SmartWatch: Accurate Traffic Analysis and Flow-state Tracking for Intrusion Prevention using SmartNICs

Sourav Panda[‡], Yixiao Feng[†], Sameer G Kulkarni[★], K. K. Ramakrishnan[‡], Nick Duffield[†], Laxmi N. Bhuyan[‡]
[‡]University of California, Riverside, [†]Texas A&M University, [★]Indian Institute of Technology, Gandhinagar

ABSTRACT

Despite advances in network security, attacks targeting mission critical systems and applications remain a significant problem for network and datacenter providers. Existing telemetry platforms detect volumetric attacks at terabit scales using approximation techniques and coarse grain analysis. However, the prevalence of low and slow attacks that require very little bandwidth, makes flow-state tracking critical to overall attack mitigation. Traffic queries deployed on network switches are often limited by hardware constraints, preventing them from carrying out flow tracking features required to detect stealthy attacks. Such attacks can go undetected in the midst of high traffic volumes.

We design SmartWatch, a novel flow state tracking and flow logging system at line rate, using SmartNICs to optimize performance and simultaneously detect a number of stealthy attacks. SmartWatch leverages advances in switch based network telemetry platforms to process the bulk of the traffic and only forward suspicious traffic subsets to the SmartNIC. The programmable network switches perform coarse-grained traffic analysis while the SmartNIC conducts the finer-grained analysis which involves additional processing of the packet as a 'bump-in-the-wire'. A control loop between the SmartNIC and programmable switch tunes the queries performed in the switch to direct the most appropriate traffic subset to the SmartNIC. SmartWatch's cooperative monitoring approach yields 2.39 times better detection rate compared to existing platforms deployed on programmable switches. SmartWatch can detect covert timing channels and perform website fingerprinting more efficiently compared to standalone programmable switch solutions, relieving switch memory and control-plane processor resources. Compared to host-based approaches, SmartWatch can reduce the packet processing latency by 72.32%.

CCS CONCEPTS

• **Networks** → **Network monitoring**; **Programmable networks**.

KEYWORDS

intrusion prevention, SmartNICs, programmable switches

ACM Reference Format:
Sourav Panda, Yixiao Feng, Sameer G Kulkarni, K. K. Ramakrishnan, Nick Duffield, Laxmi N. Bhuyan. 2021. SmartWatch: Accurate Traffic Analysis and Flow-state Tracking for Intrusion Prevention using SmartNICs. In *The 17th International Conference on emerging Networking EXperiments and Technologies (CoNEXT '21), December 7–10, 2021, Virtual Event, Germany.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3485983.3494861

1 INTRODUCTION

Network-borne attacks that aim to disrupt mission-critical systems and applications are a persistent problem for both network and datacenter (DC) operators. To counter threats, operators must deploy infrastructure to monitor systems and network traffic, driving analyses to detect anomalies and specific attacks in a timely manner. The infrastructure may directly protect against some attacks or provide alerts that trigger intervention, either automated or human, to block or ameliorate the impact of those attacks.

The main challenge addressed by our work is on designing a cost-efficient traffic monitoring and analysis infrastructure that can detect a range of network attacks within a high-rate traffic stream. Systems must detect not only relatively crude volumetric attacks that overwhelm the network through sustained network activity (e.g., SYN flooding), but also more sophisticated attacks that probe for system weaknesses (*e.g.,* SSH Brute forcing, port scan [55]), or attacks that exploit protocol dynamics (*e.g.,* low-rate TCP attacks [60]). We seek to design a framework that can:

- detect low and slow attacks using state tracking;
- accurately track flow-state changes caused by each and every packet to ensure attacks cannot bypass the monitor;
- scale to monitor Terabit scale traffic by cooperative monitoring using multiple components including programmable switches, sNICs and hosts.
- provide flow-logging for rapid volumetric attack detection as well as comprehensive inspection of all flows offline;
- support a large number of monitoring features running simultaneously on the platform.

A number of different algorithms and monitoring platforms have been designed in the past to address this. Programmable switches have been used for telemetry queries at Terabit scale [39, 50] while SmartNICs (sNIC) enable end-hosts to scale to more modest, 40/100 Gbps, rates[76, 91]. Since sNICs support more operations [14] than switches, we use them for stateful packet processing, to complement the coarse-grained query processing of programmable switches. sNICs have become critical components of DC operations due to the performance boost they provide for the tasks offloaded from server CPUs[48]. We create a network monitoring system that leverages the sNIC capabilities of programmability, co-designed the scalability of programmable switches and flexibility of host-based CPU processing. Programmable switches are used to forward just the 'right' traffic subset for fine-grained sNIC-based monitoring.

This paper proposes SmartWatch, a network monitoring platform architecture, comprising a commodity-class host and sNIC, working cooperatively with programmable switches. SmartWatch makes the following contributions:

- To detect or prevent stealthy attacks such as low-rate port scans, SmartWatch performs lossless state-tracking and flow logging. SmartWatch yields 2.39 times better detection rate compared to existing programmable switch based platforms. Compared to host-based approaches, SmartWatch reduces packet processing latency by 72.32%.

- SmartWatch works cooperatively with P4 switches for network-based monitoring. SmartWatch uses iterative-refinement between the programmable switch and sNIC to judiciously use switch resources while operating at Terabit line rates. Switches direct the right traffic subset to the sNIC for processing. Less than 16% of packets processed by the sNIC go to the host.
- SmartWatch reduces the memory requirements of programmable switches for monitoring, thus allowing it to be used for common data center operations. This is done by running coarse grained queries in the switch. Fine grained processing is done by the sNIC-host subsystem.

2 OVERVIEW

We discuss the typical needs of network monitoring and the approach taken by existing platforms. Then we describe the key design principles behind SmartWatch.

2.1 Requirements for Monitoring

2.1.1 Terabit-Scale Traffic. In this work, we refer to a programmable switch that can run telemetry queries written in P4 as a P4Switch. Examples include Intel/Barefoot Tofino switches[7]. The P4Switch enables examining traffic at intensities much higher than would be possible with sNIC or Host alone [76]. A P4Switch's forwarding ASICs are able to quickly forward and perform simple computations on packets at line-rate, thus enabling the analysis of billions of packets at the Tbps rates[33]. But, they have accompanying memory and processing constraints, that limit the ability to do all the monitoring on the P4switch alone. Impediments to fine-grained traffic analysis are described in § 2.2.1.

Table 1: Slow Attacks requiring flow-state tracking

Attack	Challenges
SSH Brute Forcing	SSH connections are encrypted Detector requires conn-attempt outcome[53], which is determined heuristically using protocol state transitions and traffic volume[6].
Stealthy Port Scan	Detector probes whether conn-attempt from a remote node elicits suitable response from local nodes[55].
Forged TCP RST (In-sequence)	Detector identifies race conditions between RST packets and in-flight data packets[82].

2.1.2 Fine Grained Traffic Analysis. Network traffic monitoring has been widely explored [40, 44, 46, 59, 61, 63, 64, 68, 71, 74, 80, 85, 86]. Based on these, we first summarize the key requirements for network monitoring and then explain how fine-grained analysis is critical for effective monitoring.

Stateful Packet Processing: Low and slow attacks such as *in-sequence forged TCP RSTs* require state to be maintained to detect them. The relative timing between a potentially forged TCP RST packet and in-flight data packets must be measured [82]. Neither switch queries [39, 50] nor Sketches[59] can effectively detect such attacks due to hardware constraints and the need for non-volumetric, stateful detection.

Flow Logging: Flow logging maintains an accurate count of every packet of the flow received by SmartWatch, in the Flowcache, and stores the connection's 5-tuple, packet count, timestamp, and required-state depending on the specific attack being monitored. Volume based attacks such as DDOS [43] can be detected using Sketches[63, 80, 85] and using switch-based detection with the appropriate queries[39, 50]. They have been shown to identify attack indicators and can trigger alerts quite well for heavy hitters and other volumetric attacks. However, it can be challenging with monitoring applications that require high fidelity traffic matrices to carry out statistical analysis on network traffic [71, 73].

As an example, we motivate the need of a combination of fine-grained and coarse-grained traffic analysis by studying monitoring for the Slowloris Attack [21]. This attack keeps a very large number of connections to a target web server open, rendering it difficult for legitimate web requests to be served. We study the difference between a coarse and fine grained detector as outlined below.

Coarsed Grained Detector: This detector identifies end-points that use many TCP connections, each with low traffic volume (e.g., $\frac{\#\ Conns\ Est.}{\#\ Bytes\ Sent} > Threshold$) [50]. Also, instead of tracking this traffic volume for each and every host IP, it tracks it at a coarser-grained IP prefix level, such as the first 16 bits of the host IP address. Tracking aggregate activity consumes less memory, making it suitable to run on a P4Switch.

Fine Grain Detector: A widely used IDS, such as Zeek[27], measure HTTP request duration and identifies "stalling" flows (e.g., $duration > 10$ seconds). This is a memory and compute intensive activity, and such a fine-grained indicator is necessary to accurately detect the existence of the attack, the victims, and the attacker [25]. A host or sNIC is suitable to deploy such a detector. We list a few additional attacks in Table 1 and their flow state tracking challenges.

2.1.3 Penalize least number of flows. Since most connections are benign, a compute and memory intensive monitoring capability in the data path will unduly penalize user traffic. For this, it is desirable to minimize the number of packets that are forwarded to a sNIC-Host system for monitoring beyond the processing performed on the packet already in the switch. Between the host and sNIC, processing packets in the sNIC avoids expensive host processing, data copies and PCIe transactions that impact end-to-end latency[62].

2.2 Challenges with Standalone Systems

We desire a monitoring platform that can detect low and slow attacks that hide in the presence of large volumes of traffic. Here, we explain the challenges faced by other platforms.

2.2.1 P4Switch. Switches have been widely explored for monitoring [33, 39, 50, 84]. Here, we study hardware constraints that limits its ability to conduct fine-grained analysis.

Memory Constraints: The SRAM memory enables P4 programs to retain state across packets and to hold the exact-match tables [33]. Tracking traffic flows at fine granularity on P4Switches requires sufficient memory. As noted in [57], given the limited on-chip SRAM on a typical P4Switch ASIC (of the order of 100MB SRAM[65], even though it can forward traffic at very high rates), it may require multiple such switches, which adds to cost. To accommodate such P4Switch memory constraints, Sonata[50] carries out dynamic query refinement, so that it only focuses on the subsets of traffic that actually satisfy a query, while ignoring the rest of the traffic, just like the coarse-grained detector above.

Flexibility accessing state: Accessing all available registers on a P4Switch can be a complex task since registers in one stage cannot be accessed at a different stage [33]. Next, the number of match-action pipeline stages limits the number of sequential processing steps (10 – 20). Thus, the amount of P4Switch computation is bounded[89]. To maintain line rate, programmable switches allow only a small constant number of memory accesses per packet. This makes it infeasible to update multiple data structures for each packet[39]. Therefore, it is preferable for switches to perform coarse-grained analysis like in [39, 63] as they minimize per-packet memory operations.

2.2.2 sNIC. Programmable NICs have also been used for monitoring in [47, 76, 91]. sNICs do not have explicit stages and have more memory compared to a P4Switch, meaning they have fewer constraints. However, a pure sNIC solution can only scale to Gigabit traffic[91]. But, they outperform host-based solutions[76, 91]. Unlike SmartWatch, other sNIC solutions generally do not focus on stealthy and slow attacks. Except for Pigasus [91], but they focus on a different approach, pattern matching, for intrusion detection.

2.3 Key techniques of SmartWatch

We build a monitoring framework to detect a wide range of anomalies, caused both by volumetric attacks as well as specific slower targeted attacks which are increasingly successful [1] in the middle of a high traffic volume. For this goal, SmartWatch leverages the following key capabilities.

2.3.1 Cooperative Monitoring. The 3 components (*i.e.,* sNIC, host, P4Switch) cooperatively perform monitoring. The sNIC offloads processing, reducing load on the host, and the P4Switch directly forwards the bulk of benign flows to their destination, preventing any unnecessary performance impact.

Two-Stage detector: The first stage relies on obtaining coarse-grained flow information. Flow subsets with attack indicators at a coarser granularity are forwarded for fine-grained processing, starting with the next monitoring interval. The first detection stage is performed on the P4Switches, where the focus is on processing high volumes of traffic. We are cognizant of the impact on latency critical user traffic, so that our in-line monitoring application minimizes the number of operations per packet and is lightweight. Following the broad approach of Sonata[50], Nitro Sketch[63], and BeauCoup[39], this first stage then steers traffic subsets that satisfy aggregate-traffic queries for finer grained analysis. The next stage is implemented on the sNIC-host subsystem of SmartWatch where processor and memory intensive operations are performed on the packet. The ability to add more monitoring functions as needed, ensuring efficiency and low latency, are all important.

