SNIPS: A Software-Defined Approach for Scaling Intrusion Prevention Systems via Offloading

Victor Heorhiadi[1], Seyed Kaveh Fayaz[2], Michael K. Reiter[1], and Vyas Sekar[2]

[1] UNC Chapel Hill, Chapel Hill, NC, USA
[2] Carnegie Mellon University, Pittsburgh, PA, USA

Abstract. Growing traffic volumes and the increasing complexity of attacks pose a constant scaling challenge for network intrusion prevention systems (NIPS). In this respect, *offloading* NIPS processing to compute clusters offers an immediately deployable alternative to expensive hardware upgrades. In practice, however, NIPS offloading is challenging on three fronts in contrast to passive network security functions: (1) NIPS offloading can impact other traffic engineering objectives; (2) NIPS offloading impacts user perceived latency; and (3) NIPS actively change traffic volumes by dropping unwanted traffic. To address these challenges, we present the SNIPS system. We design a formal optimization framework that captures tradeoffs across scalability, network load, and latency. We provide a practical implementation using recent advances in software-defined networking without requiring modifications to NIPS hardware. Our evaluations on realistic topologies show that SNIPS can reduce the maximum load by up to 10× while only increasing the latency by 2%.

1 Introduction

Network intrusion prevention systems (NIPS) are an integral part of today's network security infrastructure [38]. However, NIPS deployments face a constant battle to handle increasing volumes and processing requirements. Today, network operators have few options to tackle NIPS overload – overprovisioning, dropping traffic, or reducing fidelity of the analysis. Unfortunately, none of these options are attractive in practice. Thus, NIPS scaling has been, and continues to be, an active area of research in the intrusion detection community with several efforts on developing better hardware and algorithms (e.g., [32, 34, 36, 39]). While these efforts are valuable, they require significant capital costs and face deployment delays as networks have 3–5 year hardware refresh cycles.

A promising alternative to expensive and delayed hardware upgrades is to *offload* packet processing to locations with spare compute capacity. Specifically, recent work has considered two types of offloading opportunities:

- *On-path offloading* exploits the natural replication of a packet on its route to distribute processing load [27, 28].
- *Off-path offloading* utilizes dedicated clusters or cloud providers to exploit the economies of scale and elastic scaling opportunities [9, 29].

Such offloading opportunities are appealing as they flexibly use existing network hardware and provide the ability to dynamically scale the deployment. Unfortunately,

A. Prakash and R. Shyamasundar (Eds.): ICISS 2014, LNCS 8880, pp. 9–29, 2014.
© Springer International Publishing Switzerland 2014

current solutions either explicitly focus on passive monitoring applications such as flow monitors and NIDS [9, 28] and ignore NIPS-induced effects, e.g., on traffic volumes [23,27,29]. Specifically, we observe three new challenges in NIPS offloading that fall outside the scope of these prior solutions:

- **Interaction with traffic engineering:** Offloading NIPS to a datacenter means that we are effectively *rerouting* the traffic. This may affect network congestion and other traffic engineering objectives.
- **Impact on latency:** NIPS lie on the *critical forwarding path* of traffic. Delays introduced by overloaded NIPS or the additional latency induced by offloading can thus affect the *latency* for user applications.
- **Traffic volume changes:** NIPS *actively* change the traffic volume routed through the network. Thus, the load on a NIPS node is dependent on the processing actions of the upstream nodes along the packet forwarding path.

To address these challenges and deliver the benefits of offloading to NIPS deployments, we present the SNIPS system. SNIPS takes a first-principles approach to capture the above effects and balance the tradeoffs across scalability, latency increase, and network congestion. Perhaps counterintuitively, we show that it is feasible to capture these complex requirements and effects through a linear programming (LP) formulation that is amenable to fast computation using off-the-shelf solvers. As we show in §7, the computation takes $\leq$2 seconds, for a variety of real topologies enabling SNIPS to react in near-real-time to network dynamics. The design of SNIPS is quite general and it can be used in many deployment settings and the ideas may also be applicable to other network functions virtualization (NFV) applications [17].

We leverage *software-defined networking* (SDN) mechanisms to implement the optimal strategy derived from the LP formulation. A key benefit of SDN is that it does not require modifications to the NIPS hardware or software unlike prior work [9,27]. Using trace-driven simulations and emulations, we show that SNIPS can reduce the maximum load by up to 10$\times$ while only increasing the latency by 2%.

Contributions: In summary, this paper makes four contributions:

- Identifying challenges in applying offloading to NIPS deployments (§3);
- Designing formal models to capture new effects (e.g., rerouting, latency, traffic changes) (§5);
- Addressing practical challenges in an SDN-based implementation (§6); and
- A detailed evaluation showing that SNIPS imposes low overhead and offers significant advantages (§7).

2 Related Work

On-Path Offloading: The key difference between SNIPS and prior work on on-path offloading [27, 28] is three-fold: (1) they focus only on on-path monitoring; (2) these assume that the traffic volume does not change inside the network; and (3) they are not concerned with latency. As a result, the models from these efforts do not apply as we

highlight in the next section. SNIPS considers a generalized model of both on- and off-path offloading, models the impact of rerouting on latency, and captures effects of NIPS actively changing the traffic volume. In terms of implementation, these efforts modify software platforms such as the `yaf` flow monitor and the Bro IDS [20]. In contrast, SNIPS leverages software-defined networking (SDN) to provide an in-network offloading solution that does not require access to the NIPS software source or the hardware platform. Thus, SNIPS can accommodate legacy and proprietary NIPS solutions.

Off-Path Offloading: Recent efforts make the case for virtualizing NIPS-like functions [7] and demonstrate the viability of off-path offloading using public cloud providers [29]. Our work shares the motivation to exploit elastic scaling and reduce capital/operating expenses. However, these efforts focus more on the high-level vision and viability. As such they do not provide formal models like SNIPS to capture trade-offs across scalability, network bandwidth costs, and user-perceived latency and incorporating the effects of active traffic dropping by NIPS. The closest work in off-path offloading is our prior work [9]. However, the focus there was on NIDS or *passive monitoring* and thus the traffic is simply replicated to clusters. As such, this prior work does not model rerouting or the impact on user-perceived latency. Furthermore, this implementation requires running a Click-based shim layer [12] below the NIDS, and thus cannot work with legacy NIDS/NIPS hardware. As discussed earlier, SNIPS provides an in-network solution via SDN that accommodates legacy NIPS.

Traditional NIPS/NIDS Scaling: There are several complementary approaches for scaling NIPS, including algorithmic improvements [32], using specialized hardware such as TCAMs (e.g., [15, 39]), FPGAs (e.g., [14]), or GPUs (e.g., [10, 35]). These are orthogonal to SNIPS as they improve single NIPS throughput, while SNIPS focuses on network-wide NIPS resource management.

Distributed NIPS: Prior work in distributed NIDS and NIPS [2, 13, 21] focus on correlating events, combining alerts from different vantage points, and extracting useful information from multiple vantage points. Our framework, in contrast, focuses on distribution primarily for scalability.

SDN and Security: Recent work has recognized the potential of SDN for security tasks; e.g., FRESCO uses SDN to simplify botnet or scan detection [31]. SIMPLE [23] and SoftCell [11] use SDN for steering traffic through a desired sequence of waypoints. These do not, however, model the impact of altering the traffic volume as in SNIPS. In addition, there are subtle issues in ensuring stateful processing and load balancing that these works do not address (see §6). Shin et al. highlight security concerns where reactive controllers that set up forwarding rules dynamically per-flow can get overloaded [30]. SNIPS uses proactive rule installation and is immune to such attacks.

