level.

**Host-level network functions:** Before the advent of smart NICs, network functions had to run on the host machine. xOMB [4], CoMB [102], FastClick [10], and Metron [62] run all functions in a single process, and provide no isolation. NetVM [54], OpenNetVM [124], ClickOS [80], HyperSwitch [100], mSwitch [53], OpenBox [19], and Slick [5] rely on VMs to isolate different NFs.

NetBricks [94] requires network functions to be written in Rust; NetBricks takes advantage of Rust-level isolation features to prevent functions from tampering with each other's state. The SafeBricks extension [98] runs network functions inside enclaves to protect functions from an untrusted OS. However, as discussed earlier, SGX enclaves do not solve important challenges involving side channels, fair resource allocation, and fine-grained hardware virtualization. Also note that, even though SafeBricks uses DPDK to enable user-level networking, a malicious OS can still tamper with IO traffic; since enclave memory cannot be the target of a DMA operation, a SafeBricks function must pull packets into normal memory that is accessible by a malicious kernel.

**NIC-accelerated applications:** By offloading functionality to a smart NIC, applications reduce packet copying between the host and NIC, and leverage optimized hardware accelerators to implement common packet-handling tasks. For example, ConsensusBox [60] and NOPaxos [70] offload distributed systems primitives like atomic broadcast. Incbricks [77] provides in-network data caching. Eris [69] performs on-NIC ordering for transactions in a distributed storage system.

To offload functions to a LiquidIO NIC, developers use programming frameworks like E3 [78], Floem [97], and iPipe [76]. $\lambda$-NIC [28] supports offloading to a Netronome NIC. However, these NICs enforce weak memory isolation, and thus cannot provide hardware-level protections against malicious functions or malicious NIC OSes (§3). $\lambda$-NIC does use compiler-emitted guards to restrict a function's memory accesses, but if those restrictions are subverted (e.g., via ROP attacks [101]), all NIC state can be compromised. These NICs also cannot provide fine-grained, side-channel-free virtualization of a NIC's resources. S-NIC achieves these goals using novel hardware-level primitives.

**Formal verification:** Recent work from the hardware community has explored whether a processor's architecture can be formally verified to be free of side channels [22, 50, 114]. S-NIC hardware is amenable to such analyses, although state-of-the-art approaches currently have practical limitations with respect to scalability and completeness. For example, InSpectre [50] introduces a formal language for modeling a CPU, and can prove whether a particular CPU design (expressed in that language) is vulnerable to side channel leakage. However, InSpectre's analyses are restricted to a single core, and do not provide security guarantees about the actual RTL implementation of a CPU.

## 7 Conclusion

S-NIC is a new design for smart NICs. Using a novel hardware architecture, S-NIC isolates functions at both the ISA level and the microarchitectural level, enforcing confidentiality, integrity, and the absence of side channels in the midst of malicious tenants and malicious datacenter providers. S-NIC provides minimal degradations in performance and total-cost-of-ownership, with only modest hardware costs. S-NIC evaluation code is available at https://github.com/SNIC-EuroSys24/SNIC.

## Acknowledgments

We thank our shepherd Meni Orenbach and the anonymous reviewers for their comments. We thank Cloudlab [35] for providing us with evaluation infrastructure. This work was supported in part by ACE, one of the seven centers in JUMP 2.0, a Semiconductor Research Corporation (SRC) program sponsored by DARPA. Yang Zhou was also supported by a Google PhD Fellowship.

## References

[1] Alfred V Aho and Margaret J Corasick. Efficient String Matching: an Aid to Bibliographic Search. *Communications of the ACM*, 18(6):333–340, 1975.

[2] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port Contention for Fun

and Profit. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 870–887, 2019.

[3] Ittai Anati, Shay Gueron, Simon Johnson, and Vincent Scarlata. Innovative Technology for CPU Based Attestation and Sealing. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[4] James W Anderson, Ryan Braud, Rishi Kapoor, George Porter, and Amin Vahdat. xOMB: Extensible Open Middleboxes with Commodity Servers. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications systems*, pages 49–60, 2012.

[5] Bilal Anwer, Theophilus Benson, Nick Feamster, and Dave Levin. Programming Slick Network Functions. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–13, 2015.

[6] ARM. ARM TrustZone. https://developer.arm.com/ip-products/security-ip/trustzone, 2020.

[7] ARM. Cortex-A9 – Arm Developer. https://developer.arm.com/ip-products/processors/cortex-a/cortex-a9, 2020.

[8] Aryaka. Cloud-First WAN: Managed SD-WAN and MPLS Alternative - Aryaka. http://www.aryaka.com/, 2020.

[9] Anonymous authors. Non-NDA discussions with Mellanox, 2019.

[10] Tom Barbette, Cyril Soldani, and Laurent Mathy. Fast Userspace Packet Processing. In *Proceedings of ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, pages 5–16, 2015.

[11] Adam Bates, Benjamin Mood, Joe Pletcher, Hannah Pruse, Masoud Valafar, and Kevin Butler. On Detecting Co-Resident Cloud Instances Using Network Flow Watermarking Techniques. *International Journal of Information Security*, 13(2):171–189, 2014.

[12] J. Bech, A. Biesheuvel, M. Brown, and D. Thompson. Implications of Meltdown and Spectre: Part 2, February 7, 2018. Linaro blog. https://www.linaro.org/blog/meltdown-spectre-2/.