Control Loop: SmartWatch is responsible for configuring P4Switches, including specifying queries to be run by the switch. A more specific query would direct more targeted, and thereby less, traffic to SmartWatch from the P4Switch. We effectively create a control-loop (Fig. 1) where the sNIC-Host subsystem receives different amounts of traffic based on the degree of specificity of the query that SmartWatch installs on the P4Switch. The P4Switch queries are implemented using P4 tables where stateful operations are performed using P4 registers. At the end of a P4Switch monitoring interval, processing of the switch queries will determine some traffic subsets deemed as being suspicious (e.g., a threshold is crossed). Subsequent packets of these traffic subsets are forwarded to the sNIC-host subsystem. Over time, flows that get classified as benign by SmartWatch (e.g., after successful authentication) no longer have to be forwarded for fine-grained inspection by the sNIC-host subsystem. A P4 table in the P4Switch whitelists benign flows, thus reducing the overall traffic forwarded to the sNIC-host subsystem. This also reduces any performance impact on those flows.

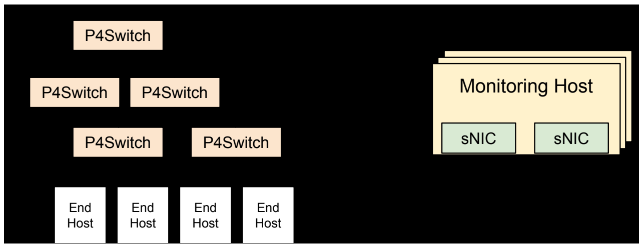


Figure 1: SmartWatch Control Loop.

2.3.2 Lossless flow monitoring. SmartWatch is a flexible network monitor for tracking flow state and logging flows with the SmartNICs (sNIC). This enables us to detect both stealthy as well as large-scale, volumetric network attacks. SmartWatch innovates by designing novel data structures in a P4-programmable Netronome sNIC. With the 40 Gbps Netronome sNIC, we are able to perform loss-less flow-state tracking and logging at 'near line rate' - a maximum 43 Mpps achieved with 64 Byte packets. This is consistent with [66] where even without any extra processing, line rate is only achievable using packets larger than 128B (This paper uses the same NIC). The bottleneck is most likely in the packet scatter-gather functionality across the micro-engines [66]. Similar limitations were also observed with the Intel XL710 NIC-based systems [8, 63]. Higher packet rates can be supported using a 100 Gbps sNIC (which we plan to do) or by sampling as in [39, 63]. However, such sampling techniques would not be able to support flow-state tracking.

Partitioning of Functions (Host vs. sNIC): The sNIC acts as an accelerator and helps track flow-state significantly faster than performing these functions on a host (e.g., TurboFlow [76] vs. Krononat [31]). On the other hand, the host has a much larger memory reservoir [57]. sNIC operations are also limited as there are no recursive functions or floating point operations available in the packet processing pipeline [14]. To ensure efficient state-tracking at line rate, we consider an sNIC cycle-budget for each and every packet. A violation of the cycle-budget potentially leads to dropping of packets at higher arrival rates. To achieve flow state-tracking at high packet arrival rates, SmartWatch designs a novel in-memory data structure on the sNIC with a flow eviction policy. This frees up a significant fraction of the time and cycle-budget on the sNIC for operations required by monitoring tasks.

sNIC FlowCache: We leverage the sNIC's memory to design a FlowCache that consist of a hash table and ring buffers. An incoming packet/flow processed by each sNIC packet processing entity updates the FlowCache using a hash of the 5-tuple. We use the sNIC memory (DRAM) to support 25 million flow entries. We propose a two-level cache on the sNIC (e.g., like a CPU's L1-L2 cache) and empirically select an eviction policy by examining the performance with a number of CAIDA traces[4]. The ring buffers accommodate evictions from the hash table and are used to periodically flush snapshots of the hash table to the host.

Reconfigurable FlowCache: Beyond providing a large reservoir of memory for tracking flow state and logging flow information, the host is also responsible for processing packets that the sNIC alone cannot process. Therefore, we need to minimize the cycles spent on the host for logging flows (e.g., handling flow records exported from sNIC to host). We develop a reconfigurable FlowCache that trades-off between sNIC packet processing throughput and the host's processing requirement. We do this adaptively by changing the eviction rate on the sNIC in response to the packet arrival rate.

To the best of our knowledge, this work is the first of its kind to cooperatively monitor traffic using a combination of programmable devices that span the range of memory and compute capabilities.

3 SMARTWATCH ARCHITECTURE

3.1 P4-based Co-operative Monitoring

In our cooperative monitoring scheme a P4Switch helps to identify attack indicators at a coarse granularity, effectively utilizing its limited memory and simple packet processing capabilities. Attacks listed in Table 2 may only be detected at a coarse granularity by aggregate traffic queries using a framework typified by SONATA [50], ConQuest [38], and Beaucoup [39].

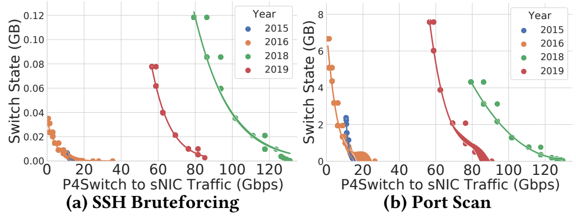


Figure 2: P4Switch State (Benign Flows)

Selective 'Bump-in-the-wire' processing: In general, packets processed in the P4Switch are directly forwarded to their intended destinations without any involvement of the P4-capable sNIC or the host. The P4 pipeline in the P4Switch passively monitors the traffic and computes the outcome of switch queries such as "is the number of ssh connections above a threshold?". If the threshold is crossed, subsequent packets of this traffic subset are steered to the sNIC (e.g., such as excessive SSH traffic destined to the same destination IP prefix). Therefore, only traffic that requires further inspection is forwarded to the sNIC-host subsystem, and are subjected to the higher latency due to this 'bump-in-the-wire'-like processing. At the sNIC-host subsystem we identify the SSH connection attempt outcomes (e.g., failure or success) and determine if there is a SSH-guess attempt. SmartWatch will then program the P4Switch to avoid benign flows (e.g., successful SSH authentication) from being subjected to the additional latency of sNIC's processing.

When installing rules in the P4Switch that whitelists benign flows, SmartWatch needs to be wary of the amount of state used in the P4Switch. To solve this problem, we borrow the 'hoverboard' intuition from Andromeda [42]. Selecting heavy flows that are benign (e.g., top-k) as opposed to mice flows helps reduce the amount of redirected traffic to SmartWatch, with relatively few rules installed in the P4Switch. Figs. 2a and 2b shows the P4Switch state vs. the traffic volume directed from P4Switch to the sNIC for the SSH brute-forcing and port scan attacks, respectively. We use CAIDA traces from different years [4] for this experiment. We see that there exists a knee, beyond which whitelisting flows does not reduce P4Switch state further. We describe these attacks in greater detail later in this paper. Furthermore, the sNIC FlowCache data structure is responsible for identifying the top-k heavy benign flows, which we describe in this section.

Switch Query Refinement: We borrow the iterative refinement approach from Sonata[50] to selectively steer flows from the P4Switch to sNIC or host. Let us consider an example where we have to track the number of SSH connections per destination IP. Treating these as key-value pairs, each destination IP is a key and the number of SSH connections is the associated value. Now, instead of tracking each individual destination IP (dIP), we aggregate them to a less-specific subset, such as based on their 16-bit prefix (dIP/16). This coarse grained analysis requires less state on the P4Switch due to fewer key-value pairs. Steering traffic that satisfies a query at the coarser dIP/16 granularity instead a dIP/32 granularity will cause much more traffic to be redirected, since dIP/32 is more specific. Iterative-Refinement zooms into traffic by filtering just the correct flows to allow multiple queries to run on the P4Switch despite its limited memory. If we compare Sonata's iterative-refinement to SmartWatch, in Sonata[50], the P4Switch memory is reused for traffic subsets at a more specific granularity (e.g., /16 instead of /8). The rest of the traffic is not examined. Instead of reusing the switch memory and only analyzing a narrow window of traffic in the P4Switch as in Sonata[50], in our design we send the narrow window of traffic to the sNIC-host subsystem, but have the switch continue to examine the coarser subset of traffic. SmartWatch reuses Sonata's[50] interface to load switch queries on the P4Switch.

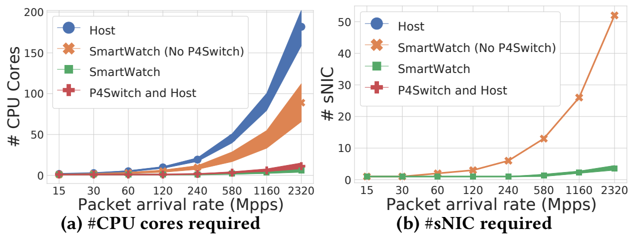


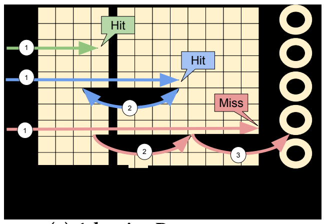
Figure 3: Scaling to Terabit traffic

Resource Usage: We simulated four different scenarios, including 1) standalone host based monitoring system, 2) SmartWatch without a P4Switch, 3) SmartWatch, and 4) host with P4Switch. The P4Switch runs the iterative query refinement algorithm derived from [50]. In this experiment, we speedup the CAIDA 2018 trace to emulate different packet arrival rates. Our findings are in Figure 3a and 3b. The y-axis is the amount of resources (i.e., CPU cores and sNIC respectively) required to sustain different packet arrival rates (x-axis). There is only one P4Switch in this simulation experiment. We observe that the P4Switch helps SmartWatch reduce the number of sNIC and CPU cores by at least 14 times by forwarding the bulk of the traffic through to the destination when the packet arrival rate is 2320 Mpps. The number of required sNIC and CPU cores are 4 and 6, respectively. This makes SmartWatch practical to scale to Terabit level traffic. The detection rate of our approach is also better, as shown in Section 5.4.

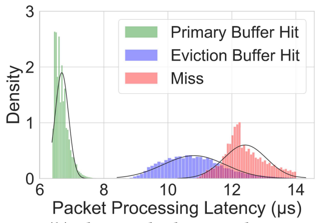
3.2 sNIC FlowCache Design

SmartWatch utilizes both the host-CPU and sNIC's packet processing engines, referred to as Micro-Engines (ME), to track flows in a loss-free manner and minimize communication overhead between host and the sNIC. The sNIC has a P4 match-action table sequence and a FlowCache data structure, which is designed using C functions. An incoming packet is first scheduled to a packet-processing micro-engine, (PME), by a 'global' load-balancer where packets are serviced in a "run-to-completion" manner (e.g., non-preemptive). The P4 match action tables provide specific packet processing/forwarding rules for the incoming flow. Packets are also processed by the FlowCache for monitoring tasks.

sNIC FlowCache: We leverage the sNIC's memory to design a FlowCache that consist of a hash table and ring buffers. The cache is in contiguous memory and is allocated at compile time. An incoming packet/flow processed by each PME updates the FlowCache using a hash of the 5-tuple. We use the sNIC memory (DRAM) to support 25 million flow entries. The ring buffers accommodate evictions from the hash table and are used to periodically flush snapshots of the hash table to the host. We dedicate 8 ring buffers each with 64K entries. Having these 8 ring buffers mitigates the access contention for ring buffers among the 80 PMEs.

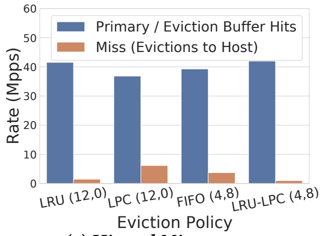


(a) Adaptive Data structure

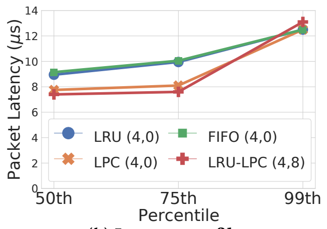


(b) FlowCache latency dist.

Figure 4: FlowCache structure and operations



(a) Hit and Miss counts



(b) Latency profile

Figure 5: FlowCache Caching Policies

Data Structure: We use a large hash table with an array of buckets to cache flow records within the sNIC. To keep the per-packet latency low, we restrict the entries for a hash index to at most 12 buckets, ensuring the sNIC can maintain high throughput and minimize packet drops. As observed in other programmable dataplanes[57], Cuckoo hashing is not suitable for caching flow records in the sNIC because it can often require multiple memory accesses. In a Cuckoo hash table, a hash collision will cause a hash entry to be moved to its secondary location, causing a write operation. On the other hand, in our proposed mechanism while there may be multiple read operations, there is just one write operation. Empirically, with a limit of 12 recursive insertions (with Cuckoo hash) vs. 12 buckets (with FlowCache), the 99.9 percentile latency for a CAIDA DC trace[4] was observed to be 2.43 times lower with FlowCache. This is because sNIC write operations are relatively expensive compared to reads. Unlike a write, for a read the calling thread yields so that another thread can continue its work while the memory is being read [23]. Note that each PME has 4 threads.