3 Motivation and Challenges

We begin by briefly describing the idea of offloading for scaling *passive monitoring* solutions. Then, we highlight the challenges in using this idea for NIPS deployments that arise as a result of NIPS-specific aspects: NIPS *actively* modify the traffic volume and NIPS placement impacts the *end-to-end latency*.

3.1 Case for Offloading

Avoiding overload is an important part of NIPS management. Some NIPS processing is computationally intensive, and under high traffic loads, CPU resources become scarce. Modern NIPS offer two options for reacting to overload: dropping packets or suspending expensive analysis modules. Neither is an attractive option. For example, Snort by default drops packets when receiving more traffic than it can process — in tests in our lab, Snort dropped up to 30% of traffic when subjected to more traffic than it had CPU to analyze — which can adversely impact end-user performance (especially for TCP traffic). Suspending analysis modules decreases detection coverage. In fact, this behavior under overload can be used to evade NIPS [19]. As such, network operators today have few choices but to provision their NIDS/NIPS to handle maximum load. For example, they can upgrade their NIPS nodes with specialized hardware accelerators (e.g., using TCAM, GPUs, or custom ASICs). While this is a valid (if expensive) option, practical management constraints restrict network appliance upgrades to a 3–5 year cycle.

A practical alternative to avoid packet drops or loss in detection coverage is by exploiting opportunities for *offloading* the processing. Specifically, prior work has exploited this idea in the context of passive monitoring in two ways: (1) *on-path* offloading to other monitoring nodes on the routing path [27, 28] and (2) *off-path* offloading by replicating traffic to a remote datacenter [9, 29].

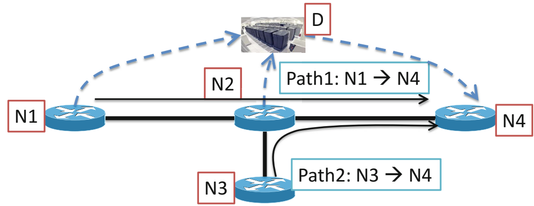

Fig. 1. An example to explain the on- and off-path offloading opportunities that have been proposed in prior work for passive monitoring solutions

To make these more concrete, consider the example network in Figure 1 with 4 nodes N1–N4, with traffic flowing on two end-to-end paths P1:N1 $\rightarrow$ N4 and P2:N3 $\rightarrow$ N4.[1] In a traditional deployment, each packet is processed at its *ingress* on each path: N1 monitors traffic on P1 and N3 monitors traffic on P2. An increase in the load on P1 or P2 can cause drops or detection misses

With on-path offloading, we can balance the processing load across the path (i.e., N1, N2, and N4 for P1 and N2, N3, and N4 for P2) to use spare capacity at N2 and N4 [27, 28]. This idea can be generalized to use processing capacity at off-path locations; e.g., N1 and N2 can offload some of their load to the datacenter; e.g., a NIDS cluster [34] or cloudbursting via public clouds [29].

3.2 Challenges in Offloading NIPS

Our goal is to extend the benefits of offloading to NIPS deployments. Unlike passive monitoring solutions, however, NIPS need to be *inline* on the forwarding path and they *actively drop* traffic. This introduces new dimensions for both on-path and off-path offloading that falls outside the scope of the aforementioned prior work.

[1] For brevity, in this section we use an abstract notion of a "node" that includes both the NIDS/NIPS functionality and the switching/routing function.

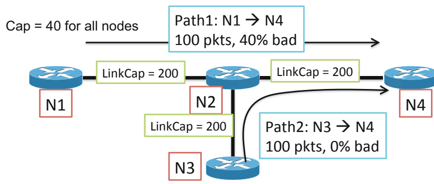

Fig. 2. Need to model the impact of inline traffic modifications

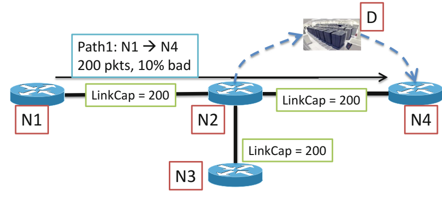

Fig. 3. Impact of rerouting to remote locations

Suppose we have a network administrator who wants to distribute the processing load across the different nodes to: (1) operate within the provisioned capacity of each node; (2) meet traffic engineering objectives w.r.t. link loads (e.g., ensure that no link is loaded to more than 30%); (3) minimize increased latency due to rerouting; and (4) ensures that the unwanted traffic is dropped as close to the origin as possible subject to (1), (2), and (3). We extend the example topology from earlier to highlight the key challenges that arise in meeting these goals

NIPS Change Traffic Patterns: In Figure 2, each NIPS N1–N4 can process 40 packets and each link has a capacity to carry 200 packets. Suppose P1 and P2 carry a total of 100 packets and the volume of unwanted traffic on P1 is 40%; i.e., if we had no NIPS resource constraints, we would drop 40% of the traffic on P1. In order to meet the NIPS load balancing and traffic engineering objectives, we need to model the effects of the traffic being dropped by each NIPS node. If we simply use the formulations for passive monitoring systems and ignore the traffic drop rate, we may incorrectly infer that there is no feasible solution—the total offered load of 200 packets exceeds the total NIPS capacity (160). Because P1 drops 40 packets, there is actually a feasible solution.

Rerouting: Next, let us consider the impact of off-path offloading to a datacenter. Here, we see a key difference between NIDS and NIPS offloading. With NIDS, we *replicate* traffic to the datacenter D. With NIPS, however, we need to actively *reroute* the traffic. In Figure 3, the traffic on P1 exceeds the total NIPS capacity even after accounting for the drop rate. In this case, we need to reroute a fraction of the traffic on P1 to the datacenter from N2. If we were replicating the traffic, then the load on the link N2-N4 would be unaltered. With rerouting, however, we are reducing the load on N2-N4 and introducing additional load on the links between N2 and D (and also between D and N4). This has implications for traffic engineering as we need to account for the impact of rerouting on link loads.

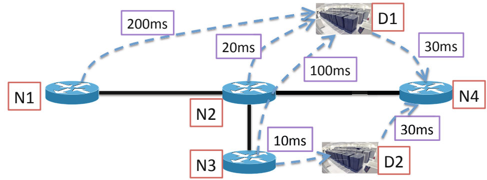

Fig. 4. Need to carefully select offload locations in order to account for the latency for user connections

Latency Addition Due to Offloading: NIDS do not actively impact user-perceived performance. By virtue of being on the critical forwarding path, however, NIPS offloading to remote locations introduces extra latency to and from the datacenter(s). In Figure 4, naively offloading traffic from N1 to D1 or from N3 to D1 can add hundreds

of milliseconds of additional latency. Because the latency is critical for interactive and web applications (e.g., [8]), we need systematic ways to model the impact of rerouting to minimize the impact on user experience.

Conflict with Early Dropping: Naive offloading may also increase the *footprint* of unwanted traffic as traffic that could have been dropped may consume extra network resources before it is eventually dropped. Naturally, operators would like to minimize this impact. Let us extend the previous scenario to case where the link loads are low, and D1 and D2 have significantly higher capacity than the on-path NIPS. From a pure load perspective, we might want to offload most of the traffic to D1 and D2. However, this is in conflict with the goal of dropping unwanted traffic early.