[13] Theophilus Benson. In-Network Compute: Considered Armed and Dangerous. In *Proceedings of HotOS*, pages 216–224, 2019.

[14] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *Proceedings of ACM ASPLOS*, pages 26–35, 2008.

[15] Ravi Bhargava, Benjamin Serebrin, Francesco Spadini, and Srilatha Manne. Accelerating Two-Dimensional Page Walks for Virtualized Systems. In *ACM SIGOPS Operating Systems Review*, pages 26–35, 2008.

[16] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R Hower, Tushar Krishna, Somayeh Sardashti, et al. The gem5 Simulator. *ACM SIGARCH computer architecture news*, 39(2):1–7, 2011.

[17] Rick Boivie and Peter Williams. SecureBlue++: CPU Support for Secure Execution, April 12, 2013. IBM Research Report: RC25369. https://domino.research.ibm.com/library/cyberdig.nsf/papers/BE73A643EFE8274B85257B51006760C0/$File/rc25369.pdf.

[18] Ferdinand Brasser, Urs Müller, Alexandra Dmitrienko, Kari Kostiainen, Srdjan Capkun, and Ahmad-Reza Sadeghi. Software Grand Exposure: SGX Cache Attacks Are Practical. In *Proceedings USENIX Workshop on Offensive Technologies*, 2017.

[19] Anat Bremler-Barr, Yotam Harchol, and David Hay. OpenBox: a Software-Defined Framework for Developing, Deploying, and Managing Network Functions. In *Proceedings of ACM SIGCOMM*, pages 511–524, 2016.

[20] J. Van Bulck, M. Minkin, O. Weisse, D. Genkin, B. Kasikci, F. Piessens, M. Silberstein, T.F. Wenisch, Y. Yarom, and R. Strackx. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *Proceedings of USENIX Security*, pages 991–1008, 2018.

[21] J.V. Bulck, N. Weichbrodt, R. Kapitza, F. Piessens, and R. Strackx. Telling Your Secrets without Page Faults: Stealthy Page Table-based Attacks on Enclaved Execution. In *Proceedings of USENIX Security*, pages 1041–1056, 2017.

[22] Gianpiero Cabodi, Paolo Camurati, Fabrizio Finocchiaro, and Danilo Vendramietto. Model-Checking Speculation-Dependent Security Properties: Abstracting and Reducing Processor Models for Sound and Complete Verification. In *Proceedings of the International Conference on Codes, Cryptology, and Information Security*, pages 462–479, 2019.

[23] Adrian M Caulfield, Eric S Chung, Andrew Putnam, Hari Angepat, Jeremy Fowers, Michael Haselman, Stephen Heil, Matt Humphrey, Puneet Kaur, Joo-Young Kim, et al. A Cloud-Scale Acceleration Architecture. In *Proceedings of IEEE/ACM MICRO*, page 7, 2016.

[24] Center for Applied Internet Data Analysis (CAIDA). The CAIDA Anonymized Internet Traces 2016 Dataset. https://www.caida.org/data/passive/passive_2016_dataset.xml, 2016.

[25] David Champagne and Ruby B Lee. Scalable Architectural Support for Trusted Software. In *Proceedings of IEEE International Symposium on High-Performance Computer Architecture*, pages 1–12, 2010.

[26] Guoxing Chen, Sanchuan Chen, Yuan Xiao, Yinqian Zhang, Zhiqiang Lin, and Ten H Lai. SgxPectre: Stealing Intel Secrets from SGX Enclaves via Speculative Execution. In *Proceedings of IEEE European Symposium on Security and Privacy*, pages 142–157, 2019.

[27] Shuo Chen, Rui Wang, XiaoFeng Wang, and Kehuan Zhang. Side-Channel Leaks in Web Applications: A Reality Today, a Challenge Tomorrow. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 191–206, 2010.

[28] Sean Choi, Muhammad Shahbaz, Balaji Prabhakar, and Mendel Rosenblum. $\lambda$-NIC: Interactive Serverless Compute on Programmable SmartNICs. In *Proceedings of IEEE ICDCS*, pages 67–77, 2020.

[29] Cisco. Community Rulesets for Snort v3.0 and v2.9. https://www.snort.org/downloads, 2020.

[30] Comcast. NetBricks Open Source. https://github.com/williamofockham/NetBricks/tree/5e92f07410a67178fb837adf8b47b40f524ade67, 2019.

[31] Victor Costan and Srinivas Devadas. Intel SGX Explained, February 20, 2017. Cryptology ePrint Archive: Version 20170221:054353. https://eprint.iacr.org/2016/086.pdf.

[32] Victor Costan, Ilia Lebedev, and Srinivas Devadas. Sanctum: Minimal Hardware Extensions for Strong Software Isolation. In *Proceedings of USENIX Security*, pages 857–874, 2016.

[33] Peter W Deutsch, Yuheng Yang, Thomas Bourgeat, Jules Drean, Joel S Emer, and Mengjia Yan. DAGguise: Mitigating Memory Timing Side Channels. In *Proceedings of ACM ASPLOS*, pages 329–343, 2022.

[34] Whitfield Diffie and Martin E. Hellman. New Directions in Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[35] Dmitry Duplyakin, Robert Ricci, Aleksander Maricq, Gary Wong, Jonathon Duerig, Eric Eide, Leigh Stoller, Mike Hibler, David Johnson, Kirk Webb, et al. The Design and Operation of CloudLab. In *Proceedings of USENIX ATC*, pages 1–14, 2019.