Partitioning & Eviction Policies: The key insights from previous work with typical Internet data center (DC) traffic characteristics and the CAIDA packet traces are: 1) a few large flows account for a majority of the packets, 2) numerous small flows frequently compete for a hash entry, and 3) packets of elephant flows arrive over several bursts. We experiment with a number of widely used eviction policies: Least Recently Used (LRU), Least Packet Count (LPC) and First-In-First-Out (FIFO) on the hash table with 2^{21} rows $\times$ 12 buckets per row. We further devise a split of the hash table into two buffers, namely a *Primary* $\mathbb{P}$ and an *Eviction* $\mathbb{E}$ buffer (Figure 4a). Note: we use the notation (x, y) to designate a configuration with x buckets in $\mathbb{P}$ and y buckets in $\mathbb{E}$ per row, respectively. Figure 5a shows the hit and miss rate when the sNIC is subjected to 43 Mpps (64 Byte packets) CAIDA 2018 trace [4]. Note that all four policies have the same memory footprint. Misses cause evictions of flow records to the host, and therefore we seek to minimize them. Figure 5b also shows the latency when processing CAIDA 2018 traffic. Among the four policies, LRU has the highest hits, but LPC has lower latency. In order to effectively reap the benefits of LRU (*i.e.,* handle a continuous train of packets from a flow) and at the same time benefit from the low latency with LPC (large number of hits coming a small set of elephant flows), we installed a hybrid LRU-LPC policy in $\mathbb{P}$ and $\mathbb{E}$ respectively, which provides the highest hit rate and lowest latency (median and 75%ile).

Data Operations: Figure 4a shows the structuring of the sNIC data structure and the corresponding packet update operations. Each packet processed by a PME results in one of three possible outcomes and the corresponding updates in FlowCache:

- $\mathbb{P}$ hit: The packet's 5-tuple matches one of the bucket's five tuples in $\mathbb{P}$. Then, we update the flow state.
- $\mathbb{E}$ hit: The packet's 5-tuple finds a match in $\mathbb{E}$ after scanning buckets of $\mathbb{P}$ at the same hash index. Then, we swap this entry with the LRU entry in $\mathbb{P}$ and update the flow state.
- Miss: The packet can not find a match in $\mathbb{P}$ or $\mathbb{E}$. Then, we evict the LPC entry from $\mathbb{E}$ and replace it with the LRU entry from $\mathbb{P}$ to accommodate the new flow in $\mathbb{P}$. The evicted flow from $\mathbb{E}$ will be stored in a FlowCache ring buffer.

Figure 4b shows that the packet processing latency for cache 'Hit' is lower than that of cache 'Miss'. The LRU-LPC cache replacement policy outperforms other eviction policies in terms of hit rate (Fig. 5a), and increases the PME cycles available to support additional monitoring features.

Pinning Flow Records: Low and slow attacks require per-packet flow state updates to accurately detect malicious flows. We seek to avoid host involvement to keep packet processing latency small, by dynamically pinning flow records in the sNIC FlowCache. This prevents the eviction of a flow record which potentially would result in inaccurate tracking of a suspect flow's state. In an event where all flow records of a row are pinned and one flow must be evicted, the packet being processed is sent to the host, which we strive to minimize. If a we cannot find a flow entry for a packet (Miss) either because it was evicted or not-pinned, then we create a new flow entry. We will not retrieve packet counters from the host as this tremendously increases packet latency.

Table 2: SmartWatch Resource Summ. (No P4Switch)

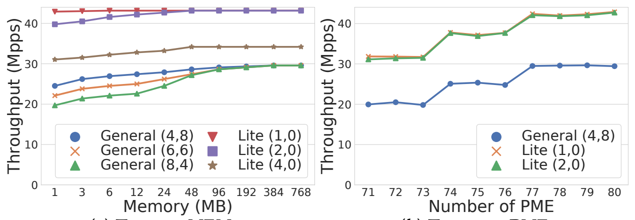
Attack	sNIC Cycles(%)	Host Processed(%)
Heavy Hitter, Heavy Changes, Cardinality, Flow Size Estimation and Slowloris	80.32	0
Zeek SSH Bruteforcing	1.79	1.24
Zeek Expiring SSL certificate	1.98	1.35
Zeek FTP Bruteforcing	1.85	1.19
Zeek Kerberos Ticket Monitoring	1.99	2.9
In-Sequence Forged TCP RST	1.94	0.95
TCP Incomplete Flows	2.01	8.3
Stealthy Port Scan	1.99	0
DNS Amplification	1.93	0
Micro-bursts	2.08	0
EarlyBird Detection Worms	2.06	0

Reduce Host Packet Processing with FlowCache We benchmark 15 attack detectors *simultaneously* running in SmartWatch. Table 2 (Host Processed col.) shows the percentage of trace packets (CAIDA 2018) processed by the host. Most are processed by the sNIC. Thus, there is a dramatic reduction in host overhead. Depending on the flow-state, select packets are forwarded to the host. The average packet processing latency reduces to just 28% compared to when everything runs on the host. PCIe transactions and packet copies contribute to the host-based processing being slower [62].

3.3 sNIC Reconfigurable FlowCache

Increasing the number of buckets accommodates more flows with fewer evictions from the sNIC to the host. But, it lowers throughput due to higher processing latency. We designed a reconfigurable FlowCache with two operational modes on the sNIC. Fig. 7a illustrates the transition from *General Mode* (4,8) to *Lite Mode* (2,0) while reordering bucket entries. To adapt to fluctuating packet arrival rates, we dynamically mutate between the General and Lite mode, with minimal overhead. General Mode has 12 buckets per row in (4,8) configuration. This captures most of the large flows and operates in a loss-free mode for arrival rates upto 30Mpps. But, it is insufficient to keep up with the maximum achievable line-rate (43Mpps) for the 40Gbps NIC. But, it also has the benefit of a lower eviction rate (*i.e.,* reduced transfers to host). The Lite Mode on the other hand supports higher packet rates. Fig. 6a, shows that both (2,0) and (1,0) meet near line-rate for 64B packets. So, we select (2,0) as Lite mode. Unfortunately, the Lite mode results in a higher eviction rate because of a lack of buckets to resolve collisions, despite having the same memory footprint. Figure 7b shows that the CPU time [19] for the host thread responsible for snapshotting flow records increases by 2.08 times when we employ the Lite (2,0) mode compared to the General mode of the sNIC FlowCache. This is because of the 47% increase in eviction rate.

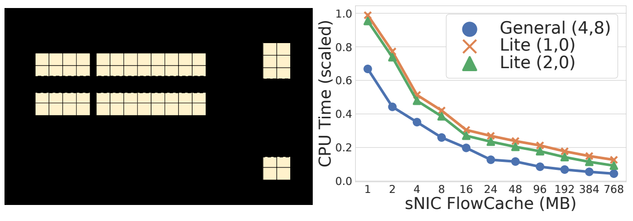
Map Offline Tasks to Micro-engines. There are a total of 80 sNIC MEs usable for i) packet processing (PMEs) or ii) custom processing (CMEs), separated from the packet processing pipeline. Fig. 6b shows the variation of throughput as we change the number of PMEs. We are able to allocate a total of 3 MEs for custom processing (as CMEs) without any degradation in the maximum packet processing throughput. We use CMEs for the task of switching between the two modes. Later, we also show their utility for other monitoring tasks such as reporting flows that cause micro-bursts.



(a) Tput vs. MEM (b) Tput vs.# PME

Figure 6: FlowCache Throughput vs. sNIC resources

Correct State-Tracking without Flow Duplicates Alg. 1 shows the candidate buckets for the two modes. For the *General Mode*, packets have to be checked against bucket [0, 12) across $\mathbb{P}$ (e.g. [0,4)) and $\mathbb{E}$ (e.g. [4,12)). For the *Lite Mode*, only two buckets are checked at an offset determined by the higher order bits of the hash digest. Clearly, the candidate buckets for the *Lite Mode* are a subset of the *General Mode*. Thus, there is no overhead to switch from the *Lite Mode* to *General Mode*, and there is less flow-matching penalty in the *Lite Mode*. On the contrary, flow entries will have to be reordered when transitioning from *General* to *Lite Mode*. The contiguous memory and the logical partitioning between $\mathbb{P}$ and $\mathbb{E}$ allows us to move the logical boundary between them and resize the number of rows in the hash table. We use a global variable `mode` for the current operation mode. A CME periodically tracks the packet arrival rate (EWMA with $\alpha = 0.75$ over a window of 100 samples) and compares with a threshold to switch modes (see Alg. 4 in Appendix 9.4).



(a) General to lite mode (b) Host snapshotting overhead

Figure 7: FlowCache overheads

General to Lite Transition When the packet arrival rate exceeds the rate supported by the General mode, the CME triggers a *switch over* to Lite mode. The CME marks all the hash table rows as 'dirty'. The first PME that finds a bucket within a 'dirty' row performs the cleanup, which involves reordering the flow entries and then marking the row as 'clean'. The PME gains exclusive access to the row (i.e., allow no concurrent flow updates) and reorders the flow records honoring the Lite mode's logical boundary. We do this lazily, as packets arrive, as it is slower for a single CME to reorder all flow records, taking at most 14μs (see Alg. 3, Appendix 9.3). The 80 PME on the sNIC process packets in parallel for other FlowCache rows even if one PME has exclusive access to a specific row. However, some packets end up waiting for the clean up process to relinquish the exclusive access to a row. We observed this wait time to be less than 5μs, before it can make its own state updates.

Lite to General Transition When the packet arrival rate drops below a threshold, the CME initiates the transition to General mode.

Flow entries do not have to be reordered and the eviction process can run as usual. For example, a General mode's row consists of 12 buckets, which is sub-divided to support six rows of two buckets each in the Lite mode. Alg. 1 Line 4 show that both the Lite and General modes share the same hash index. Therefore, the packet is forwarded to the same '12 buckets' in physical memory. Since all 12 buckets will be probed in the General Mode and since the candidate buckets in the Lite mode are a subset of the General mode (Alg. 1 Line 6 and 9), correctness is ensured.

Algorithm 1 Candidate Bucket Selection

```
1: Initialize:
       $B \leftarrow 12$                          ▷ General Mode buckets
       $b \leftarrow 2$                           ▷ Lite Mode buckets
       $x \leftarrow 21$                          ▷ Number of bits (Hash Digest)
       $row \leftarrow 2^x$                       ▷ Hash Table Rows
2:
3: procedure BUCKET_SELECTOR($hash\_digest$)
4:     $hash\_index \leftarrow hash\_digest \ \& \ (rows - 1)$
5:     if General Mode then
6:         $candidate\_buckets \leftarrow [0, B)$
7:     else if Lite Mode then
8:         $offset \leftarrow ((hash\_digest >> x) \ mod \ \lceil \frac{B}{b} \rceil) \times b$
9:         $candidate\_buckets \leftarrow [offset, \ offset + b)$
```

3.4 Sub-components for Host Support

The host can provide a large amount of memory (> 200GB) and storage (~2TB) compared to the switch and sNIC. We leverage the host as a global pool to collect and store all flow-related information over multiple snapshots for detailed forensics. We leverage the sNIC to aggregate flow records and export them to the host periodically (*e.g.,* every 5 seconds). Since the sNIC may export a particular flow's entry several times due to eviction or being aged out, the host is responsible to correctly aggregate each flow's information. We DMA flow records by borrowing the implementation from [69]. The host caches these flows with a $2^{30} \times 1$ hash table. The entries from host cache are periodically (per measurement interval) flushed to a Redis [20] datastore for flow-logging. The host CPU also has the capability to process packets through SR-IOV ports (DPDK [49]). Some IDS/IPS components cannot be offloaded to the sNIC as they require complex operations (See §2.3.2). We dedicate distinct SR-IOV ports for each supported function on the host. Host NFs include: 1) Zeek[27] for IDS/IPS scripts, 2) Timing Wheel[81] to buffer and release packets, and 3) NFs using the host's larger memory pool.

4 IMPLEMENTATION

Low Cost to Add Monitoring Features. The required state for baseline flow logging is $768MB$ to cache 25 million flow records, including timestamps and packet counters. Given the sNIC has 8 GB memory and supports bulk operations, additional attributes can be added to the flow entry to track flow-state, enabling support of additional monitoring features. However, the packet processing overhead must be minimized to perform lossless tracking. The baseline FlowCache, without any additional monitoring features, which supports flow logging consumes most of the sNIC cycles (80.32% of the total cycles). Flow logs exported to the host (always enabled) can be analyzed offline for heavy hitter detection, heavy changes, cardinality estimation, flow size estimation, and Slowloris. Our eviction policy ensures FlowCache processes packets at near-line-rate. Table 2 shows that the cycles consumed by other monitoring features is very small compared to FlowCache, and therefore do not reduce packet processing throughput. This is because of the parallel processing across a large number of PMEs on the sNIC, including hardware support of atomic operations. More on the FlowCache lockless flow record update scheme can be found in Appendix §9.1.

Symmetric Hash Function: Detectors require session based flow-state tracking. We need to ensure IP packets in the reverse direction map to the same bucket as that of the forward direction. Hence, we use a symmetric hash function[83].