Together, these examples motivate the need for a systematic way to capture NIPS-specific aspects in offloading including: (1) changes to traffic patterns due to NIPS actions; (2) accounting for the impact of rerouting in network load; (3) modeling the impact of off-path offloading on latency for users; and (4) balancing the tension between load balancing and dropping unwanted traffic early.

4 SNIPS System Overview

In order to address the challenges from the previous section, we present the design of the SNIPS system. Figure 5 shows a high-level view of the system. The design of SNIPS is general and can be applied to several contexts: enterprise networks, datacenter networks, and ISPs, though the most common use-case (e.g., as considered by past network security literature) is typically for enterprise networks.

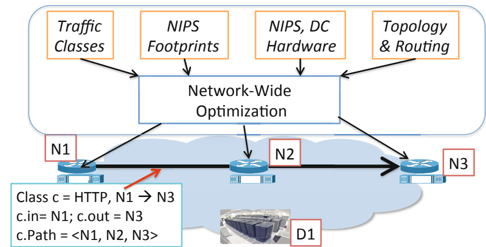

Fig. 5. Overview of the SNIPS architecture for NIPS offloading

We envision a logically centralized *controller* that manages the NIPS deployment as shown, analogous to many recent network management efforts (e.g., [3]). Network administrators specify high-level objectives such as bounds on acceptable link congestion or user-perceived latency. The controller runs a network-wide optimization and translates these high-level goals into physical data plane configurations.

This network-wide optimization is run periodically (e.g., every 5 minutes) or triggered by routing or traffic changes to adapt to network dynamics. To this end, it uses information about the current traffic patterns and routing policies using data feeds that are routinely collected for other network management tasks [5]. Based on these inputs, the controller runs the optimization procedures (described later) to assign NIPS processing responsibilities. We begin by describing the main inputs to this NIPS controller.

- **Traffic Classes:** Each *traffic class* is identified by a specific application-level port (e.g., HTTP, IRC) and network ingress and egress nodes. Each class is associated with some type of NIPS analysis that the network administrator wants to run. We use the variable c to identify a specific class. We use $c.in$ and $c.out$ to denote the ingress

and egress nodes for this traffic class; in particular, we assume that a traffic class has exactly one of each. For example, in Figure 5 we have a class c consisting of HTTP traffic entering at $\mathsf{c}.in = \mathsf{N1}$ and exiting at $\mathsf{c}.out = \mathsf{N3}$. Let $S(c)$ and $B(c)$ denote the (expected) volume of traffic in terms of the number of sessions and bytes, respectively. We use $Match(c)$ to denote the expected rate of unwanted traffic (which, for simplicity, we assume to be the same in sessions or bytes) on the class c, which can be estimated from summary statistics exported by the NIPS.

- **Topology and Routing:** The path traversed by traffic in a given class (before any rerouting due to offloading) is denoted by $c.path$. For clarity, we assume that the routing in the network is symmetric; i.e., the path $c.path = Path(c.in, c.out)$ is identical to the reverse of the path $Path(c.out, c.in)$. In our example, $\mathsf{c}.path = \langle \mathsf{N1}, \mathsf{N2}, \mathsf{N3} \rangle$. Our framework could be generalized to incorporate asymmetric routing as well. For simplicity, we restrict the presentation of our framework to assume symmetric routing.

  We use the notation $N_j \in Path(src, dst)$ to denote that the NIPS node N_j is *on the routing path* between the source node src and the destination node dst. In our example, this means that $\mathsf{N1}, \mathsf{N2}, \mathsf{N3} \in Path(\mathsf{N1}, \mathsf{N3})$. Note that some nodes (e.g., a dedicated cluster such as D1 in Figure 5) are off-path; i.e., these do not observe traffic unless we explicitly re-route traffic to them. Similarly, we use the notation $l \in Path(src, dst)$ to denote that the link l is on the path $Path(src, dst)$. We use $|Path(src, dst)|$ to denote the latency along a path $Path(src, dst)$. While our framework is agnostic to the units in which latency is measured, we choose hop-count for simplicity.
- **Resource Footprints:** Each class c may be subject to different types of NIPS analysis. For example, HTTP sessions may be analyzed by a payload signature engine and through web firewall rules. We model the cost of running the NIPS for each class on a specific *resource* r (e.g., CPU cycles, memory) in terms of the expected per-session resource footprint F_c^r, in units suitable for that resource (F_c^r for *Footprint* on r). These values can be obtained either via NIPS vendors' datasheets or estimated using offline benchmarks [4].
- **Hardware Capabilities:** Each NIPS hardware device N_j is characterized by its resource capacity Cap_j^r in units suitable for the resource r. In the general case, we assume that hardware capabilities may be different because of upgraded hardware running alongside legacy equipment.

We observe that each of these inputs (or the instrumentation required to obtain them) is already available in most network management systems. For instance, most centralized network management systems today keep a network information base (NIB) that has the current topology, traffic patterns, and routing policies [5]. Similarly, the hardware capabilities and resource footprints of the different traffic classes can be obtained with simple *offline* benchmarking tools [4]. Note that our assumption on the availability of these inputs is in line with existing work in the network management literature. The only additional input that SNIPS needs is $Match(c)$, which is the expected drop rate for the NIPS functions. These can be estimated using historical logs reported by the NIPS; anecdotal evidence from network administrators suggests that the match rates

are typically quite stable [1]. Furthermore, SNIPS can provide significant benefits even with coarse estimates. In this respect, our guiding principle is to err on the conservative side; e.g., we prefer to overestimate resource footprints and underestimate the drop rates.

Note that SNIPS does not compromise the security of the network relative to a traditional ingress-based NIPS deployment. That is, any malicious traffic that would be dropped by an ingress NIPS will also be dropped in SNIPS; this drop may simply occur elsewhere in the network as we will see.

Given this setup, we describe the optimization formulations for balancing the trade-off between the load on the NIPS nodes and the latency and congestion introduced by offloading.

5 SNIPS Optimization

Given the inputs from the previous section, our goal is to optimally distribute the NIPS processing through the network. To this end, we present a linear programming (LP) formulation. While LP-based solutions are commonly used in traffic engineering [6, 28], NIPS introduce new dimensions that make this model significantly different and more challenging compared to prior work [9, 28]. Specifically, rerouting and active manipulation make it challenging to systematically capture the effective link and NIPS loads using the optimization models from prior work, and thus we need a first-principles approach to model the NIPS-specific aspects.

Our formulation introduces decision variables that capture the notion of *processing and offloading fractions*. These variables, defined for each node along a routing path, control the number of flows processed at each node. Let $p_{c,j}$ denote the fraction of traffic on class c that the router N_j processes locally and let $o_{c,j,d}$ denote the fraction of traffic on class c that the NIPS node N_j offloads to the datacenter d. For clarity of presentation, we assume there is a single datacenter d and thus drop the d subscript; it is easy to generalize this formulation to multiple datacenters, though we omit the details here due to space considerations.