[36] Eddie Kohler. MazuNAT. https://github.com/kohler/click/blob/master/conf/mazu-nat.click, 2011.

[37] Daniel E Eisenbud, Cheng Yi, Carlo Contavalli, Cody Smith, Roman Kononov, Eric Mann-Hielscher, Ardas Cilingiroglu, Bin Cheyney, Wentao Shang, and Jinnah Dylan Hosein. Maglev: A Fast and Reliable Software Network Load Balancer. In *Proceedings of USENIX NSDI*, pages 523–535, 2016.

[38] Reouven Elbaz, David Champagne, Catherine Gebotys, Ruby B Lee, Nachiketh Potlapally, and Lionel Torres. Hardware Mechanisms for Memory Authentication: A Survey of Existing Techniques and Engines. In *Transactions on Computational Science IV*, pages 1–22. 2009.

[39] Emerging Threats Site. DPI Rulesets. https://rules.emergingthreats.net/open/, 2020.

[40] Emerging Threats Site. Firewall Rulesets. https://rules.emergingthreats.net/fwrules/, 2020.

[41] Haggai Eran, Lior Zeno, Maroun Tork, Gabi Malka, and Mark Silberstein. NICA: An Infrastructure for Inline Acceleration of Network Applications. In *Proceedings of USENIX ATC*, pages 345–362, 2019.

[42] D. Evtyushkin, R. Riley, N. Abu-Ghazaleh, and D. Ponomarev. BranchScope: A New Side-Channel Attack on Directional Branch Predictor. In *Proceedings of ACM ASPLOS*, pages 693–707, 2018.

[43] Dmitry Evtyushkin, Jesse Elwell, Meltem Ozsoy, Dmitry Ponomarev, Nael Abu Ghazaleh, and Ryan Riley. Iso-X: A Flexible Architecture for Hardware-Managed Isolated Execution. In *Proceedings of IEEE/ACM MICRO*, pages 190–202, 2014.

[44] Yangchun Fu, Erick Bauman, Raul Quinonez, and Zhiqiang Lin. SGX-LPAD: Thwarting Controlled Side Channel Attacks via Enclave Verifiable Page Faults. In *Proceedings of International Symposium on Research in Attacks, Intrusions, and Defenses*, pages 357–380, 2017.

[45] GlobeNewswire. Stingray SmartNIC Powering Baidu Cloud. https://www.globenewswire.com/news-release/2020/03/31/2009195/0/en/Broadcom-Stingray-SmartNIC-Accelerates-Baidu-Cloud-Services.html, 2020.

[46] Johannes Götzfried, Moritz Eckert, Sebastian Schinzel, and Tilo Müller. Cache Attacks on Intel SGX. In *Proceedings of European Workshop on Systems Security*, pages 1–6, 2017.

[47] Stewart Grant, Anil Yelam, Maxwell Bland, and Alex C Snoeren. Smartnic performance isolation with fairnic: Programmable networking for the cloud. In *Proceedings of ACM SIGCOMM*, pages 681–693, 2020.

[48] B. Gras, K. Razavi, H. Bos, and C. Giuffrida. Translation Leak-aside Buffer: Defeating Cache Side-channel Protections with TLB Attacks. In *Proceedings of USENIX Security*, pages 995–972, 2018.

[49] A. Greenberg, J. R. Hamilton, N. Jain, S. Kandula, C. Kim, P. Lahiri, D. A. Maltz, P. Patel, and S. Sengupta. VL2: A Scalable and Flexible Data Center Network. In *Proceedings of ACM SIGCOMM*, pages 51–62, 2009.

[50] Roberto Guanciale, Musard Balliu, and Mads Dam. InSpectre: Breaking and Fixing Microarchitectural Vulnerabilities by Formal Analysis. In *Proceedings of ACM CCS*, pages 1853–1869, 2020.

[51] Shay Gueron. A Memory Encryption Engine Suitable for General Purpose Processors, February 25, 2016. Cryptology ePrint Archive: Version 20160225:211316. https://eprint.iacr.org/2016/204.pdf.

[52] Pankaj Gupta, Steven Lin, and Nick McKeown. Routing Lookups in Hardware at Memory Access Speeds. In *Proceedings of IEEE INFOCOM*, pages 1240–1247, 1998.

[53] Michio Honda, Felipe Huici, Giuseppe Lettieri, and Luigi Rizzo. mSwitch: a Highly-Scalable, Modular Software Switch. In *Proceedings of ACM SIGCOMM Symposium on Software Defined Networking Research*, pages 1–13, 2015.

[54] Jinho Hwang, K K_ Ramakrishnan, and Timothy Wood. NetVM: High Performance and Flexible Networking Using Virtualization on Commodity Platforms. *IEEE Transactions on Network and Service Management*, 12(1):34–47, 2015.

[55] IETF. RFC 7348: Virtual eXtensible Local Area Network (VXLAN): A Framework for Overlaying Virtualized Layer 2 Networks over Layer 3 Networks. https://tools.ietf.org/html/rfc7348, 2014.

[56] Intel. PCI-SIG SR-IOV Primer: An Introduction to SR-IOV Technology (Intel LAN Access Division). https://www.intel.com/content/dam/doc/application-note/pci-sig-sr-iov-primer-sr-iov-technology-paper.pdf, 2011.