Table 3: sNIC Comparison

Attribute / sNIC	**Bluefield MBF1L516A -ESNAT**	**LiquidIO OCTEON TX2 DPU**	**Netronome Agilio LX**
Processor	2.5 GHz [17],[2]	2.2 GHz [10]	1.2 GHz [13]
Parallelism	16 cores [17]	36 cores [10]	96 cores [13]
L1 Size	32 KB [17]	32 KB [62]	64 KB [9]
L1 Access Time	5.0 ns [62]	8.3 ns [62]	13 ns [23]
L2 Size	1 MB [17]	24 MB [10]	256 KB [9]
L2 Access Time	25.6 ns [62]	55.8 ns [62]	51 ns [23]
DRAM Size	16 GB [62]	16 GB [62]	8 GB [15]
DRAM Access Time	132.0 ns [62]	115.0 ns [62]	137 ns [23]
Atomic Primitives	Yes [62]	Yes [62]	Yes [23]
Programmability	GNU [17]	GCC [10]	Micro C/P4 [9]

4.1 Generality of sNIC Implementation

In this section, we study the generality of our implementation on the Netronome sNIC, the potential for adoption with other, such as BlueField and LiquidIO sNICs. The details of the 3 sNICs are listed in Table 3. Using this, we ran a discrete event simulation with a CAIDA trace[4] containing almost 2 billion packets, where all packets are reduced to 64B to create a worst case stress test. All 3 sNICs support programmability and atomic primitives with a cache structure and multiple packet processing cores. We model the number of cycles consumed in the FlowCache for the BlueField and LiquidIO sNICs by performing the measurements on our Netronome sNIC (e.g., no. cycles for hits / misses). We then estimate the packet processing latency and the number of packets processed per second across all compute units based on the different processor speeds and memory access latencies (Table 3 specification). Using our trace-driven simulation, we derive the packet throughput for BlueField and LiquidIO sNICs to be 40.7 and 42.2 Mpps respectively, compared to the throughput for Netronome sNICs, which was 43 Mpps. The reason for their slightly lower throughput compared to Netronome is because of the fewer number of processing cores. SmartWatch is a monitoring platform. But, thanks to its P4 and SR-IOV capability, it can support common data plane functionalities such as switching, tunneling, and QoS, which are typically supported by OVS [42] in today's DCs. All three sNICs support OVS offload and SR-IOV [10, 11, 13, 17]. SmartWatch and OVS can act as the monitoring and connection tracking modules of the pipeline, respectively.

5 EVALUATION

Testbed: We evaluate the effectiveness of SmartWatch on our local testbed consisting of Linux servers (kernel ver. 4.4.0-142), each with 10 Intel Xeon 2.20GHz CPU cores, 256GB memory and Netronome Agilio LX 2×40 GbE sNICs with 8GB DDR3 memory and 96 highly threaded flow processing cores. We use three packet generators, each running Moongen[45] to replay PCAP traces at the high rate

(43 Mpps using 64B packets) for our stress tests. Attack and background traces are timestamp shifted and then merged using editcap [5] and mergecap [12], respectively. To truncate packets to 64B, we use tcprewrite[22].

First, we compare SmartWatch against monitoring systems deployed on the host, such as Zeek. Second, we show how SmartWatch can reduce the memory (SRAM) pressure on a P4Switch for cooperative monitoring for detecting covert timing channels [84] and website fingerprinting [33]. Third, we show how FlowCache helps improve long (5 sec) and short (<200μs) timescale traffic analysis. Lastly, we compare cooperative monitoring technique to Sonata[50] in terms of accuracy. In our experiments, all detectors that are based on flow logging are processed offline on the host, while other attacks, such as SSH Brute Forcing, are detected online.

Evaluation Traces: We use four different traces: 1) CAIDA Traces [4] (years 2015 to 2019) containing 1 to 1.9 billion packets., 2) For stress testing, we created traces with 64B packets with CAIDA traces, 3) For targeted attacks, we used official test traces from Zeek IDS [27], and 4) Univ. of Wisconsin DC measurement traces [35].

5.1 Stateful Attack Detection: SmartWatch vs. Host-based Detection

We study three monitoring usecases and detail how SmartWatch outperforms traditional Zeek[27] or NFs deployed on the host.

5.1.1 SSH brute-forcing.

Attack: Many nodes with distinct source IPs use different username/password combinations on SSH login servers [53]. For this attack we leverage the host running Zeek[27] and partition processing tasks between the sNIC and host. The host is only involved in the authentication phase, and if successful packets avoids PCIe transactions to the host.

Detection: Track the number of failed SSH login attempts, ψ, by a remote node in a given time interval. Raise an alert when ψ exceeds a threshold[28].

P4Switch Role: For the SmartWatch framework, the P4Switch measures if the number of SSH connections attempts have exceeded a threshold. Flow subsets with significant number of authentication attempts observed in P4Switch are forwarded to the sNIC. The P4Switch cannot conduct fine-grained analysis by itself as it cannot determine the number of failed SSH connection attempts ψ for a specific remote node, since they are encrypted. It is not possible to heuristically determine connection attempt outcomes in the P4Switch as it requires per-packet state transitions (see §2.2.1).

SmartWatch Role: As a new SSH connection arrives, we pin the flow entry within the sNIC until the outcome of the connection attempt is determined. FlowCache ensures that subsequent packets of pinned entries are forwarded to a host NF running Zeek. Zeek heuristically guesses the login attempt outcome by tracking connection state transitions and the amount of data communicated [6]. If the host NF detects authentication success, FlowCache unpins the flow entry (evicted as needed) and does not send subsequent packets to the host NF by updating the match action table. If ψ exceeds a threshold, the P4 table blacklists the source IP. Here, the gain is experienced by all packets beyond the SSH authentication phase. The sNIC tracks approved vs. non-approved flows, causing only 1.24% packets in the SSH trace to go to the host (Table 2). The remaining packets of the trace avoid costly transfers to the host.

Evaluation: Figure 8a shows the SSH packet latency for three scenarios: 1) successful authentication with SmartWatch 2) successful authentication with baseline Zeek, and 3) multiple authentication failure attempts with SmartWatch. Here, 3 failure attempts in 30 minutes generates an *SSH_GUESS _ATTEMPT* event. We utilize traces available in the Zeek package for this experiment. Once a SSH connection is approved by Zeek (*SSH_AUTH _SUCCESS*), packets are no longer processed in the Zeek NF, reducing the average packet processing latency by 77% compared to baseline Zeek. In Fig. 8a, SYN packets result in unavoidable latency spikes up to ~2250μs because of the need to remove connection state.

Similar Attacks: Expiring SSL certificates[29], FTP brute-forcing[24], and Kerberos ticket traffic[26]. Similar to SSH connections, Zeek can scrutinize the validity of the connection in the host. Following Zeek's approval, their packets are entirely processed in the sNIC.

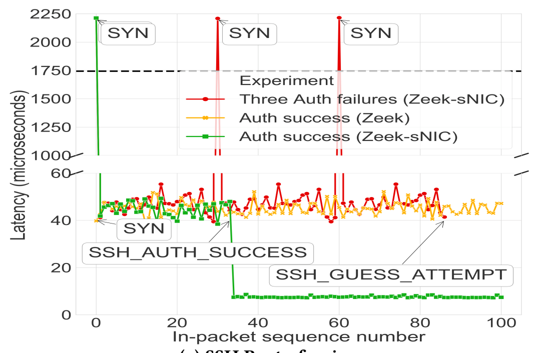
5.1.2 TCP Forged Resets.

Attack: An adversary disrupts TCP connections by sending a forged TCP RST packet to either end of a connection. We leverage the host running a timing wheel where potentially forged RST packets can be buffered until the RST packet is classified as malicious or benign. In this example, the sNIC is responsible for steering the smallest possible traffic subset to the host, thus reducing the high latency PCIe transactions.

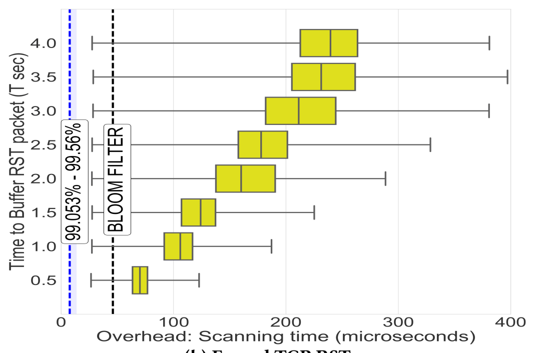
Detection: This can be detected using 1) RST packets with outdated SEQ numbers; 2) Multiple RST packets with increasing SEQ number; 3) race conditions between the RST packet and in-flight end-host data packets. Here, we focus on the race condition between the RST packet and data packet as it requires the most flow-state tracking and is difficult for attackers to avoid. Race conditions are unlikely if the RST packet has been generated by an end-host. Hence, it is recommended that a monitor maintain state for a time interval T=2 seconds from the arrival of the RST packets to determine whether the RST is genuine or forged [82]. On the host, we implement a timing wheel [81] to buffer RST packets.

P4Switch Role: In the SmartWatch framework, the P4Switch measures if the number of RST packets exceeded a threshold. Flow subsets with a large number of RST packets seen in the P4Switch are forwarded to the sNIC. The P4Switch by itself cannot conduct this fine-grained analysis (see §2.2.1). The precise victim in the flow subset requires tracking the arrival of RST packets and inspecting the sequence of subsequent packets [82].

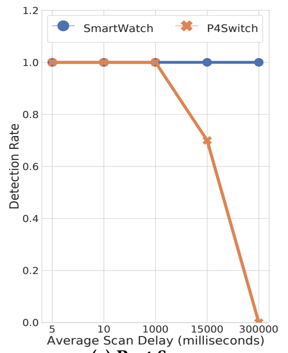
SmartWatch Role: FlowCache steers TCP RST packets to the host via a dedicated SR-IOV port and pins the flow entry in sNIC FlowCache. The RST packet is released by the timing wheel after $T = 2$ seconds if no race conditions are identified (i.e., not forged). On arrival of a genuine data packet right after a forged RST packet, sNIC FlowCache notifies the timing wheel (i.e., state-tracking) after which the forged RST packet is discarded from the timing wheel, without reaching the destination. sNIC FlowCache records are unpinned when the buffered RST packet is released to its destination, or when a forged RST is actually detected. The gain is experienced by all packets arriving prior to the monitor seeing a RST packet in a connection. In our experiments, only 0.95% packets of the CAIDA



(a) SSH Brute-forcing



(b) Forged TCP RST



(c) Port Scan

Figure 8: SmartWatch Resource Requirement and Latency

data center trace go to the host and experience the additional processing delay of the monitoring NF and PCIe transaction (Table 2).

Evaluation: Only unique RST packets should be inserted into the timing wheel while duplicate RST or data-after-RST must be immediately notified to operator (i.e., an attack). Ensuring uniqueness requires the timing wheel to be scanned while buffering the RST packet, potentially degrading packet processing latency. This processing can be bypassed for some packets using a Bloom Filter, accelerating the RST buffering operation. Fig. 8b shows the packet processing latency and percentage of packets experiencing it for different values of T for the 2018 CAIDA trace [4]. As T increases, so does the scanning time, as more buffered RST packets have to be checked. The blue vertical line is the mean round-trip latency for packets processed solely in sNIC FlowCache. This accounts for 99.053% packets of the trace. As for RST packets, uniqueness identified using the Bloom Filter incurs an avg. 411 ns extra processing time and accounts for 69.7% of RST packets. Remaining RST packets incur extra latency due to the scan operation on the timing wheel, which is necessary to identify the previous (unexpired) RST packet.

Similar Attacks: Similarly, we can detect TCP Incomplete Flows. Instead of looking for race conditions, we check if a SYN packet wasn't followed by DATA packet for some time [72]. SYN packets aren't blocked, as in forged TCP RST.

5.1.3 *Port Scan Attacks.*

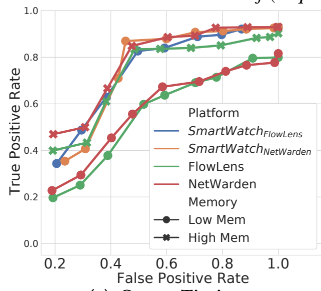
Attack: Port scan is a common method for discovering exploitable channels (i.e., open ports) on network servers[55]. SmartWatch partitions the monitoring between the host and the sNIC. The sNIC inspects and reports to the host the outcome of TCP three-way handshake (i.e., incomplete vs. established). The host tracks this over longer time scales, to classify if the remote node is a scanner or benign. But, no packets are forwarded to the host.

Detection: [55] describes a detection scheme using the number of failed connection attempts (i.e., failed three-way handshake) as an indicator to identify scanners. It determines the outcome of the ith connection attempt from remote server r as an indicator variable ϕ_i^r. [55] then runs a hypothesis test determining whether the remote node r is an attacker or not.

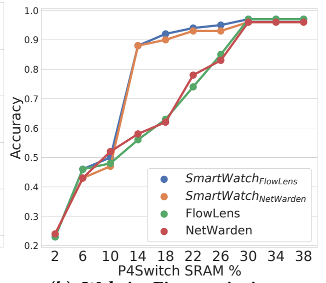
P4Switch Role: For the SmartWatch framework, the P4Switch measures the number of connection attempts. Flow subsets with significant number of connection attempts observed in the P4Switch will be forwarded to the sNIC. The P4Switch itself cannot conduct fine grained analysis as it cannot track the connection outcomes which requires flow-state tracking over long time scales, requiring significant amounts of P4Switch memory (see §2.2.1).