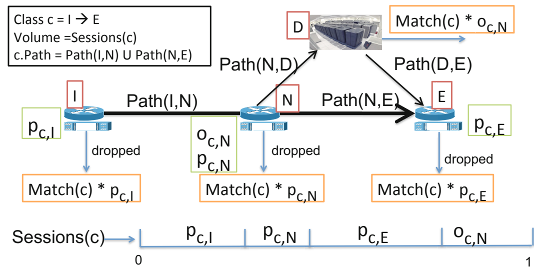

Fig. 6. An example to highlight the key concepts in our formulation and show modeling of the additional latency due to rerouting

Intuitively, we can imagine the set of traffic sessions belonging to class c entering the network (i.e., before any drops or rerouting) as being divided into non-overlapping buckets, e.g., either using hashing or dividing the traffic across prefix ranges [28,34,37]. The fractions $p_{c,j}$ and $o_{c,j}$ represent the length of these buckets as shown in Figure 6.

Figure 7 shows the optimization framework we use to systematically balance the trade-offs involved in NIPS offloading. We illustrate the key aspects of this formulation using the example topology in Figure 6 with a single class c of traffic flowing between

Minimize: $(1-\alpha-\beta) \times NLdCost + \alpha \times HopsUnwanted + \beta \times LatencyInc$, subject to:

$$\forall c: \sum_{N_j \in c.path} p_{c,j} + o_{c,j} = 1 \quad (1)$$

$$\forall r, j: NLd_{j,r} = \sum_{c: N_j \in c.path} p_{c,j} \times S(c) \times F_c^r \quad (2)$$

$$\forall r, j: NLd_{j,r} \leq Cap_j^r \quad (3)$$

$$\forall r: \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times F_c^r \leq DCap^r \quad (4)$$

$$\forall r, j: NLdCost \geq NLd_{j,r} \quad (5)$$

$$\forall l: BG_l = \sum_{c: l \in c.path} B(c) \quad (6)$$

$$\forall l: LLd_l \leq MaxLLd \times LCap_l \quad (7)$$

$$LatencyInc = \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times \begin{pmatrix} |Path(N_j, d)| + |Path(d, c.out)| \\ -|Path(N_j, c.out)| \end{pmatrix} \quad (8)$$

$$HopsUnwanted = \sum_{c} \sum_{N_j \in c.path} p_{c,j} \times S(c) \times Match(c) \times |Path(c.in, N_j)| + \sum_{c} \sum_{N_j \in c.path} o_{c,j} \times S(c) \times Match(c) \times \begin{pmatrix} |Path(c.in, N_j)| \\ +|Path(N_j, d)| \end{pmatrix} \quad (9)$$

$$\forall l: LLd_l = BG_l + \sum_{c} \sum_{\substack{N_j: N_j \in c.path \\ \wedge\, l \in Path(N_j, d)}} o_{c,j} \times B(c) + \sum_{c: l \in Path(d, c.out)} \sum_{N_j \in c.path} o_{c,j} \times B(c) \times (1 - Match(c)) - \sum_{c} \sum_{N_j \prec_c l} o_{c,j} \times B(c) - \sum_{c} \sum_{N_j \prec_c l} p_{c,j} \times B(c) \times Match(c) \quad (10)$$

Fig. 7. Formulation for balancing the scaling, latency, and footprint of unwanted traffic in network-wide NIPS offloading

the ingress I and egress E. This toy topology has a single data center D and traffic being offloaded to D from a given node N.

Goals: As discussed earlier, NIPS offloading introduces several new dimensions: (1) ensure that the NIPS hardware is not overloaded; (2) keep all the links at reasonable loads to avoid unnecessary network congestion; (3) add minimal amount of extra latency for user applications; and (4) minimize the network footprint of unwanted traffic. Of these, we model (2) as a *constraint* and model the remaining factors as a multi-criterion objective.[2]

Note that these objectives could possibly be in conflict and thus we need to systematically model the trade-offs between these objectives. For instance, if are not worried about the latency impact, then the optimal solution is to always offload traffic to the datacenter. To this end, we model our objective function as a weighted combination of factors (1), (3), and (4). Our goal here is to devise a general framework rather than

[2] The choice of modeling some requirement as a strict constraint vs. objective may differ across deployments; as such, our framework is quite flexible. We use strict bounds on the link loads to avoid congestion.

mandate specific values of the weights. We discuss some natural guidelines for selecting these weights in §7.

Coverage (Eqn. 1): Given the process and offload variables, we need to ensure that every session in each class is processed somewhere in the network. Eqn. 1 captures this coverage requirement and ensures that for each class c the traffic is analyzed by some NIPS on that path or offloaded to the datacenter. In our example, this means that $p_{\mathsf{c,I}}$, $p_{\mathsf{c,N}}$, $p_{\mathsf{c,E}}$, and $o_{\mathsf{c,N}}$ should sum up to 1.

Resource Load (Eqn. 2–Eqn. 5): Recall that F_c^r is the per-session processing cost of running the NIPS analysis for traffic on class c. Given these values, we model the load on a node as the product of the processing fraction $p_{c,j}$, the traffic volume along these classes and the resource footprint F_c^r. That is, the load on node N_j due to traffic processed on c is $S(c) \times p_{c,j} \times F_c^r$. Since our goal is to have all nodes operating within their capacity, we add the constraint in Eqn. 3 to ensure that no node exceeds the provisioned capacity. The load on the datacenter depends on the total traffic offloaded to it, which is determined by the $o_{c,j}$ values, i.e., $o_{\mathsf{c,N}}$ in our example of Figure 6. Again, this must be less than the capacity of the datacenter, as shown in Eqn. 4. Furthermore, since we want to minimize resource load, Eqn. 5 captures the maximum resource consumption across all nodes (except the datacenter).[3]

Latency Penalty due to Rerouting (Eqn. 8): Offloading means that traffic takes a detour from its normal path to the datacenter (and then to the egress). Thus, we need to compute the latency penalty caused by such rerouting. For any given node N_j, the original path $c.path$ can be treated as the logical concatenation of the path $Path(in, N_j)$ from ingress in to node N_j and the path $Path(N_j, out)$ from N_j to the egress out. When we offload to the datacenter, the additional cost is the latency from this node to the datacenter and datacenter to the egress. However, since this traffic does not traverse the path from N_j to the egress, we can subtract out that latency. In Figure 6, the original latency is $|Path(\mathsf{I},\mathsf{N})| + |Path(\mathsf{N},\mathsf{E})|$; the offloaded traffic incurs a latency of $|Path(\mathsf{I},\mathsf{N})| + |Path(\mathsf{N},\mathsf{D})| + |Path(\mathsf{D},\mathsf{E})|$ which results in a latency increase of $|Path(\mathsf{N},\mathsf{D})| + |Path(\mathsf{D},\mathsf{E})| - |Path(\mathsf{N},\mathsf{E})|$. This models the latency increase for a given class; the accumulated latency across all traffic is simply the sum over all classes (Eqn. 8).

Unwanted Footprint (Eqn. 9): Ideally, we want to drop unwanted traffic as early as possible to avoid unnecessarily carrying such traffic. To capture this, we compute the total "network footprint" occupied by unwanted traffic. Recall that the amount of unwanted traffic on class c is $Match(c) \times B(c)$. If the traffic is processed locally at router N_j, then the network distance traversed by the unwanted traffic is simply $|Path(c.in, N_j)|$. If the traffic is offloaded to the datacenter by N_j, however, then the network footprint incurred will be $|Path(c.in, N_j)| + |Path(N_j, d)|$. Given a reasonable bucketing function, we can assume that unwanted traffic will get mapped uniformly across the different logical buckets corresponding to the process and offload

[3] At first glance, it may appear that this processing load model does not account for reduction in processing load due to traffic being dropped upstream. Recall, however, that $p_{c,j}$ and $o_{c,j}$ are defined as fractions of original traffic that enters the network. Thus, traffic dropped upstream will not impact the processing load model.

variables. In our example, the volume of unwanted traffic dropped at N is simply $Match(\mathsf{c}) \times B(\mathsf{c}) \times p_{\mathsf{c},\mathsf{N}}$. Given this, we can compute the network footprint of the unwanted traffic as a combination of the locally processed and offloaded fractions as shown in Eqn. 9.