[57] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf, 2015.

[58] Intel. Resources and Response to Side Channel L1 Terminal Fault. https://www.intel.com/content/www/us/en/architecture-and-technology/l1tf.html, 2018.

[59] Intel. Intel Xeon Processor E5-2680 v3 (30M Cache, 2.50 GHz) Product Specifications. https://ark.intel.com/content/www/us/en/ark/products/81908/intel-xeon-processor-e5-2680-v3-30m-cache-2-50-ghz.html?q=E5-2680v3, 2020.

[60] Zsolt István, David Sidler, Gustavo Alonso, and Marko Vukolic. Consensus in a Box: Inexpensive Coordination in Hardware. In *Proceedings of USENIX NSDI*, pages 425–438, 2016.

[61] Murad Kablan, Azzam Alsudais, Eric Keller, and Franck Le. Stateless Network Functions: Breaking the Tight Coupling of State and Processing. In *Proceedings of USENIX NSDI*, pages 97–112, 2017.

[62] Georgios P Katsikas, Tom Barbette, Dejan Kostic, Rebecca Steinert, and Gerald Q Maguire Jr. Metron: NFV Service Chains at the True Speed of the Underlying Hardware. In *Proceedings of USENIX NSDI*, pages 171–186, 2018.

[63] Johannes Krude, Jaco Hofmann, Matthias Eichholz, Klaus Wehrle, Andreas Koch, and Mira Mezini. Online Reprogrammable Multi-Tenant Switches. In *Proceedings of Workshop on Emerging in-Network Computing Paradigms*, pages 1–8, 2019.

[64] Chang Lan, Justine Sherry, Raluca Ada Popa, Sylvia Ratnasamy, and Zhi Liu. Embark: Securely Outsourcing Middleboxes to the Cloud. In *Proceedings of USENIX NSDI*, pages 255–273, 2016.

[65] Yanfang Le, Hyunseok Chang, Sarit Mukherjee, Limin Wang, Aditya Akella, Michael M Swift, and TV Lakshman. UNO: Unifying Host and Smart NIC Offload for Flexible Packet Processing. In *Proceedings of ACM SoCC*, pages 506–519, 2017.

[66] Ruby B Lee, Peter CS Kwan, John P McGregor, Jeffrey Dwoskin, and Zhenghong Wang. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of IEEE International Symposium on Computer Architecture*, pages 2–13, 2005.

[67] Michael Lescisin and Qusay Mahmoud. Tools for Active and Passive Network Side-Channel Detection for Web Applications. In *Proceedings of USENIX Workshop on Offensive Technologies*, 2018.

[68] Huaicheng Li, Mingzhe Hao, Stanko Novakovic, Vaibhav Gogte, Sriram Govindan, Dan RK Ports, Irene Zhang, Ricardo Bianchini, Haryadi S Gunawi, and Anirudh Badam. LeapIO: Efficient and Portable Virtual NVMe Storage on ARM SoCs. In *Proceedings of ACM ASPLOS*, pages 591–605, 2020.

[69] Jialin Li, Ellis Michael, and Dan RK Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of ACM SOSP*, pages 104–120, 2017.

[70] Jialin Li, Ellis Michael, Naveen Kr Sharma, Adriana Szekeres, and Dan RK Ports. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *Proceedings of USENIX OSDI*, pages 467–483, 2016.

[71] Sheng Li, Jung Ho Ahn, Richard D Strong, Jay B Brockman, Dean M Tullsen, and Norman P Jouppi. McPAT: an Integrated Power, Area, and Timing Modeling Framework for Multicore and Manycore Architectures. In *Proceedings of IEEE/ACM MICRO*, pages 469–480, 2009.

[72] David Lie, Chandramohan Thekkath, Mark Mitchell, Patrick Lincoln, Dan Boneh, John Mitchell, and Mark Horowitz. Architectural Support for Copy and Tamper Resistant Software. *Acm Sigplan Notices*, 35(11):168–177, 2000.

[73] Jiaxin Lin, Kiran Patel, Brent E Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *Proceedings of USENIX OSDI*, pages 243–259, 2020.

[74] Linaro Limited. Open Portable Trusted Execution Environment. https://www.op-tee.org/, 2020.

[75] M. Lipp, D. Gruss, R. Spreitzer, C. Maurice, and S. Mangard. ARMageddon: Cache Attacks on Mobile Devices. In *Proceedings of USENIX Security*, pages 549–564, 2016.

[76] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. Offloading Distributed Applications onto SmartNICs Using iPipe. In *Proceedings of ACM SIGCOMM*, pages 318–333, 2019.

[77] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. IncBricks: Toward In-Network Computation with an In-Network Cache. In *Proceedings of ACM ASPLOS*, pages 795–809, 2017.

[78] Ming Liu, Simon Peter, Arvind Krishnamurthy, and Phitchaya Mangpo Phothilimthana. E3: Energy-Efficient Microservices on SmartNIC-Accelerated Servers. In *Proceedings of USENIX ATC*, pages 363–378, 2019.

[79] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of ACM SIGCOMM*, pages 101–114, 2016.

[80] Joao Martins, Mohamed Ahmed, Costin Raiciu, Vladimir Olteanu, Michio Honda, Roberto Bifulco, and Felipe Huici. ClickOS and the Art of Network Function Virtualization. In *Proceedings of USENIX NSDI*, pages 459–473, 2014.