SmartWatch Role: FlowCache computes ϕ_i^r by tracking flow state on a per-packet basis. It waits a short period of time to see the responses for the SYN packet: a SYN ACK (successful); a RST (incorrect service); or no response (incorrect destination/port) from the destination. The flow record is pinned until ϕ_i^r is determined (e.g., 1 if it completes the three-way establishment handshake, 0 otherwise). The indicator variable ϕ_i^r for the flow is stored in the flow record and gets exported to the host. The host then classifies the remote node r as an attacker/benign using hypothesis testing[55].

Evaluation: We use NMAP [16] to generate scanning traffic with different scanning intervals. We merged this attack traffic with the Univ. of Wisconsin datacenter measurement trace [35]. Thus, the attack traffic is hiding in a much larger data stream. Larger scanning intervals become more difficult to detect (i.e., paranoid scanner). Figure 8c shows the detection rate in relation to different scanning delays comparing SmartWatch and (standalone) P4Switch. As SmartWatch carries out memory intensive operations, it can track protocol state transitions, allowing for it to compute the indicator variable ϕ_i^r and detect scans with long scanning intervals.
Similar Attacks: Detecting DNS amplification by computing the amplification factor $\frac{sizeof(response)}{sizeof(request)}$ instead of ϕ_i^r [56].



(a) Covert Timing



(b) Website Fingerprinting

Figure 9: Covert Channels

5.2 Reducing P4Switch memory pressure

We study two monitoring usecases in this section and detail how SmartWatch assists a P4Switch (NetWarden[84] or FlowLens[33]) consume less SRAM and control plane cores, aiding common forwarding operations in the DC.

5.2.1 *Covert Timing Channel.*

Evasion Covert timing channels can exfiltrate secret data by modulating the inter-packet delays (IPDs) of network traffic, e.g., by using large (small) IPDs to encode ones (zeros)[84]. SmartWatch helps achieve the same True Positive and False Positive Rate (TPR and FPR) with 8 times less P4Switch SRAM occupancy to collect the IPD distribution. Further, no control-plane or co-located server resources are used.

Detection: Since the modulated traffic trace would have different IPD distributions (bimodal) from those of usual traffic (normal), timing channel detectors look for statistical deviations (KS-Test[30]) between a given IPD distribution and a known-good distribution as obtained from training data[84]. We use a CAIDA[4] workload, where 90% flows are benign and the other 10% are modulated by the attacker to leak data. The modulation ranges from 1µs to 100µs.

P4Switch Role: Since iterative-refinement does not support IPD collection, we compare against two implementations of the P4Switch. FlowLens maintains a flow lookup table, assigning a flow offset to each flow ID. The flow offset locates the flow's set of bins to store the IPD distribution. NetWarden is similar, but instead of using k bins for each connection, it uses k CountMin Sketches to collect the IPD for all connections. The NetWarden dataplane consists of pre-checks that executes range checks on the IPD distribution. On the other hand, FlowLen's control plane reads the batch of collected data from the switch when a timer expires. We have extended these two P4Switch data structures ($SmartWatch_{FlowLens}$, $SmartWatch_{NetWarden}$) to forward packets to SmartWatch's sNIC subsytem when a pre-check[84] range query is satisfied.

SmartWatch Role: On the sNIC, we program the flow IDs that were determined suspicious on the switch (e.g., pre-check). For all the programmed flows, we maintain fine-grained bins (e.g., bin size = $1\mu s$), intended to detect modulation between 1-100μs. Since the number of flows directed to the sNIC is small, this is feasible. On the sNIC's CME, when a timer expires we carry out the complete statistical test (KS-Test) within the sNIC and classify the channel as benign or if it is being modulated by an attacker to leak information. Flows programmed on the sNIC are pinned on SmartWatch's FlowCache to prevent evictions. The benefit of this sNIC-based co-design is less SRAM resource consumption on the P4Switch along with the complete elimination of the need to use CPU cores in the switch's control-plane (or co-located host) to run the statistical test.

Evaluation: P4Switches have a limited amount of SRAM (order of 100MB [65]) that is required for tables and registers [33]. FlowLens and Sonata occupy less than 40% and 20% (e.g., 8 of 32 Mb per stage) SRAM, respectively[33, 50]. As SmartWatch concurrently leverages both their data structures for the P4Switch deployment, it would leave only 40% SRAM total to support common forwarding behaviors, like access control, rate limiting or encapsulation. We show SmartWatch can have the P4Switch operate with substantially more, 75% SRAM available for general operations (instead of only 40%) while achieving similar True and False Positive Rate (TPR/FPR). We consider a high and low memory implementation of NetWarden and FlowLens. For high memory FlowLens implementation, we set the quantization level (QL - influencing bin size and number of bins) to 0, causing each flow to take up 3000 bytes. In contrast, we set the QL to 3 for the low memory implementation (376 bytes per flow). For NetWarden, the low-memory implementation uses a CountMin Sketch with 8 times less memory (0.5 MB as opposed to 4 MB) by altering the Sketch's dimensions. The sNIC fine-grained bins alongside the CME running the KS-Test ensures the complete statistics calculation is carried out for packets forwarded to the sNIC-host subsystem, attaining similar TPR and FPR, despite substantially lower SRAM occupancy (Fig. 9a). In SmartWatch, when a timing covert channel is detected, we simply copy over the packet contents to the sNIC memory and create a new packet after a random delay. However, given the limited sNIC memory, when the sNIC 's buffers exhaust, we then do this on a host NF.

5.2.2 *Website Fingerprinting.*

Evasion: Allows users to hide the destination address behind a proxy and the content of website visits from external observers using encryption [33]. SmartWatch helps achieve the same accuracy with 14% P4Switch SRAM occupancy compared to 30% needed for FlowLens and NetWarden.

Detection: Identify which sites are access by collecting the flows' packet length distributions (PLD) and feeding them to a Naive Bayes classifier[33]. We use widely used traces containing web page accesses over OpenSSH [70, 92].

P4Switch/SmartWatch Role: The P4Switch and SmartWatch play the same role as with the covert timing channel detection case, except we now collect PLD instead of IPD, and the sNIC CME runs a Naive Bayes classifier instead of KS-Test.

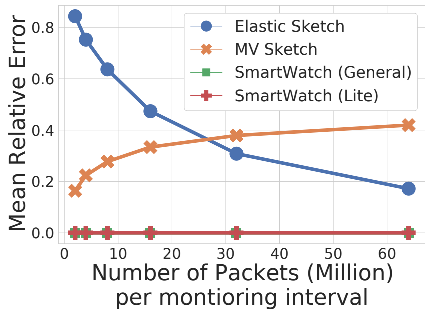
Evaluation: The CME classifies destination IPs as a "hidden destination address" or not. We calculate the accuracy using a Multinomial Naive-Bayes classifier, which leverages the PLD of the incoming and outgoing data of a connection as features [33]. Fig. 9b shows the website fingerprinting accuracy with respect to the P4Switch SRAM occupancy. With SmartWatch, we can bring down this occupancy to 14% from 30% and still achieve good accuracy (> 90%). SmartWatch sees a steep drop in accuracy around 10% SRAM occupancy because the range checks cannot identify what traffic needs to be sent to the sNIC-host subsystem (Fig. 9b).

5.3 Traffic Analysis

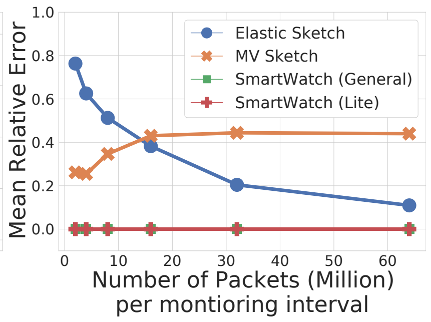
In this section, we compare SmartWatch's FlowCache to common Sketch designs for traffic measurement over long-timescales (5 sec) and examine its ability to perform fine-grained traffic measurement ($< 200\mu s$) compared to a P4Switch.

5.3.1 Volumetric Analysis. For packet rates below the capacity of SmartWatch, we support completely lossless flow-logging (distinct from flow state tracking). When simultaneously running all monitoring functions shown in Table 2, the packet throughput is not impacted by additional features since they consume minute fractions of the cycles compared to FlowCache (Table 2). The baseline FlowCache has flow-logging always active, so it can be used for heavy hitter, heavy change detection, etc. As each PME operates at 1.2 GHz[13], the small number of cycles used for additional monitoring features has only a small impact on packet processing latency.

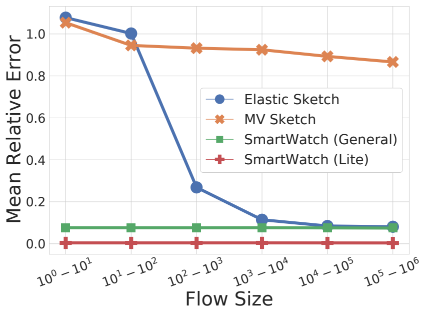
Throughput: Fig. 11b compares SmartWatch to Sketch based mechanisms. We implemented Elastic Sketch [85], Nitro Sketch [63], and MVSketch [80]. We compare SmartWatch running FlowCache in both the General and Lite modes: In this experiment, all the



(a) Heavy Hitter Detection



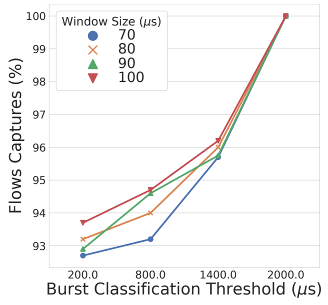
(b) Heavy Change Detection



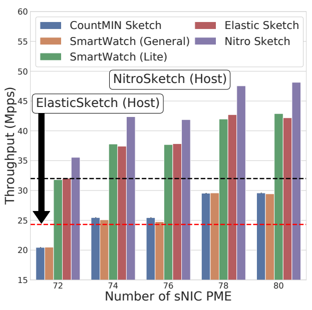
(c) Flow Size Distribution

Figure 10: Volumetric Analysis (Accuracy)

monitoring functions listed in Table 2 are simultaneously running in SmartWatch. We also compare SmartWatch with a host based processing alternative (using an Intel NIC) to show the benefit of partitioning the functionality with a sNIC. We use Fig. 11b to guide us in only using 3 of the total 80 MEs for background processing (e.g., tracking the packet rate for switching between Lite and General mode). The remaining are PMEs (x-axis). The General Mode supports loss-less monitoring for packet rates below 30 Mpps. For higher rates, up to the max. of 43 Mpps, SmartWatch performs loss-less monitoring using the Lite mode. The control-loop (i.e., specific query adaptation with P4Switch) ensures that the traffic sent to SmartWatch is limited to what it can process. The only platform that yields higher throughput than SmartWatch is Nitro Sketch [63], but that is because it performs packet sampling to reduce the average memory operations per packet. CountMIN Sketch throughput is low due to multiple hash calculations per packet [63].



(a) Microburst Error



(b) Throughput

Figure 11: SmartWatch Traffic Analysis

Accuracy for Volumetric Analysis: Figure 10 shows the accuracy of our platform compared with the invertible Sketch based solutions, Elastic Sketch [85] and MV Sketch [80]. For all experiments we use the CAIDA traces[4] from years 2015 to 2019, and reduce the packet size to 64 Bytes, to be replayed at 43 Mpps. First, we conduct heavy hitter detection. We use a predefined threshold for a heavy hitter (0.001% of total packets received in the monitoring interval) and vary the monitoring interval from 2 to 64 million packets. Second, we conduct heavy change detection. The predefined threshold for heavy change is 0.05% of the total changes across two consecutive intervals. Third, we compared the platforms based on the collected flow size distribution. For heavy hitter and heavy change detection, as the monitoring interval increases so does the error in Sketch based methods due to more hash collisions. SmartWatch's lossless monitoring approach instead evicts flow records to the host, preventing any accuracy drop. For heavy hitter and heavy change detection, both modes of SmartWatch have overlapping lines, with zero mean relative error. For flow size distributions, Elastic and MV Sketch prioritize the retention of heavy flows, causing the small flows to be inaccurate. SmartWatch, on the other hand, tracks all flows in a lossless manner and has lower error rate. However, for flow size distributions, the Lite mode has a higher accuracy compared to General mode as the latter does not sustain the high packet arrival rate in this experiment.

5.3.2 *Micro-bursts.*

Anomaly: Micro-bursts are congestion events that (typically) last $< 200\mu s$, 40% of inter-burst gaps are $< 100\mu s$ [90]. In this task the sNIC is responsible for reporting the culprit flows in a microburst without any approximation.

Detection: ConQuest [38] and BurstRadar [54] propose detecting micro-bursts when queuing delays go above an operator-specified threshold and then report the responsible flows.

P4Switch Role: In SmartWatch, the P4Switch identifies the link experiencing the microbursts and forwards the flow subsets that suffered the microburst event to the sNIC. Fine-grained analysis at sNIC accurately identifies the source of micro-burst (unlike the overestimation in [38]).