Due to the processing coverage constraint, we can guarantee that SNIPS provides the same the security functionality as provided by a traditional ingress NIPS deployment. That is, any malicious traffic that should be dropped will be dropped *somewhere* under SNIPS. (And conversely, no legitimate traffic will be dropped.)

Link Load (Eqn. 6, Eqn. 7, Eqn. 10): Last, we come to the trickiest part of the formulation — modeling the link loads. To model the link load, we start by considering the baseline volume that a link will see if there were no traffic being dropped and if there were no offloading. This is the background traffic that is normally being routed. Starting with this baseline, we notice that NIPS offloading introduces both positive and negative components to link loads.

First, rerouting can induce additional load on a given link if it lies on a path between a router and the datacenter; either on the forward path to the datacenter or the return path from the data center to the egress. These are the additional positive contributions shown in Eqn. 10. In our example, any link that lies on the path $Path(\mathsf{N}, \mathsf{D})$ will see additional load proportional to the offload value $o_{\mathsf{c},\mathsf{N}}$. Similarly, any link on the path from the data center will see additional induced load proportional to $o_{\mathsf{c},\mathsf{N}} \times (1 - Match(\mathsf{c}))$ because some of the traffic will be dropped.

NIPS actions and offloading can also reduce the load on some links. In our example, the load on the link N-E is lower because some of the traffic has been offloaded from N; this is captured by the first negative term in Eqn. 10. There is also some traffic dropped by the NIPS processing at the upstream nodes. That is, the load on link N-E will be lowered by an amount proportional to $(p_{\mathsf{c},\mathsf{I}} + p_{\mathsf{c},\mathsf{N}}) \times Match(\mathsf{c})$. We capture this effect with the second negative term in Eqn. 10 where we use the notation $N_j \prec_c l$ to capture routers that are upstream of l along the path $c.path$.

Together, we have the link load on each link expressed as a combination of three factors: (1) baseline background load; (2) new positive contributions if the link lies on the path to/from the datacenter, and (3) negative contributions due to traffic dropped upstream and traffic being rerouted to the data center. Our constraint is to ensure that no link is overloaded beyond a certain fraction of its capacity; this is a typical traffic engineering goal to ensure that there is only a moderate level of congestion at any time.

Solution: Note that our objective function and all the constraints are *linear* functions of the decision variables. Thus, we can leverage commodity linear programming (LP) solvers such as `CPLEX` to efficiently solve this constrained optimization problem. In §6 we discuss how we map the output of the optimization (fractional $p_{c,j}$ and $o_{c,j}$ assignments) into *data plane* configurations to load balance and offload the traffic.

We note that this basic formulation can be extended in many ways. For instance, administrators may want different types of guarantees on NIPS failures: fail-open (i.e., allow some bad traffic), fail-safe (i.e., no false negatives but allow some benign traffic to be dropped), or strictly correct. SNIPS can be extended to support such policies; e.g., modeling redundant NIPS or setting up forwarding rules to allow traffic to pass through.

6 Implementation Using SDN

In this section, we describe how we implement SNIPS using software-defined networking (SDN). At a high-level, an SDN architecture consists of a network controller and SDN-enabled switches [3]. The controller installs rules on the switches using an open API such as OpenFlow [18] to specify forwarding actions for different flow match patterns. The flow match patterns are exact or wildcard expressions over packet header fields. This ability to programmatically set up forwarding actions enables a *network-layer* solution for NIPS offloading that does not require NIPS modifications and can thus work with legacy/proprietary NIPS hardware.

SNIPS Using SDN/OpenFlow: We want to set up forwarding rules to steer traffic to the different NIPSes. That is, given the $p_{c,j}$ and $o_{c,j}$ values, we need to ensure that each NIPS receives the designated amount of traffic. In order to decouple the formulation from the implementation, our goal is to translate *any* configuration into a correct set of forwarding rules.

As discussed in §4, each traffic class c is identified by application-level ports and network ingress/egress. Enterprise networks typically use structured address assignments; e.g., each site may be given a dedicated IP subnet. Thus, in our prototype we identify the class using the IP addresses (and TCP/UDP port numbers). Note that we do not constrain the addressing structure; the only requirement is that hosts at different locations are assigned addresses from non-overlapping IP prefix ranges and that these assignments are known.

For clarity, we assume that each NIPS is connected to a single SDN-enabled switch. In the context of our formulation, each abstract node N_j can be viewed as consisting of a SDN switch S_j connected to the NIPS $NIPS_j$.[4]

6.1 Challenges in Using SDN

While SDN is indeed an enabler, there are three practical challenges that arise in our context. We do not claim that these are fundamental limitations of SDN. Rather, SNIPS induces new requirements outside the scope of traditional SDN/OpenFlow applications [3] and prior SDN use cases [23, 24].

Stateful Processing: NIPS are *stateful* and must observe both forward and reverse flows of a TCP/UDP session for correct operation. In order to pin a session to a specific node, prior solutions for NIDS load balancing use bidirectional hash functions [9, 34]. However, such capabilities do not exist in OpenFlow and we need to explicitly ensure stateful semantics.

To see why this is a problem, consider the example in Figure 8 with class c1 (c1.in=S1 and c1.out = S2) with $p_{\mathsf{c1},\mathsf{NIPS1}}$=$p_{\mathsf{c1},\mathsf{NIPS2}}$=$0.5$. Suppose hosts with gateways S1 and S2 are assigned IP addresses from prefix ranges $Prefix_1$=$10.1/16$ and $Prefix_2$=$10.2/16$ respectively. Then, we set up forwarding rules so that packets with $src = 10.1.0/17$, dst=$10.2/16$ are directed to NIPS NIPS1 and those with

[4] For "inline" NIPS deployments, the forwarding rules need to be on the switch immediately upstream of the NIPS and the NIPS needs to be configured to act in "bypass" mode to allow the remaining traffic to pass through untouched.

src=10.1.128/17, *dst*=10.2/16 are directed to NIPS2 as shown in the top half of Figure 8. Thus, the volume of traffic each NIPS processed matches the SNIPS optimization. Note that we need two rules, one for each direction of traffic. [5]

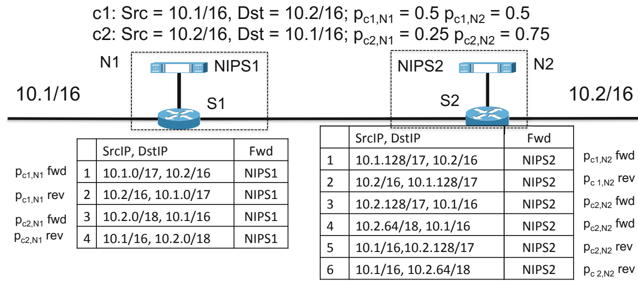

Fig. 8. Potentially conflicting rules with bidirectional forwarding rules for stateful processing. The solution in this case is to logically merge these conflicting classes.