[81] Marvell. Marvell/Cavium LiquidIO Smart NICs. https://www.marvell.com/ethernet-adapters-and-controllers/liquidio-smart-nics/, 2020.

[82] Frank McKeen, Ilya Alexandrovich, Alex Berenzon, Carlos V. Rozas, Hisham Shafi, Vedvyas Shanbhogue, and Uday R. Savagaonkar. Innovative Instructions and Software Model for Isolated Execution. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy*, 2013.

[83] J. Mickens, E. B. Nightingale, J. Elson, D. Gehring, B. Fan, A. Kadav, V. Chidambaram, and O. Khan. Blizzard: Fast, Cloud-scale Block Storage for Cloud-oblivious Applications. In *Proceedings of USENIX NSDI*, pages 257–273, 2014.

[84] Saeid Mofrad, Fengwei Zhang, Shiyong Lu, and Weidong Shi. A Comparison Study of Intel SGX and AMD Memory Encryption Technology. In *Proceedings of International Workshop on Hardware and Architectural Support for Security and Privacy*, pages 1–8, 2018.

[85] R. N. Mysore, A. Pamboris, N. Farrington, N. Huang, P. Miri, S. Radhakrishnan, V. Subramanya, and A. Vahdat. PortLand: A Scalable Fault-Tolerant Layer 2 Data Center Network Fabric. In *Proceedings of ACM SIGCOMM*, pages 39–50, 2009.

[86] D. Naylor, K. Schomp, M. Varvello, I. Leontiadis, J. Blackburn, D.R. López, K. Papagiannaki, R. Rodriguez, and P. Steenkiste. Multi-Context TLS (mcTLS): Enabling Secure In-Network Functionality in TLS. In *Proceedings of ACM SIGCOMM*, pages 199–212, 2015.

[87] Netronome. Netronome Agilio LX Smart NICs. https://www.netronome.com/products/agilio-lx/, 2020.

[88] Palo Alto Networks. Global Cybersecurity Leader - Palo Alto Networks. https://www.paloaltonetworks.com/, 2020.

[89] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone Explained: Architectural Features and Use Cases. In *Proceedings of IEEE International Conference on Collaboration and Internet Computing*, pages 445–451, 2016.

[90] E. B. Nightingale, J. Elson, J. Fan, O. Hofmann, J. Howell, and Y. Suzue. Flat Datacenter Storage. In *Proceedings of USENIX OSDI*, pages 1–15, 2012.

[91] Oleskii Oleksenko, Bodhan Trach, Robert Krahn, Andre Martin, Christof Fetzer, and Mark Silberstein. Varys: Protecting SGX Enclaves from Practical Side-channel Attacks. In *Proceedings of USENIX ATC*, pages 227–239, 2018.

[92] Emmanuel Owusu, Jorge Guajardo, Jonathan McCune, Jim Newsome, Adrian Perrig, and Amit Vasudevan. OASIS: On Achieving a Sanctuary for Integrity and Secrecy on Untrusted Platforms. In *Proceedings of ACM SIGSAC conference on Computer & communications security*, pages 13–24, 2013.

[93] D. Page. Partitioned Cache Architecture as a Side-Channel Defence Mechanism, August 22, 2005. Cryptology ePrint Archive: Version 20050825:073958. http://eprint.iacr.org/2005/280.pdf.

[94] Aurojit Panda, Sangjin Han, Keon Jang, Melvin Walls, Sylvia Ratnasamy, and Scott Shenker. NetBricks: Taking the V out of NFV. In *Proceedings of USENIX OSDI*, pages 203–216, 2016.

[95] B. Parno, J.M. McCune, and A. Perrig. Bootstrapping Trust in Modern Computers, 2011. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/BootstrappingTrustBook.pdf.

[96] Ben Pfaff, Justin Pettit, Teemu Koponen, Ethan Jackson, Andy Zhou, Jarno Rajahalme, Jesse Gross, Alex Wang, Joe Stringer, Pravin Shelar, et al. The Design and Implementation of Open vSwitch. In *Proceedings of USENIX NSDI*, pages 117–130, 2015.

[97] Phitchaya Mangpo Phothilimthana, Ming Liu, Antoine Kaufmann, Simon Peter, Rastislav Bodik, and Thomas Anderson. Floem: a Programming System for NIC-Accelerated Network Applications. In *Proceedings of USENIX OSDI*, pages 663–679, 2018.

[98] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. SafeBricks: Shielding Network Functions in the Cloud. In *Proceedings of USENIX NSDI*, pages 201–216, 2018.

[99] Moinuddin K Qureshi and Yale N Patt. Utility-Based Cache Partitioning: A Low-Overhead, High-Performance, Runtime Mechanism to Partition Shared Caches. In *Proceedings of IEEE/ACM MICRO*, pages 423–432, 2006.

[100] Kaushik Kumar Ram, Alan L Cox, Mehul Chadha, and Scott Rixner. Hyper-switch: A Scalable Software Virtual Switching Architecture. In *Proceedings of USENIX ATC*, pages 13–24, 2013.

[101] Ryan Roemer, Erik Buchanan, Hovav Shacham, and Stefan Savage. Return-Oriented Programming: Systems, Languages, and Applications. *ACM Transactions on Information and System Security*, 15(1):1–34, 2012.