SmartWatch Role: FlowCache works with a linear array, $\mathbb{L}$, of size 96MB storing the unique IP 5-tuple entries, to accurately report details of flows causing microbursts to the host, along with the packet count. We use a doubly-linked entry (i.e., reference from FlowCache entry to $\mathbb{L}$ entry and vice versa) to ensure connection uniqueness in $\mathbb{L}$ and to quickly locate flow entries in FlowCache from $\mathbb{L}$. SmartWatch monitors the queuing delay on the sNIC as a trigger to activate micro-burst analysis. The PMEs calculate the difference between the current timestamp and the MAC ingress timestamp to compute per-packet queuing delay. When this delay exceeds the threshold, PMEs flag it, and generate an identifier for this micro-burst event. Subsequent packets update FlowCache and then $\mathbb{L}$. Once the micro-burst ends, the CME is responsible for scanning $\mathbb{L}$. The small size of $\mathbb{L}$ allows for rapid scanning for computing metrics of interest (within 200ms). A micro-burst ends when the queuing delay drops below a threshold. Following this, the contributing flows are identified by scanning the log, and all FlowCache records are allowed to be evicted to the host.

Evaluation: We use the Wisconsin trace [35], replayed at 10x the original rate, to detect and identify contributing flows of the micro-burst patterns as in [37]. We test each burst event by quantifying the flows present in the ground truth vs. the fraction in $\mathbb{L}$, reflecting how SmartWatch identifies (and reacts to) bursts. Then, there is no error introduced in SmartWatch when reporting the flow responsible for the queue build-up. But, there may be false micro-bursts identified due to a conservative setting of the threshold (*i.e.,* of queuing delay). The number of bursts estimated in SmartWatch vs. ground truth was higher by 1.32% to 8.23%, for queuing delay thresholds ranging from 2000μs to 200μs. Fig. 11a shows that as we reduce the queuing threshold to classify a micro-burst, we miss a fraction of the flows that were a part of the burst in the ground truth. A burst classification threshold of 200μs, captured 92.7% of the flows in the ground truth. But, a burst classification above 1700μs identifies all 100% of flows.

Similar Attacks: Worm detection where we lookup the hash of the combined payload and destination IP and check whether the worm signature match (i.e., stored in $\mathbb{L}$)[75].

5.4 Effectiveness of Co-op Monitoring

5.4.1 Detection Rate. SmartWatch and Sonata allow processing traffic across multiple links incident on the switch, achieving an aggregate terabit scale monitoring. The standalone host will have the highest detection rate because of the highest degree of flexibility and most memory. However, it is the least scalable option, as shown in Figure 3a. Of all the stealthy attacks detected by the host, we show the fraction of such attacks that are detected by SmartWatch and Sonata in Table 4. The drop in detection rate for Sonata is because of the lack of fine-grained processing. In SmartWatch, the higher detection rate is due to fine-grained processing for flow-subsets. The slight reduction in detection rate for SmartWatch relative to standalone host is due to the attacks expiring within the P4Switch before those packets are forwarded to the sNIC.

Table 4: Detection rate relative to host

Attack	Sonata	SmartWatch
Slowloris	0.44	0.94
Zeek SSH Bruteforcing	0.24	0.79
Zeek Expiring SSL certificate	0.68	0.68
Zeek FTP Bruteforcing	0.25	0.81
Zeek Kerberos Ticket Monitoring	0.73	0.78
In-Sequence Forged TCP RST	0.11	0.80
TCP Incomplete Flows	0.84	0.93
Stealthy Port Scan	0.4	0.90
DNS Amplification	0.38	0.88
EarlyBird Detection Worms	0.59	0.70

6 RELATED WORK

Switch Queries: Recent work that support a wide range a queries such as Sonata [50], NetQRE [88], OpenSOC [18] Gigascope [41], Omnimon [52], PINT [34], NetWarden [84], FlowLens [33] and BeauCoup [39] exploit programmable switches. However, these systems do not conduct protocol-level inspection and thus cannot detect a wide range of stealthy attacks. Marple [68] performs queries on a programmable switch at line-rate, using a large backing store for evictions from the switch data structure (implemented as a hash table). Omnimon[52] conducts network-wide measurement at full accuracy, but is also limited in detecting low-rate attacks because of P4Switch limitations. SmartWatch can complement Omnimon for a more comprehensive network-centric monitoring solution. Turboflow [76] uses a sNIC with a hash table to store every packet by having microflow records (mFRs). But their very high eviction rate puts a substantial load on the host. SmartWatch instead partitions the aggregation function between sNIC and host and stores flow records in the host without loss. Trumpet [67] and Pathdump [78] offload query processing to end-hosts, limiting its processing capacity. NetWarden[84] and FlowLens[33] can carryout website fingerprinting and detect covert timing channels, but consume significant data plane memory and control plane CPU resources. SmartWatch significantly reduces this.

Sketches: Sketch-based solutions like Elastic [85], MVSketch [80], Univmon [64], Sketchlearn [51], and NitroSketch [63] focus on identifying heavy hitters and heavy change accurately, and less on low rate or low volume attacks. SmartWatch instead seeks to ensure all the flows are tracked, to support both volumetric analysis and specialized monitoring tasks including slow-rate attack detection and prevention. The sNIC data structure in SmartWatch adapts to packet arrival rate, ensuring loss-free, near line-rate processing.

NIDS over SmartNICs: Pigasus[91] and others [32, 36, 77, 79, 87] support high rate pattern matching capabilities. This paper assumes that data center traffic is encrypted[3], making these NIDS not valid[58]. We only focus on a generic caching algorithm to support anomaly detectors based on traffic analysis.

7 CONCLUSIONS

Providing a comprehensive monitoring infrastructure that can detect stealthy attacks in the midst of high traffic volumes is a challenge. State-of-the-art monitoring techniques either detect stealthy attacks at very low packet rates or limit their detection capabilities to volumetric attacks for high packet arrival rates. SmartWatch bridges this dichotomy by cooperatively splitting up the monitoring tasks between P4 programmable networking switches, P4-capable SmartNICs and the host. Our proposed control loop helps avoid having to make a trade-off between detection rate vs. packet processing rate. Furthermore, SmartWatch helps reduce the SRAM memory pressure on programmable switches, by reducing the required state on the switches. On the other hand, the SmartNIC helps reduce the packet processing latency even further, by offloading flow-state tracking and flow-logging tasks from the host to the network-centric components of sNIC and P4Switch. In summary, SmartWatch selects the correct monitoring-granularity and monitoring-target to detect both volumetric and stealthy attacks. SmartWatch's network switch and host co-design for cooperative monitoring yields 2.39 times better detection rate compared to just programmable switches, thanks to SmartWatch's fined-grained processing without compromising packet processing throughput. Compared to host-based fine-grained approaches, SmartWatch reduces the packet processing latency by 72.32%.

8 ACKNOWLEDGEMENTS

We sincerely thank the US NSF for their generous support through grants CRI-1823270, CNS-1618030 and CCF-1815643. We thank our shepherd Professor Salvatore Pontarelli and the anonymous reviewers for their suggestions and comments. We also thank Victor G. Hill, Aditya Dhakal, Vivek Jain, and Viyom Mittal for invaluable help setting up the testbed, including the sNIC.

REFERENCES

[1] 2020 DDOS attacks. https://www.globenewswire.com/news-release/2020/06/23/2052054/0/en/Imperva-Research-Labs-Reveals-Abnormal-Increase-in-DDoS-Attack-Length-Despite-Popularity-of-Short-Term-Attacks.html

[2] ARM Cortex-A72 Architecture. https://www.tomshardware.com/reviews/arm-cortex-a72-architecture,4424.html

[3] Azure Encryption. https://docs.microsoft.com/en-us/azure/security/fundamentals/encryption-overview

[4] The CAIDA Anonymized Internet Traces. http://www.caida.org/data/passive/passive_dataset.xml.

[5] Editcap. https://www.wireshark.org/docs/man-pages/editcap.html

[6] How Zeek can provide insights despite encrypted communications. https://corelight.blog/2019/05/07/how-zeek-can-provide-insights-despite-encrypted-communications/.

[7] Intel Tofino 2. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html

[8] Intel XL710 Datasheet. https://www.intel.com/content/www/us/en/design/products-and-solutions/networking-and-io/ethernet-controller-xl710/technical-library.html?grouping=EMT_Content%20Type&sort=title:asc

[9] Mapping P4 to SmartNICs. https://opennetworking.org/wp-content/uploads/2020/12/p4_d2_2017_nfp_architecture.pdf

[10] Marvell LiquidIO III. https://www.marvell.com/content/dam/marvell/en/public-collateral/embedded-processors/marvell-liquidio-III-solutions-brief.pdf

[11] Marvell User Guide. https://www.marvell.com/content/dam/marvell/en/public-collateral/ethernet-adaptersandcontrollers/marvell-ethernet-adapters-fastlinq-41000-series-user-guide.pdf

[12] Mergecap. https://www.wireshark.org/docs/man-pages/mergecap.html

[13] Netronome Agilio LX. https://www.netronome.com/media/documents/PB_Agilio_LX_2x40GbE-7-20.pdf

[14] Netronome MicroC. https://cdn.open-nfp.org/media/documents/the-joy-of-micro-c_fcjSfra.pdf

[15] Netronome NFP-6000 Flow Processor. https://www.netronome.com/media/documents/PB_NFP-6000-7-20.pdf

[16] NMAP. https://nmap.org/

[17] NVIDIA Mellanox BlueField. https://www.mellanox.com/files/doc-2020/pb-bluefield-vpi-smart-nic.pdf

[18] OpenSOC. http://opensoc.github.io/

[19] PROFILING CPU USAGE IN REAL TIME WITH PERF TOP. https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/monitoring_and_managing_system_status_and_performance/profiling-cpu-usage-in-real-time-with-top_monitoring-and-managing-system-status-and-performance

[20] Redis. https://redis.io/

[21] Slowloris. https://www.cloudflare.com/learning/ddos/ddos-attack-tools/slowloris/

[22] TCPRewrite. https://linux.die.net/man/1/tcprewrite

[23] Towards Machine Learning Inference in the Data Plane. https://www.diva-portal.org/smash/get/diva2:1328601/FULLTEXT01.pdf

[24] Zeek FTP Bruteforcing. https://docs.zeek.org/en/v3.2.4/scripts/policy/protocols/ftp/detect-bruteforcing.zeek.html

[25] Zeek HTTP Stalling Detection. https://github.com/corelight/http-stalling-detector

[26] Zeek Kerberos. https://docs.zeek.org/en/master/scripts/base/protocols/krb/main.zeek.html

[27] The Zeek Network Security Monitor. https://www.zeek.org/

[28] Zeek SSH Bruteforcing Script. https://docs.zeek.org/en/current/scripts/policy/protocols/ssh/detect-bruteforcing.zeek.html

[29] Zeek SSL Certificate. https://docs.zeek.org/en/lts/scripts/policy/protocols/ssl/expiring-certs.zeek.html

[30] 2008. *Kolmogorov–Smirnov Test.* Springer New York, New York, NY, 283–287. https://doi.org/10.1007/978-0-387-32833-1_214

[31] Fabien André, Stéphane Gouache, Nicolas Le Scouarnec, and Antoine Monsifrot. 2018. Don't share, Don't lock: Large-scale Software Connection Tracking with Krononat. In *2018 USENIX Annual Technical Conference (USENIX ATC 18).* USENIX Association, Boston, MA, 453–466. https://www.usenix.org/conference/atc18/presentation/andre

[32] Zachary K. Baker and Viktor K. Prasanna. 2005. High-Throughput Linked-Pattern Matching for Intrusion Detection Systems. In *Proceedings of the 2005 ACM Symposium on Architecture for Networking and Communications Systems (ANCS '05).* Association for Computing Machinery, New York, NY, USA, 193–202. https://doi.org/10.1145/1095890.1095918

[33] Diogo Barradas, Nuno Santos, Luís Rodrigues, S. Signorello, Fernando M. V. Ramos, and André Madeira. 2021. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *NDSS.*

[34] Ran Ben Basat, Sivaramakrishnan Ramanathan, Yuliang Li, Gianni Antichi, Minian Yu, and Michael Mitzenmacher. 2020. PINT: Probabilistic In-Band Network Telemetry. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20).* Association for Computing Machinery, New York, NY, USA, 662–680. https://doi.org/10.1145/3387514.3405894

[35] Theophilus Benson, Aditya Akella, and David A. Maltz. 2010. Network Traffic Characteristics of Data Centers in the Wild. In *Proceedings of the 10th ACM SIGCOMM Conference on Internet Measurement (IMC '10).* ACM, New York, NY, USA, 267–280. https://doi.org/10.1145/1879141.1879175

[36] Milan Češka, Vojtech Havlena, Lukáš Holík, Jan Korenek, Ondrej Lengál, Denis Matoušek, Jirí Matoušek, Jakub Semric, and Tomáš Vojnar. 2019. Deep Packet Inspection in FPGAs via Approximate Nondeterministic Automata. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM).* 109–117. https://doi.org/10.1109/FCCM.2019.00025

[37] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, and Ori Rottenstreich. 2018. Catching the Microburst Culprits with Snappy. In *Proceedings of the Afternoon Workshop on Self-Driving Networks (SelfDN 2018).* Association for Computing Machinery, New York, NY, USA, 22–28. https://doi.org/10.1145/3229584.3229586

[38] Xiaoqi Chen, Shir Landau Feibish, Yaron Koral, Jennifer Rexford, Ori Rottenstreich, Steven A Monetti, and Tzuu-Yi Wang. 2019. Fine-Grained Queue Measurement in the Data Plane. In *Proceedings of the 15th International Conference on Emerging Networking Experiments And Technologies (CoNEXT '19).* Association for Computing Machinery, New York, NY, USA, 15–29. https://doi.org/10.1145/3359989.3365408

[39] Xiaoqi Chen, Shir Landau-Feibish, Mark Braverman, and Jennifer Rexford. 2020. BeauCoup: Answering Many Network Traffic Queries, One Memory Update at a Time. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20).* Association for Computing Machinery, New York, NY, USA, 226–239. https://doi.org/10.1145/3387514.3405865

[40] B. Claise. 2004. *Cisco Systems NetFlow Services Export Version 9.* RFC 3954. RFC Editor. http://www.rfc-editor.org/rfc/rfc3954.txt http://www.rfc-editor.org/rfc/rfc3954.txt.