There is, however, a subtle problem. Consider a different class c2 whose c2.in = S2 and c2.out = S1. Suppose $p_{\mathsf{c2},\mathsf{NIPS1}} = 0.25$ and $p_{\mathsf{c2},\mathsf{NIPS2}} = 0.75$. Without loss of generality, let the split be $src = 10.2.0/18$, $dst = 10.1/16$ for NIPS1 and rest to NIPS2 as shown in bottom half of Figure 8. Unfortunately, these new rules will create conflict. Consider a bidirectional session $src = 10.1.0.1$, $dst = 10.2.0.1$. This session will match two sets of rules; e.g., the forward flow of this session matches rule 1 on S1 while the reverse flow matches rule 4 (a reverse rule for c2) on S2. Such ambiguity could violate the stateful processing requirement if the forward and reverse directions of a session are directed to different NIPS.

Skewed Volume Distribution: While class merging ensures stateful processing, using prefix-based partitions may not ensure that the load on the NIPS matches the optimization result. To see why, consider Figure 9 with a single class and two NIPS, NIPS1 and NIPS2, with an equal split. The straw man solution steers traffic between 10.1.0/17–10.2/16 to NIPS1 and the remaining (10.1.128/17–10.2/16) to NIPS2. While this splits the prefix space equally, the actual load may be skewed if the volume is distributed as shown. The actual load on the NIPS nodes will be 0.3 and 0.7 instead of the intended 0.5:0.5. This non-uniform distribution could happen for several reasons; e.g., hotspots of activity or unassigned regions of the address space.

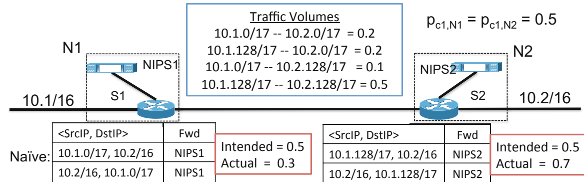

Fig. 9. NIPS loads could be violated with a non-uniform distribution of traffic across different prefix subranges. The solution in this case is a weighted volume-aware split

Potential Routing Loops: Finally, there is a corner case if the same switch is on the path to/from the data center. Consider the route: $\langle in, \ldots, S_{\mathit{offload}}, \ldots, S_i, S_j, \ldots, S_d, d, S_d, \ldots, S_i, S_j, \ldots, out\rangle$. With flow-based forwarding rules, S_j cannot decide if a

[5] For clarity, the example only shows forwarding rules relevant to NIPS; there are other basic routing rules that are not shown.

packet needs to be sent toward the datacenter d or toward egress out. (Note that this is not a problem for S_d itself; it can use the input interface on which the packet arrived to determine the forwarding action.)

We could potentially address some of these issues by modifying the optimization (e.g., choose a loop-free offload point for (2) or rewrite the optimization w.r.t merged classes for (1).) Our goal is to decouple the formulation from the implementation path. That is, we want to provide a correct SDN-based realization of SNIPS without making assumptions about the structure of the optimization solution or routing strategies.

6.2 Our Approach

Next, we discuss our approaches to address the above challenges. At a high-level, our solution builds on and extends concurrent ideas in the SDN literature [11,23,24]. However, to the best of our understanding, these current solutions do not handle conflicts due to stateful processing or issues of load imbalance across prefixes.

Class Merging for Stateful Processing: Fortunately, there is a simple yet effective solution to avoid such ambiguity. We identify such conflicting classes—i.e., classes c_1 and c_2 with $c_1.in = c_2.out$ and vice versa[6]—and logically *merge* them. We create a merged class c' whose $p_{c',j}$ and $o_{c',j}$ are (weighted) combinations of the original responsibilities so that the load on each NIPS $NIPS_j$ matches the intended loads. Specifically, if the resource footprints $F^r_{c_1}$ and $F^r_{c_2}$ are the same for each resource r, then it suffices to set $p_{c',j} = \frac{S(c_1) \times p_{c_1,j} + S(c_2) \times p_{c_2,j}}{S(c_1) + S(c_2)}$. In Figure 8, if the volumes for c1 and c2 are equal, the effective fractions are $p_{\mathsf{c}',\mathsf{NIPS1}} = \frac{0.5+0.25}{2}$ and $p_{\mathsf{c}',\mathsf{NIPS2}} = \frac{0.5+0.75}{2}$. We can similarly compute the effective offload values as well. If the resource footprints $F^r_{c_1}$ and $F^r_{c_2}$ are not the same for each resource r, however, then an appropriate combination can be computed using an LP (not shown for brevity).

Volume-Aware Partitioning: A natural solution to this problem is to account for the volumes contributed by different prefix ranges. While this problem is theoretically hard (being reducible to knapsack-style problems), we use a simple heuristic described below that performs well in practice, and is quite efficient.

Let $PrefixPair_c$ denote the IP subnet pairs for the (merged) class c. That is, if $c.in$ is the set $Prefix_{in}$ and $c.out$ is the set $Prefix_{out}$, then $PrefixPair_c$ is the cross product of $Prefix_{in}$ and $Prefix_{out}$. We partition $PrefixPair_c$ into non-overlapping blocks $PrefAtom_{c,1} \dots PrefAtom_{c,n}$. For instance, if each block is a $/24 \times /24$ subnet and the original $PrefixPair$ is a $/16 \times /16$, then the number of blocks is $n = \frac{2^{16} \times 2^{16}}{2^8 \times 2^8} = 65536$. Let $S(k)$ be the volume of traffic in the kth block.[7] Then, the fractional weight for each block is $w_k = \frac{S(k)}{\sum_{k'} S(k')}$.

We discretize the weights so that each block has weight either δ or zero, for some suitable $0 < \delta < 1$. For any given δ, we choose a suitable partitioning granularity so that

[6] If the classes correspond to different well-known application ports, then we can use the port fields to disambiguate the classes. In the worst case, they may share some sets of application ports and so we could have sessions whose port numbers overlap.

[7] These can be generated from flow monitoring reports or statistics exported by the OpenFlow switches themselves.

the error due to this discretization is minimal. Next, given the $p_{c,j}$ and $o_{c,j}$ assignments, we run a pre-processing step where we also "round" each fractional value to be an integral multiple of δ.

Given these rounded fractions, we start from the first assignment variable (some $p_{c,j}$ or $o_{c,j}$) and block $PrefAtom_{c,1}$. We *assign* the current block to the current fractional variable until the variable's demand is satisfied; i.e., if the current variable, say $p_{c,j}$, has the value 2δ, then it is assigned two non-zero blocks. The only requirement for this procedure to be correct is that each variable value is satisfied by an integral number of blocks; this is true because each weight is 0 or δ and each variable value is an integral multiple of δ. With this assignment, the volume of traffic meets the intended $p_{c,j}$ and $o_{c,j}$ values (modulo rounding errors).

Handling Loops Using Packet Tagging: To handle loops, we use *packet tags* similar to prior work [11,23]. Intuitively, we need the switches on the path from the datacenter to the egress to be able determine that a packet has already been forwarded. Because switches are stateless, we add tags so that the packet itself carries the relevant "state" information. To this end, we add an OpenFlow rule at S_d to set a *tag bit* to packets that are entering *from* the datacenter. Downstream switches on the path to out use this bit (in conjunction with other packet header fields) to determine the correct forwarding action. In the above path, S_j will forward packets with tag bit 0 toward d and packets with bit 1 toward out.