[102] Vyas Sekar, Norbert Egi, Sylvia Ratnasamy, Michael K Reiter, and Guangyu Shi. Design and Implementation of a Consolidated Middlebox Architecture. In *Proceedings of USENIX NSDI*, pages 323–336, 2012.

[103] Ali Shafiee, Akhila Gundu, Manjunath Shevgoor, Rajeev Balasubramonian, and Mohit Tiwari. Avoiding Information Leakage in the Memory Controller with Fixed Service Policies. In *Proceedings of IEEE/ACM MICRO*, pages 89–101, 2015.

[104] D. Shen. Exploiting Trustzone on Android. In *Black Hat*, August 2015. https://www.blackhat.com/docs/us-15/materials/us-15-Shen-Attacking-Your-Trusted-Core-Exploiting-Trustzone-On-Android-wp.pdf.

[105] Justine Sherry, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. Blindbox: Deep Packet Inspection over Encrypted Traffic. In *Proceedings of ACM SIGCOMM*, pages 213–226, 2015.

[106] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. In *Proceedings of NDSS*, 2017.

[107] Vishal Shrivastav. Fast, Scalable, and Programmable Packet Scheduler in Hardware. In *Proceedings of ACM SIGCOMM*, pages 367–379, 2019.

[108] Arjun Singhvi, Aditya Akella, Dan Gibson, Thomas F Wenisch, Monica Wong-Chan, Sean Clark, Milo MK Martin, Moray McLaren, Prashant Chandra, Rob Cauble, et al. 1RMA: Re-envisioning Remote Memory Access for Multi-tenant Datacenters. In *Proceedings of ACM SIGCOMM*, pages 708–721, 2020.

[109] Site Selection Group. Power in the Data Center and its Cost Across the U.S. https://info.siteselectiongroup.com/blog/power-in-the-data-center-and-its-costs-across-the-united-states, 2017.

[110] Brent Stephens, Aditya Akella, and Michael Swift. Loom: Flexible and Efficient NIC Packet Scheduling. In *Proceedings of USENIX NSDI*, pages 33–46, 2019.

[111] G Edward Suh, Dwaine Clarke, Blaise Gassend, Marten Van Dijk, and Srinivas Devadas. AEGIS: Architecture for Tamper-Evident and Tamper-Resistant Processing. In *Proceedings of ACM International Conference on Supercomputing*, pages 357–368, 2014.

[112] Jakub Szefer and Ruby B Lee. Architectural Support for Hypervisor-Secure Virtualization. *ACM SIGPLAN Notices*, 47(4):437–450, 2012.

[113] The UCSB iCTF. Network Traces Collected During the 2010 iCTF. https://ictf.cs.ucsb.edu/archive/ictf_2010.html, 2010.

[114] Caroline Trippel, Daniel Lustig, and Margaret Martonosi. CheckMate: Automated Synthesis of Hardware Exploits and Security Litmus Tests. In *Proceedings of IEEE/ACM MICRO*, pages 947–960, 2018.

[115] Trusted Computing Group. TCG Infrastructure Working Group Architecture Part II: Integrity Management, 2006. Specification Version 1.0, Revision 1.0.

[116] Trusted Computing Group. TCG Attestation PTS Protocol: Binding to TNC IF-M, 2011. Specification Version 1.0, Revision 28.

[117] VMware. Network Functions Virtualization (NFV) - vCloud NFV. https://www.vmware.com/products/network-functions-virtualization.html, 2020.

[118] Wenhao Wang, Guoxing Chen, Xiaorui Pan, Yinqian Zhang, XiaoFeng Wang, Vincent Bindschaedler, Haixu Tang, and Carl A Gunter. Leaky Cauldron on the Dark Land: Understanding Memory Side-Channel Jazards in SGX. In *Proceedings of ACM CCS*, pages 2421–2434, 2017.

[119] Yao Wang, Andrew Ferraiuolo, and G Edward Suh. Timing Channel Protection For a Shared Memory Controller. In *Proceedings of IEEE International Symposium on High Performance Computer Architecture*, pages 225–236, 2014.

[120] Yao Wang, Andrew Ferraiuolo, Danfeng Zhang, Andrew C Myers, and G Edward Suh. SecDCP: Secure Dynamic Cache Partitioning for Efficient Timing Channel Protection. In *Proceedings of Design Automation Conference*, pages 1–6, 2016.

[121] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-Channel Attacks: Deterministic Side Channels for Untrusted Operating Systems. In *Proceedings of IEEE Symposium on Security and Privacy*, pages 640–656, 2015.

[122] Yao Wang. Open Source for Temporal Partitioning Bus Arbitration. https://github.com/xiaoyaozi5566/GEM5_DRAMSim2, 2014.

[123] Ning Zhang, Kun Sun, Deborah Shands, Wenjing Lou, and Y Thomas Hou. TruSpy: Cache Side-Channel Information Leakage from the Secure World on ARM Devices, October 10, 2016. Cryptology ePrint Archive: Version 20161015:190703. https://eprint.iacr.org/2016/980.pdf.

[124] Wei Zhang, Guyue Liu, Wenhui Zhang, Neel Shah, Phillip Lopreiato, Gregoire Todeschi, KK Ramakrishnan, and Timothy Wood. OpenNetVM: a Platform for High Performance Network Service Chains. In *Proceedings of Workshop on Hot topics in Middleboxes and Network Function Virtualization*, pages 26–31, 2016.