[41] Chuck Cranor, Theodore Johnson, Oliver Spataschek, and Vladislav Shkapenyuk. 2003. Gigascope: A Stream Database for Network Applications. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD '03).* Association for Computing Machinery, New York, NY, USA, 647–651. https://doi.org/10.1145/872757.872838

[42] Michael Dalton, David Schultz, Jacob Adriaens, Ahsan Arefin, Anshuman Gupta, Brian Fahs, Dima Rubinstein, Enrique Cauich Zermeno, Erik Rubow, James Alexander Docauer, Jesse Alpert, Jing Ai, Jon Olson, Kevin DeCabooter, Marc de Kruijf, Nan Hua, Nathan Lewis, Nikhil Kasinadhuni, Riccardo Crepaldi, Srinivas Krishnan, Subbaiah Venkata, Yossi Richter, Uday Naik, and Amin Vahdat. 2018. Andromeda: Performance, Isolation, and Velocity at Scale in Cloud Network Virtualization. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 373–387. https://www.usenix.org/conference/nsdi18/presentation/dalton

[43] Shi Dong, Khushnood Abbas, and Raj Jain. 2019. A Survey on Distributed Denial of Service (DDoS) Attacks in SDN and Cloud Computing Environments. *IEEE Access* 7 (2019), 80813–80828. https://doi.org/10.1109/ACCESS.2019.2922196

[44] Nick Duffield, Carsten Lund, Mikkel Thorup, and Mikkel Thorup. 2003. Estimating Flow Distributions from Sampled Flow Statistics. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03).* ACM, New York, NY, USA, 325–336. https://doi.org/10.1145/863955.863992

[45] Paul Emmerich, Sebastian Gallenmüller, Daniel Raumer, Florian Wohlfart, and Georg Carle. 2015. MoonGen: A Scriptable High-Speed Packet Generator. In *Internet Measurement Conference 2015 (IMC'15).* Tokyo, Japan.

[46] Cristian Estan and George Varghese. 2001. New directions in traffic measurement and accounting. *ACM SIGCOMM Computer Communication Review* 32, 75–80. https://doi.org/10.1145/505202.505212

[47] Yixiao Feng, Sourav Panda, Sameer G Kulkarni, K. K. Ramakrishnan, and Nick Duffield. 2020. A SmartNIC-Accelerated Monitoring Platform for In-band Network Telemetry. In *2020 IEEE International Symposium on Local and Metropolitan Area Networks (LANMAN.* 1–6. https://doi.org/10.1109/LANMAN49260.2020.9153279

[48] Daniel Firestone, Andrew Putnam, Sambhrama Mundkur, Derek Chiou, Alireza Dabagh, Mike Andrewartha, Hari Angepat, Vivek Bhanu, Adrian Caulfield, Eric Chung, Harish Kumar Chandrappa, Somesh Chaturmohta, Matt Humphrey, Jack Lavier, Norman Lam, Fengfen Liu, Kalin Ovtcharov, Jitu Padhye, Gautham Popuri, Shachar Raindel, Tejas Sapre, Mark Shaw, Gabriel Silva, Madhan Sivakumar, Nisheeth Srivastava, Anshuman Verma, Qasim Zuhair, Deepak Bansal, Doug Burger, Kushagra Vaid, David A. Maltz, and Albert Greenberg. 2018. Azure Accelerated Networking: SmartNICs in the Public Cloud. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18).* USENIX Association, Renton, WA, 51–66. https://www.usenix.org/conference/nsdi18/presentation/firestone

[49] Linux Foundation. Data Plane Development Kit (DPDK). http://www.dpdk.org

[50] Arpit Gupta, Rob Harrison, Marco Canini, Nick Feamster, Jennifer Rexford, and Walter Willinger. 2018. Sonata: Query-driven Streaming Network Telemetry. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 357–371. https://doi.org/10.1145/3230543.3230555

[51] Qun Huang, Patrick Lee, and Yungang Bao. 2018. Sketchlearn: relieving user burdens in approximate measurement with automated statistical inference. 576–590. https://doi.org/10.1145/3230543.3230559

[52] Qun Huang, Haifeng Sun, Patrick P. C. Lee, Wei Bai, Feng Zhu, and Yungang Bao. 2020. OmniMon: Re-Architecting Network Telemetry with Resource Efficiency and Full Accuracy. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 404–421. https://doi.org/10.1145/3387514.3405877

[53] Mobin Javed and Vern Paxson. 2013. Detecting stealthy, distributed SSH brute-forcing. 85–96. https://doi.org/10.1145/2508859.2516719

[54] Raj Joshi, Ting Qu, Mun Choon Chan, Ben Leong, and Boon Thau Loo. 2018. BurstRadar: Practical Real-Time Microburst Monitoring for Datacenter Networks. In *Proceedings of the 9th Asia-Pacific Workshop on Systems (APSys '18)*. Association for Computing Machinery, New York, NY, USA, Article Article 8, 8 pages. https://doi.org/10.1145/3265723.3265731

[55] Jaeyeon Jung, V. Paxson, Arthur Berger, and Hari Balakrishnan. 2004. Fast portscan detection using sequential hypothesis testing. 211 – 225. https://doi.org/10.1109/SECPRI.2004.1301325

[56] Georgios Kambourakis, Tassos Moschos, Dimitris Geneiatakis, and Stefanos Gritzalis. 2007. Detecting DNS Amplification Attacks. In *Proceedings of the Second International Conference on Critical Information Infrastructures Security (CRITIS'07)*. Springer-Verlag, Berlin, Heidelberg, 185–196. https://doi.org/10.1007/978-3-540-89173-4_16

[57] Daehyeok Kim, Zaoxing Liu, Yibo Zhu, Changhoon Kim, Jeongkeun Lee, Vyas Sekar, and Srinivasan Seshan. 2020. TEA: Enabling State-Intensive Network Functions on Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20)*. Association for Computing Machinery, New York, NY, USA, 90–106. https://doi.org/10.1145/3387514.3405855

[58] Tiina Kovanen, Gil David, and Timo Hämäläinen. 2016. Survey: Intrusion Detection Systems in Encrypted Traffic. In *Internet of Things, Smart Spaces, and Next Generation Networks and Systems*, Olga Galinina, Sergey Balandin, and Yevgeni Koucheryavy (Eds.). Springer International Publishing, Cham, 281–293.

[59] Balachander Krishnamurthy, Subhabrata Sen, Yin Zhang, and Yan Chen. 2003. Sketch-based change detection: Methods, evaluation, and applications. In *Proceedings of the 2003 ACM SIGCOMM Internet Measurement Conference, IMC 2003*. 234–247.

[60] Aleksandar Kuzmanovic and Edward W. Knightly. 2003. Low-Rate TCP-Targeted Denial of Service Attacks: The Shrew vs. the Mice and Elephants. In *Proceedings of the 2003 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications (SIGCOMM '03)*. Association for Computing Machinery, New York, NY, USA, 75–86. https://doi.org/10.1145/863955.863966

[61] Jihyung Lee, Sungryoul Lee, Junghee Lee, Yung Yi, and KyoungSoo Park. 2015. FloSIS: A Highly Scalable Network Flow Capture System for Fast Retrieval and Storage Efficiency. In *Proceedings of the 2015 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '15)*. USENIX Association, USA, 445–457.

[62] Ming Liu, Tianyi Cui, Henry Schuh, Arvind Krishnamurthy, Simon Peter, and Karan Gupta. 2019. Offloading Distributed Applications Onto smartNICs Using iPipe. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 318–333. https://doi.org/10.1145/3341302.3342079

[63] Zaoxing Liu, Ran Ben-Basat, Gil Einziger, Yaron Kassner, Vladimir Braverman, Roy Friedman, and Vyas Sekar. 2019. Nitrosketch: Robust and General Sketch-based Monitoring in Software Switches. In *Proceedings of the ACM Special Interest Group on Data Communication (SIGCOMM '19)*. ACM, New York, NY, USA, 334–350. https://doi.org/10.1145/3341302.3342076

[64] Zaoxing Liu, Antonis Manousis, Gregory Vorsanger, Vyas Sekar, and Vladimir Braverman. 2016. One Sketch to Rule Them All: Rethinking Network Flow Monitoring with UnivMon. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. ACM, New York, NY, USA, 101–114. https://doi.org/10.1145/2934872.2934906

[65] Rui Miao, Hongyi Zeng, Changhoon Kim, Jeongkeun Lee, and Minlan Yu. 2017. SilkRoad: Making Stateful Layer-4 Load Balancing Fast and Cheap Using Switching ASICs. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 15–28. https://doi.org/10.1145/3098822.3098824

[66] YoungGyoun Moon, SeungEon Lee, Muhammad Asim Jamshed, and KyoungSoo Park. 2020. AccelTCP: Accelerating Network Applications with Stateful TCP Offloading. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. USENIX Association, Santa Clara, CA, 77–92. https://www.usenix.org/conference/nsdi20/presentation/moon

[67] Masoud Moshref, Minlan Yu, Ramesh Govindan, and Amin Vahdat. 2016. Trumpet: Timely and Precise Triggers in Data Centers. In *Proceedings of the 2016 ACM SIGCOMM Conference (SIGCOMM '16)*. Association for Computing Machinery, New York, NY, USA, 129–143. https://doi.org/10.1145/2934872.2934879

[68] Srinivas Narayana, Anirudh Sivaraman, Vikram Nathan, Prateesh Goyal, Venkat Arun, Mohammad Alizadeh, Vimalkumar Jeyakumar, and Changhoon Kim. 2017. Language-Directed Hardware Design for Network Performance Monitoring. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17)*. Association for Computing Machinery, New York, NY, USA, 85–98. https://doi.org/10.1145/3098822.3098829

[69] R. Neugebauer, G. Antichi, J. F. Zazo, Yury Audzevich, S. López-Buedo, and A. Moore. 2018. Understanding PCIe performance for end host networking. *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication* (2018).

[70] Andriy Panchenko, Lukas Niessen, Andreas Zinnen, and Thomas Engel. 2011. Website Fingerprinting in Onion Routing Based Anonymization Networks. In *Proceedings of the 10th Annual ACM Workshop on Privacy in the Electronic Society (WPES '11)*. Association for Computing Machinery, New York, NY, USA, 103–114. https://doi.org/10.1145/2046556.2046570

[71] Haakon Ringberg, Augustin Soule, Jennifer Rexford, and Christophe Diot. 2007. Sensitivity of PCA for Traffic Anomaly Detection. In *Proceedings of the 2007 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '07)*. Association for Computing Machinery, New York, NY, USA, 109–120. https://doi.org/10.1145/1254882.1254895

[72] Volker Roth and Randy H. Katz. 2004. Listen and Whisper: Security Mechanisms for BGP. In *First Symposium on Networked Systems Design and Implementation (NSDI 04)*. USENIX Association, San Francisco, CA. https://www.usenix.org/conference/nsdi-04/listen-and-whisper-security-mechanisms-bgp

[73] F. Silveira and C. Diot. 2010. URCA: Pulling out Anomalies by their Root Causes. In *2010 Proceedings IEEE INFOCOM*. 1–9. https://doi.org/10.1109/INFCOM.2010.5462151

[74] Fernando Silveira, Christophe Diot, Nina Taft, and Ramesh Govindan. 2010. ASTUTE: Detecting a Different Class of Traffic Anomalies. *SIGCOMM Comput. Commun. Rev.* 40, 4 (Aug. 2010), 267–278. https://doi.org/10.1145/1851275.1851215

[75] Sumeet Singh, C. Estan, G. Varghese, and S. Savage. 2005. The EarlyBird System for Real-time Detection of Unknown Worms.