6.3 Putting it Together

Given these building blocks we translate the LP solution into an SDN configuration in three steps:

1. Identify conflicting classes and merge them.
2. Use a weighted scheme to partition the prefix space for each (merged) class so that the volume matches the load intended by the optimization solution.
3. Check for possible routing loops in offloaded paths and add corresponding tag addition rules on the switches.

We implement these as custom modules in the POX SDN controller [22]. We choose POX mostly due to our familiarity; these extensions can be easily ported to other platforms. One additional concern is how packets are handled during SNIPS rule updates to ensure stateful processing. To address this we can borrow known techniques from the SDN literature [25].

7 Evaluation

In evaluating SNIPS, we focus on two key aspects:

- System benchmarks using our SDN implementation (§7.1).
- Performance benefits over other NIPS architectures (§7.2).

Setup: We use a combination of custom trace-driven simulations, a real system emulation using Mininet [16], and optimization-driven analysis. We use OpenvSwitch as the SDN switch and use Snort as the NIPS.

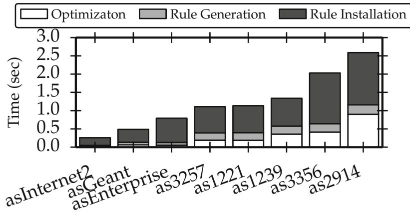

Fig. 10. Overhead of SNIPS: Time to run the optimization, and generate/install forwarding rules

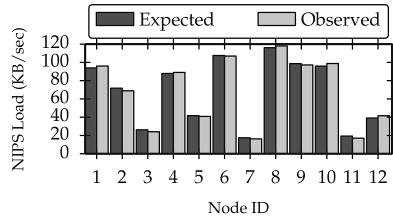

Fig. 11. Validating that our SDN implementation faithfully realizes the SNIPS optimization on the Internet2 topology

We use realistic network topologies from educational backbones, ISPs [33], and an enterprise network; these topologies range in size between 11 and 70 nodes. Due to the absence of public traffic data, we use a *gravity model* based on location populations [26] to generate the traffic matrix specifying the volume of traffic between every pair of network nodes for the AS-level topologies. For the enterprise topology, we obtained the enterprise's empirical traffic matrix. For simplicity, we consider only one application-level class and assume there is a single datacenter located at the node that observes the largest volume of traffic.

We configure the node and link capacities as follows. We assume a baseline *ingress* deployment (without offloading or on-path distribution) where all NIPS processing occurs at the ingress of each end-to-end path. Then, we compute the maximum load across all ingress NIPS and set the capacity of each NIPS to this value and the datacenter capacity to be $10\times$ this node capacity. For link capacities, we simulate the effect of routing traffic without any offloading or NIPS-induced packet drops, and compute the maximum volume observed on the link. Then, we configure the link capacities such that the maximum loaded link is at $\approx 35\%$ load.

7.1 System Benchmarks

Computation Overhead: A potential concern with centralized management is the time to recompute the network configurations, especially in reaction to network dynamics. The SNIPS system has three potential overheads: solving the linear program using CPLEX; translating the LP solution to OpenFlow rules; and rule dissemination. Figure 10 shows the breakdown of these three components for the different topologies. Even with the largest topology (AS2914) with 70 nodes, the total time for configuration is only 2.6 seconds. Given that typical network reconfiguration tasks need to be performed every few minutes, this overhead is quite low [5].

Validation: We validated that our SDN implementation faithfully matches the load distribution intended by the optimization. Figure 11 shows this validation result in terms of the normalized NIPS loads (measured in total volume of traffic) for the Internet2 topology. (We have verified this for other topologies but do not show it for brevity.) Nodes 1–11 are the local NIPS and Node 12 is the data center. We use the LP solution to generate the expected load using the original traffic matrix. The result shows that

the observed load closely tracks the intended load.[8] In this specific configuration, the volume of traffic offloaded to the datacenter (node 12) is small, but as we will see in the following sections, in other topologies the datacenter can help significantly.

7.2 Benefits of SNIPS

Next, we evaluate the performance benefits of SNIPS. We start with a baseline result with a simple configuration before evaluating the sensitivity to different parameters. For the baseline, we set the SNIPS parameters $\beta = \alpha = 0.333$; i.e., all three factors (latency, unwanted hops, load) are weighted equally in the optimization. We fix the fraction of unwanted traffic to be 10%. For all results, the maximum allowable link load is 40%.

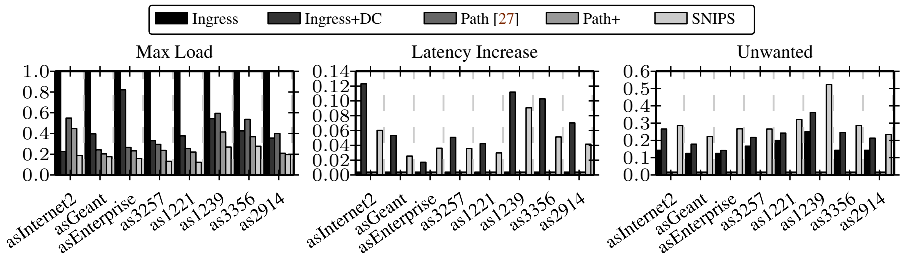

Fig. 12. Trade-offs between current deployments and SNIPS

Improvement Over Current NIPS Architectures: We compare the performance of SNIPS against today's *Ingress* NIPS deployments. As an intermediary point, we also consider three other deployments: 1) *Ingress+DC* deployment, where all processing/offloading happens at the ingress of each path and the datacenter. 2) *Path* deployment, modeling the on-path deployment described in [27]; and 3) *Path+*: identical to *Path* except each node has an increased capacity of $DCap^r/N$.

Figure 12 shows three normalized metrics for the topologies: load, added latency, and unwanted footprint. For ease of presentation, we normalize each metric by the maximum possible value for a specific topology so that it is between 0 and 1.[9] Higher values indicate less desirable configurations (e.g., higher load or latency).

By definition, the Ingress deployment introduces no additional latency and unwanted footprint is low[10], since all of the processing is at the edge of the network. Such a deployment, however, can suffer overload problems as shown in the result. SNIPS offers a more flexible trade-off: a small increase in latency and unwanted footprint for a significant reduction in the maximum compute load. We reiterate that SNIPS does not affect the security guarantees; it will drop *all* unwanted traffic, but it may choose to do so after a few extra hops. In some topologies (e.g., AS3356) SNIPS can reduce the maximum load by $10\times$ compared to a naive ingress deployment while only increasing the latency

[8] The small discrepancies are due to the variability in flow sizes.

[9] Hence the values could be different across topologies even for the ingress deployment.

[10] It is impossible for this footprint to be 0, since unwanted traffic enters the network and must be flagged as such.

by 2%. Similarly, SNIPS can provide a 4× reduction in load without increasing latency over the Ingress+DC deployment (e.g., AS3257). Note that these benefits arise with a very simple equi-weighted trade-off across the three objective components; the benefits could be even better with other configurations.