[125] Ziqiao Zhou, Yizhou Shan, Weidong Cui, Xinyang Ge, Marcus Peinado, and Andrew Baumann. Core Slicing: Closing the Gap Between Leaky Confidential VMs and Bare-metal Cloud. In *Proceedings of USENIX OSDI*, pages 247–267, 2023.

[126] Noa Zilberman, Yury Audzevich, Georgina Kalogeridou, Neelakandan Manihatty-Bojan, Jingyun Zhang, and Andrew Moore. NetFPGA: Rapid Prototyping of Networking Devices in Open Source. *ACM SIGCOMM Computer Communication Review*, 45(4):363–364, 2015.

[127] Zscaler. Zscaler Cloud Security - Secure Your Digital Transformation. https://www.zscaler.com/, 2020.

# APPENDIX

## A S-NIC's Attestation Protocol

Attestation [3, 95, 115, 116] enables a machine to prove to a remote party that the machine is running a specific piece of software. Attestation requires the trusted hardware to sign statements about the software that has been loaded. At manufacturing time, an S-NIC receives an asymmetric key pair. This key pair, called the endorsement key pair (*EK*), is burned into the NIC hardware, along with a certificate for the public half of the *EK*. The NIC never reveals the private half to external parties; the certificate for the public part is signed by the NIC vendor. After a reboot, the NIC generates a random asymmetric key pair known as the attestation key pair (*AK*). The NIC stores the private half in a private on-NIC register, and signs the public half with the *EK*.

Suppose that a remote party wishes to verify that a NIC is running function $F$. The remote party (i.e., the *verifier*) and $F$ engage in the following attestation protocol, which is based on the classic Diffie-Hellman exchange:

- The verifier sends a hello message to the NIC. The message contains a verifier-choosen nonce $n$.
- $F$ generates a random value $x$, and calculates $g^x \mod p$, where $g$ and $p$ are the public Diffie-Hellman parameters. $F$ then invokes the nf_attest instruction, passing a pointer to a memory buffer that contains $g$, $p$, $n$, and $g^x \mod p$. The nf_attest instruction uses $AK_{priv}$ to sign the hash of $F$'s inital state concatenated with $g$, $p$, $n$, and $g^x \mod p$.
- $F$ sends a four-part message to the verifier. The first part contains $g, p, n, g^x \mod p$, and the hash of $F$'s initial state. The second part contains the hardware-generated signature over that state. The third part contains the $AK_{pub}$ signed by $EK_{priv}$. The final part contains the certificate for $EK_{pub}$; remember that this certificate is signed by the NIC vendor.
- The verifier receives the response, and checks whether the hash corresponds to $F$'s hash, and whether the signature was generated by a certified S-NIC. The verifier also ensures that the signature covers the appropriate nonce; this prevents replay attacks. If all of the checks succeed, the verifier picks a random number $y$, calculates $g^y \mod p$, and sends the latter value to $F$.
- Both sides can then compute the shared symmetric key $g^{xy} \mod p$.

## B Memory Profiling Results

We define the *memory utilization ratio (MUR)* as the ratio between the memory size allocated by S-NIC and the actual memory usage by the NF. Table 6 shows the memory usage profiles for six NFs. For the Monitor function, we recorded the maximum memory usage when measuring every five-minute CAIDA trace (as in the UnivMon paper [79]). The memory usages of the other functions are bounded, regardless of the number of flows. For the specific parameter settings (e.g., number of rules, number of cached flows) of each NF, please refer to §5.1. Table 7 shows the memory usage profiles for three hardware accelerators.

| | (Maximum) memory usage (MB) | | | | | Estimated # of TLB entries | | | MUR |
|---|---|---|---|---|---|---|---|---|---|
| | Text | Data | Code | Heap&stack | Total | Equal | Flex-low | Flex-high | |
| FW | 0.87 | 0.08 | 2.50 | 13.75 | 17.20 | 11 | 34 | 11 | 100.0% |
| DPI | 1.34 | 0.56 | 2.59 | 46.65 | 51.14 | 28 | 51 | 13 | 100.0% |
| NAT | 0.86 | 0.05 | 2.49 | 40.48 | 43.88 | 25 | 37 | 10 | 72.3% |
| LB | 0.86 | 0.05 | 2.49 | 10.40 | 13.80 | 10 | 22 | 10 | 30.2% |
| LPM | 0.86 | 0.06 | 2.51 | 64.90 | 68.33 | 37 | 23 | 7 | 100.0% |
| Mon. | 0.85 | 0.05 | 2.48 | 357.15 | 360.54 | 183 | 46 | 12 | 68.3% |

**Table 6.** Memory usage profiles for six NFs. We calculate the number of TLB entries based on three page-size settings: the *Equal* setting which only has 2MB pages; the *Flex-low* setting with 128KB, 2MB, and 64MB pages; and the *Flex-high* setting with 2MB, 32MB and 128MB pages. When allocating pages for a function's code, static data, heap, and stack regions, we try to minimize the amount of wasted memory.

| | Memory usage (Bytes) | | | | | | | | | | Est. # of |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | IQ | PktDB | PktB | ResB | ParaB | OutB | SGP | Graph | Dict | Total | TLB entries |
| DPI | 256K | 128K | 2M | 2M | 256K | N/A | N/A | 97.28M | N/A | 101.90M | 54 |
| ZIP | 64K | 128K | 2M | 24K | N/A | 2M | 128M | N/A | 32K | 132.24M | 70 |
| RAID | 4M | 128K | 2M | N/A | N/A | 2M | N/A | N/A | N/A | 8.13M | 5 |

**Table 7.** Memory usage profiles for three hardware accelerators. IQ = instruction queue, PktDB = packet descriptor buffers, PktB = packet buffers, ResB = result buffers, ParaB = parameter buffers, OutB = output buffers, SGP = scatter-gather-pointer buffers, Graph = the state machine graph for rules in DPI, Dict = the dictionary used in the ZIP data compressor.