[76] John Sonchack, Adam J. Aviv, Eric Keller, and Jonathan M. Smith. 2018. Turboflow: Information Rich Flow Record Generation on Commodity Switches. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, Article 11, 16 pages. https://doi.org/10.1145/3190508.3190558

[77] Haoyu Song, T. Sproull, M. Attig, and J. Lockwood. 2005. Snort offloader: a reconfigurable hardware NIDS filter. In *International Conference on Field Programmable Logic and Applications, 2005*. 493–498. https://doi.org/10.1109/FPL.2005.1515770

[78] Praveen Tammana, Rachit Agarwal, and Myungjin Lee. 2016. Simplifying Datacenter Network Debugging with PathDump. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 233–248. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/tammana

[79] Lin Tan and T. Sherwood. 2005. A high throughput string matching architecture for intrusion detection and prevention. In *32nd International Symposium on Computer Architecture (ISCA'05)*. 112–122. https://doi.org/10.1109/ISCA.2005.5

[80] L. Tang, Q. Huang, and P. P. C. Lee. 2019. MV-Sketch: A Fast and Compact Invertible Sketch for Heavy Flow Detection in Network Data Streams. In *IEEE INFOCOM 2019 - IEEE Conference on Computer Communications*. 2026–2034. https://doi.org/10.1109/INFOCOM.2019.8737499

[81] G. Varghese and Tony Lauck. 1987. Hashed and hierarchical timing wheels: Data structures for the efficient implementation of a timer facility. *ACM SIGOPS Operating Systems Review* 21 (11 1987), 25–38. https://doi.org/10.1145/37499.37504

[82] Nicholas Weaver, Robin Sommer, and Vern Paxson. 2009. Detecting forged TCP reset packets.

[83] S. Woo and K. Park. 2012. Scalable TCP Session Monitoring with Symmetric Receive-side Scaling.

[84] Jiarong Xing, Qiao Kang, and Ang Chen. 2020. NetWarden: Mitigating Network Covert Channels while Preserving Performance. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 2039–2056. https://www.usenix.org/conference/usenixsecurity20/presentation/xing

[85] Tong Yang, Jie Jiang, Peng Liu, Qun Huang, Junzhi Gong, Yang Zhou, Rui Miao, Xiaoming Li, and Steve Uhlig. 2018. Elastic Sketch: Adaptive and Fast Network-wide Measurements. In *Proceedings of the 2018 Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '18)*. ACM, New York, NY, USA, 561–575. https://doi.org/10.1145/3230543.3230544

[86] Minlan Yu, Lavanya Jose, and Rui Miao. 2013. Software Defined Traffic Measurement with OpenSketch. In *Presented as part of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI 13)*. USENIX, Lombard, IL, 29–42. https://www.usenix.org/conference/nsdi13/technical-sessions/presentation/yu

[87] Ruan Yuan, Yang Weibing, Chen Mingyu, Zhao Xiaofang, and Fan Jianping. 2010. Robust TCP Reassembly with a Hardware-Based Solution for Backbone Traffic. In *2010 IEEE Fifth International Conference on Networking, Architecture, and Storage.* 439–447. https://doi.org/10.1109/NAS.2010.53

[88] Yifei Yuan, Dong Lin, Ankit Mishra, Sajal Marwaha, Rajeev Alur, and Boon Thau Loo. 2017. Quantitative Network Monitoring with NetQRE. In *Proceedings of the Conference of the ACM Special Interest Group on Data Communication (SIGCOMM '17).* Association for Computing Machinery, New York, NY, USA, 99–112. https://doi.org/10.1145/3098822.3098830

[89] Kaiyuan Zhang, Danyang Zhuo, and Arvind Krishnamurthy. 2020. Gallium: Automated Software Middlebox Offloading to Programmable Switches. In *Proceedings of the Annual Conference of the ACM Special Interest Group on Data Communication on the Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '20).* Association for Computing Machinery, New York, NY, USA, 283–295. https://doi.org/10.1145/3387514.3405869

[90] Qiao Zhang, Vincent Liu, Hongyi Zeng, and Arvind Krishnamurthy. 2017. High-Resolution Measurement of Data Center Microbursts. In *Proceedings of the 2017 Internet Measurement Conference (IMC '17).* Association for Computing Machinery, New York, NY, USA, 78–85. https://doi.org/10.1145/3131365.3131375

[91] Zhipeng Zhao, Hugo Sadok, Nirav Atre, James C. Hoe, Vyas Sekar, and Justine Sherry. 2020. Achieving 100Gbps Intrusion Prevention on a Single Server. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20).* USENIX Association, 1083–1100. https://www.usenix.org/conference/osdi20/presentation/zhao-zhipeng

[92] Zhongliu Zhuo, Yang Zhang, Zhi-li Zhang, Xiaosong Zhang, and Jingzhong Zhang. 2018. Website Fingerprinting Attack on Anonymity Networks Based on Profile Hidden Markov Model. *IEEE Transactions on Information Forensics and Security* 13, 5 (2018), 1081–1095. https://doi.org/10.1109/TIFS.2017.2762825

9 APPENDIX

While we believe the main body of the paper is self-contained, we have included a number of additional details here in the Appendix for the sake of completeness, which we did not include in the main part of the paper (both to maintain the flow of the paper, and the limit on the length of the paper).

9.1 Flow record update scheme on sNIC

A global load balancer on the sNIC distributes incoming packets among a large number of PMEs, which allows different PMEs (each with 4 concurrent threads) to process incoming packets *in parallel*. Multiple PMEs may process packets for the same flow, resulting in:

- multiple PMEs update an existing flow's counter in $\mathbb{P}$.
- acquire a bucket in $\mathbb{P}$ to insert a new flow and possibly evict one of the existing entries from $\mathbb{P}$ and $\mathbb{E}$ to make room for the new flow in $\mathbb{P}$.

To ensure correctness, we need to guarantee there are no duplicate flow entries in a row and all memory updates (*e.g.,* counters) are serialized, avoiding overwrites. However, a naive approach using a lock per row in the hash table severely impacts performance. Since a train of packets of the same flow arrive often, several PMEs may contend for the lock. Further, a lock per row results in unnecessary serialization when PMEs need to update different bucket entries in a row. We distinguish two kinds of operations demanding distinct serialization for the two scenarios mentioned above.

Lockless updates on sNIC: For an update operation, we only need to ensure memory updates for a bucket within a row are serialized. We designed a lock free update scheme to allow multiple PMEs to update the packet count, timestamp, or other variables in parallel by taking advantage of the *atomic write* memory operations supported by the sNIC to serialize memory operations in hardware. However, a bucket with one or more ongoing update operations must not be evicted by another PME. Thus, we also keep a counter `up_th_ctr` for tracking of the number of threads across different PMEs that are concurrently accessing the same bucket.

Flow record Insertion and Eviction in sNIC: When the flow can not find a match in $\mathbb{P}$, we probe $\mathbb{E}$ and accordingly select the eviction candidates from both $\mathbb{P}$ and $\mathbb{E}$ to insert the new flow in $\mathbb{P}$. For correctness:

- only one PME performs the insertion of a new flow and evict an existing flow entry in a row.
- no other threads are currently performing updates.

Before inserting a flow entry, a PME first acquires exclusive access to the row using the lockless *test-and-set* hardware primitive and check to ensure no other threads are waiting. To avoid the write thread of a PME from being stalled by packets performing updates on the eviction candidate in $\mathbb{P}$, we preclude any further updates on the eviction candidate. After this the eviction candidate is evicted and replaced. Meanwhile subsequent updates of the recently evicted flow fallback to inserting the flow entry. However, this is extremely unlikely as the least recently used flow record is selected for eviction.

9.2 Flow Record Update

Algorithm 2 Flow records Operation

```
1:  procedure UPDATE($f$)
2:      if $P\_hit$ then
3:          if $fetch\_and\_add(up\_th\_ctr, 1) == 0$ then
4:              while $test\_and\_set(mt\_excl\_inst) == 1$ do
5:          $f_c \leftarrow f_c + 1$                ▷ increment counters of flow $f$
6:          if $fetch\_and\_sub(up\_th\_ctr, 1) == 1$ then
7:              $mt\_excl\_inst \leftarrow 0$
8:      else
9:          INSERT($f$)
10: procedure INSERT($f$)
11:     $\tilde{p} \leftarrow new\_flow$                    ▷ $E\_Hit$ or $Miss$
12:     while $test\_and\_set(row) == 1$ do
13:     $key \leftarrow 0$                 ▷ stop further update on this entry
14:     while $test\_and\_set(mt\_excl\_inst) == 1$ do
15:     $E \leftarrow p_{victim}$             ▷ write victim flow from $P$ into $E$
16:     $p_{victim} \leftarrow \tilde{p}$                    ▷ insert new flow into $P$
17:     $mt\_excl\_inst \leftarrow 0, row \leftarrow 0$
```

Algorithm 2 is used to select an eviction candidate and swap entries in the Primary P and Eviction E buffers. It then conditionally moves the eviction candidate entry from E to the ring buffer R in a lockless manner. Variable up_th_ctr tracks the number of threads accessing the flow entry. Variable mt_excl_inst is 1 if at least one thread accesses the flow entry, otherwise 0. Variable row is set to 1 by the thread that has exclusive access to the row for the purposed of evicting flow entries. Atomic operations include: atomic increment $fetch\_and\_add$, atomic decrement $fetch\_and\_sub$, and an instruction that writes to a memory location and return its old value atomically $test\_and\_set$.

The algorithm updates flow entries in the Primary Buffer P in a lockless manner, allowing multiple PMEs to update the flow counters in parallel. In the event of a flow entry miss, a new flow entry is inserted in the Primary Buffer while ensuring there are no other PMEs simultaneously updating the flow attribute of the Primary Buffer eviction candidate. This Primary Buffer eviction candidate replaces an Eviction Buffer victim and the Eviction Buffer victim is sent to the Ring Buffer that is read by the host. In the event of an Eviction Buffer hit, the Primary Buffer victim is swapped against the incoming packet's flow entry.

9.3 Cleaning the Hash Row

Algorithm 3 Clean

```
1: Initialize:
       $size[bins] \leftarrow 0, bins = 0, \ldots, P-1$
       $temp[bins][cap] \leftarrow -1, bins = 0, \ldots, P-1,$
                                  $cap = 0, \ldots, S-1$
       $to\_evict[i] \leftarrow -1, i = 0, \ldots, row\_len - 1$
2: procedure CLEAN($B, P, S$)
3:     $e \leftarrow 0$
4:     for $i \leftarrow 0, row\_len$ do
5:         $new\_bin \leftarrow hash\_bin(B_i)$
6:         if $size[new\_bin] < S$ then
7:             $temp[new\_bin][size[new\_bin]] \leftarrow i$
8:             $size[new\_bin] \leftarrow size[new\_bin] + 1$
9:         else
10:            $oldest \leftarrow$ GETOLDEST$(temp[new\_bin], i)$
11:            if $oldest \neq i$ then
12:                $to\_evict[e] \leftarrow oldest$
13:                SWAP$(oldest, i)$
14:            else
15:                $to\_evict[e] \leftarrow i, e \leftarrow e + 1$
```

Cleanup process: We leverage the 'dirty bit' marking by the CME to determine the need to re-order the bucket entries across the Lite mode rows according to the higher order bits of the hash digest. The first PME that finds a bucket with the 'dirty' bit (based on the hash $h(K)$ of the incoming packet) will perform the cleanup operation described in the Algorithm 3. First, the PME gains exclusive access to the row, then locks all of the buckets in the row and copies over the bucket entries to a local buffer B. Thereafter, it iteratively computes the hash and identifies the right Lite row based on the higher order bits. If there are collisions, we retain the recent entry in the hash table and evict the oldest of the entries to the ring buffers to be moved to the host. Since the cleanup is performed in the critical (packet processing) path by the PMEs, The key to this algorithm is that it is necessary to perform cleanup with very little overhead.

We use three local variables *size*, *temp*, and *to_evict* to store the occupied size of each row, the bucket index we wish to store in that Lite row and the buckets we wish to evict to the ring buffer, respectively. The 'Clean' procedure maps each of the buckets B_i into the correct Lite row *new_bin* and stores the entry currently in this bucket index into the corresponding Lite row, if it is not already occupied. Otherwise, only the recent packet entry (based on the latest update timestamp stored in the bucket) is preferred over the other entry (output of the GETOLDEST($temp[new\_bin], i$)); and the older entry is evicted to the ring buffer.

Algorithm 3 enables FlowCache to seamlessly transition from the General mode of operation to a Lite mode of operation.

9.4 Switch Over

Algorithm 4 is loaded on a CME that evaluates the exponential moving average on the packet arrival rate and then selects the appropriate mode of operation.

Algorithm 4 SwitchOver

```
1: procedure SWITCHOVER($A_t$)
2:     $F_{t+1} = \alpha A_t + (1 - \alpha) F_t$
3:     if $F_{t+1} > \eta_1$ then
4:         for $i \leftarrow 0, Row$ do
5:             $Dirty[i] \leftarrow 1$
6:         $Mode \leftarrow Lite\_mode$
7:     else if $F_{t+1} < \eta_2$ then
8:         $Mode \leftarrow General\_mode$
```

Note: The switchover from General to Lite happens only when the packet rate exceeds the configured thresholds.