## C Micro Benchmarks

We conducted several micro-benchmark experiments using a 10G Marvell NIC with 16 1.2GHz MIPS cores.

**Instruction execution latency:** To simulate the latency of nf_launch, nf_attest, and nf_destroy, we wrote code for the Marvell NIC to simulate each instruction's activities. We used the NIC's security co-processor to accelerate cryptographic operations. Figure 6 shows the results.

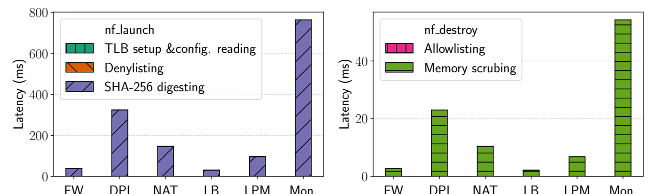



**Figure 6.** Instruction execution latency. For all functions, TLB setup and configuration reading in nf_launch takes almost the same amount of time: 0.0196*ms* on average across NF types. Denylisting and allowlisting costs are also similar: 0.0044*ms* and 0.0038*ms*.

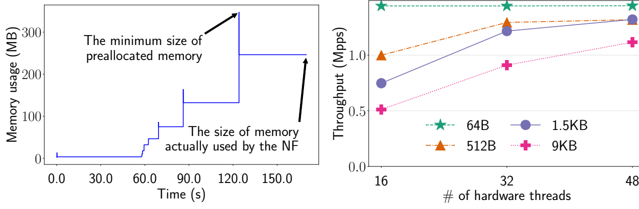



**Figure 7.** Time series of the memory usage of Monitor.

**Figure 8.** DPI performance vs. cluster size and frame size.

| | FW | DPI | NAT | LB | LPM | Mon. |
|---|---|---|---|---|---|---|
| Mem. prealloc. (MB) | 17.20 | 51.14 | 43.88 | 13.80 | 68.33 | 360.54 |
| Mem. used (MB) | 17.20 | 51.14 | 31.72 | 4.16 | 68.33 | 246.31 |
| MURs | 100.0% | 100.0% | 72.3% | 30.2% | 100.0% | 68.3% |

**Table 8.** Memory utilization ratios (MURs).

nf_attest took roughly 5.6*ms* and was independent of a function's size. For the other instructions, the latency was dominated by memory-related operations, and thus was sensitive to function size. For example, for nf_launch, the SHA digesting of function memory took 29.62*ms* for LB, and 763.52*ms* for Monitor. nf_attest spends 5.596*ms* on RSA signing and 0.004*ms* on SHA digesting. nf_destroy took 2.11–54.23*ms*; memory scrubbing takes 99.99% of the time.

**The cost of fixed memory size.** S-NIC only allocates a fixed amount of memory at the launch time of an NF, but does not have an OS that can dynamically allocate memory during the lifetime of the NF. This means that we may waste some memory due to the preallocation. For example, Figure 7 shows a time series of the memory usage of Monitor (using a five-minute CAIDA trace). There are several spikes of the memory usage which are caused by DPDK hugepage initialization and multiple HashMap resizings. Hugepage initialization is expensive because DPDK allocates a temporary normal memory block for storing the hugepage data, and then writes all that data into the hugepage memory. In total, S-NIC has to allocate 360.54MB at a minimum, but a dynamic allocation strategy would only need 246.31MB to handle the steady-state memory consumption of the NF.

Table 8 shows the memory utilization ratios for six NFs. For Firewall, DPI, and LPM, preallocation in S-NIC does not waste any memory. For NAT and Monitor, preallocation wastes around a third of the memory due to HashMap resizing. For LB, nearly two-thirds of the allocated memory is wasted because of a large amount of temporary memory allocated during the DPDK initialization and a relatively small amount of steady-state memory required by the packet processing in LB. Overall, these results suggest the memory savings that would arise from NF-optimized versions of DPDK and Rust's standard-library data structures.

**DPI thread clustering.** S-NIC groups an accelerator's hardware threads into allocatable clusters with different sizes. We now study how to properly partition and allocate these clusters to NFs. Figure 8 shows how DPI performance changes for differently-sized input packets and thread clusters. We show results for cluster sizes of 16, 32, and 48 because current NICs only support clustering threads at a granularity of 16 threads. To stress-test the DPI accelerator, we use it to process large packets which are randomly generated on 16 programmable cores without IPSec. Note that 1.5KB is the maximum size of a standard Ethernet frame, while 9KB is the maximum size of a jumbo frame. The high-level takeaway from Figure 8 is that, as packet sizes grow, the per-packet processing costs increase and a function benefits from access to more hardware threads.