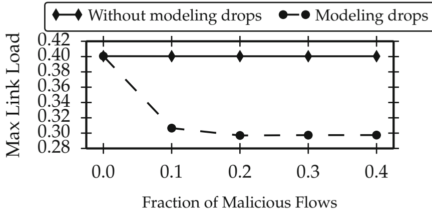

Fig. 13. Link load as a function of fraction of "unwanted" traffic

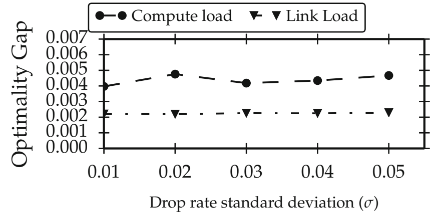

Fig. 14. Compute and link load optimality gap as functions of drop rate deviation; estimated drop rate = distribution mean $\mu = 0.1$

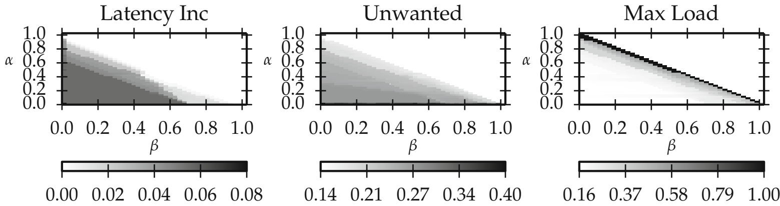

Fig. 15. Visualizing trade-offs in choosing different weight factors on Internet2 topology

Impact of Modeling Traffic Drops: SNIPS provides a higher fidelity model compared to past works in NIDS offloading because it explicitly incorporates the impact of traffic drops. We explore the impact of modeling these effects. For this result, we choose the Internet2 topology and use our simulator to vary the fraction of malicious flows in the network. Figure 13 shows the maximum observed link loads, averaged over 50 simulation runs. In addition to directly using the SNIPS-recommended strategy, we also consider a naive setup that does not account for such drops.

There are two key observations. First, the max link load is significantly lower with SNIPS which means that SNIPS can exploit more opportunities to offload under overload compared to the naive model. Second, by assuming no drops, "no drop" setup ignores the $HopsUnwanted$ factor, thus potentially obstructing the link to the datacenter with unwanted traffic that could have been dropped at an earlier point in the network (this effect is represented in Figure 13).

7.3 Sensitivity Analysis

Sensitivity to Weights: As an illustrative result, we show the result of varying the weighting factors for the Internet2 topology in Figure 15. (We show only one topology due to space limitations). In the figure, darker regions depict higher values, which are less desirable. Administrators can use such visualizations to customize the weights to

suit their network topology and traffic patterns and avoid undesirable regions. In particular, our equi-weighted configuration is a simple but reasonable choice (e.g., mostly low shades of gray in this graph).

Sensitivity to Estimation Errors: We also show that the parameter estimation (such as drop rate) for our framework need not be precise. For this, we choose to run a number of simulations with imperfect knowledge of the drop rate. In that case, the drop rate is sampled from a Gaussian distribution with mean of 0.1 (the estimated drop rate) and changing standard deviation σ. Figure 14 shows the relative gap for compute and link loads, between values predicted by the optimization with exact drop rate knowledge and the simulated values. This result shows that even with large noise levels the difference in load on links and nodes is insignificant.

8 Conclusions

Offloading has recently emerged as an appealing alternative to traditional approaches for scaling in-network processing. The goal of this paper is to bring the benefits of offloading to NIPS deployments. As we discussed, NIPS create new dimensions—active dropping, rerouting, and user-perceived latency—that fall outside the purvey of prior offloading systems that apply to passive monitoring solutions. To address these challenges, we presented the design and implementation of SNIPS. We presented a linear programming framework to model the new effects and trade-offs and addressed practical challenges in an SDN-based implementation. We showed that SNIPS offers greater scalability and flexibility with respect to current NIPS architectures; it imposes low overhead; and is robust to variations in operating parameters.

Acknowledgments. This work was supported in part by grant number N00014-13-1-0048 from the Office of Naval Research; NSF awards 1040626, 1330599, 1440056, and 1440065; and an NSF Graduate Research Fellowship.

References

1. Private communication with UNC administrators (2013)
2. Abraham, A., Jain, R., Thomas, J., Han, S.Y.: D-SCIDS: Distributed soft computing intrusion detection system. Journal of Network and Computer Applications 30 (2007)
3. Casado, M., et al.: Ethane: Taking control of the enterprise. ACM SIGCOMM (2007)
4. Dreger, H., Feldmann, A., Paxson, V., Sommer, R.: Predicting the resource consumption of network intrusion detection systems. In: Lippmann, R., Kirda, E., Trachtenberg, A. (eds.) RAID 2008. LNCS, vol. 5230, pp. 135–154. Springer, Heidelberg (2008)
5. Feldmann, A., et al.: Deriving traffic demands for operational IP networks: methodology and experience. In: Proc. SIGCOMM (2000)
6. Fortz, B., Rexford, J., Thorup, M.: Traffic engineering with traditional IP routing protocols. IEEE Communications Magazine 40 (2002)
7. Gibb, G., Zeng, H., McKeown, N.: Outsourcing network functionality. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (2012)
8. Google Research: No Mobile Site = Lost Customers, http://goo.gl/f8lBbR

9. Heorhiadi, V., Reiter, M.K., Sekar, V.: New opportunities for load balancing in network-wide intrusion detection systems. ACM CoNEXT (2012)
10. Jamshed, M.A., Lee, J., Moon, S., Yun, I., Kim, D., Lee, S., Yi, Y., Park, K.: Kargus: a highly-scalable software-based intrusion detection system. In: ACM CCS (2012)
11. Jin, X., Li, L.E., Vanbever, L., Rexford, J.: SoftCell: Scalable and Flexible Cellular Core Network Architecture. In: Proc. CoNext (2013)
12. Kohler, E., Morris, R., Chen, B., Jannotti, J., Kaashoek, M.F.: The Click modular router. TOCS 18, 263–297 (2000)
13. Kreibich, C., Sommer, R.: Policy-controlled event management for distributed intrusion detection. In: Distributed Computing Systems Workshops (2005)
14. Lee, J., et al.: A high performance NIDS using FPGA-based regular expression matching. In: ACM Symposium on Applied Computing (2007)
15. Meiners, C.R., et al.: Fast regular expression matching using small TCAMs for network intrusion detection and prevention systems. In: USENIX Security Symposium (2010)
16. Mininet, http://www.mininet.org
17. Network functions virtualisation – introductory white paper, http://portal.etsi.org/NFV/NFV_White_Paper.pdf
18. Openflow standard, http://www.openflow.org/
19. Papadogiannakis, A., Polychronakis, M., Markatos, E.P.: Tolerating Overload Attacks Against Packet Capturing Systems. In: USENIX Annual Technical Conference (2012)
20. Paxson, V.: Bro: a system for detecting network intruders in real-time. In: Proc. USENIX Security (1998)
21. Porras, P.A., Neumann, P.G.: EMERALD: Event monitoring enabling response to anomalous live disturbances. In: National Information Systems Security Conference (1997)
22. POX Controller, http://www.noxrepo.org/pox/about-pox/
23. Qazi, Z., Tu, C.-C., Chiang, L., Miao, R., Sekar, V., Yu, M.: Simple-fying middlebox policy enforcement using sdn. In: Proc. SIGCOMM (2013)
24. Wang, R., Butnariu, D., Rexford, J.: Openflow-based server load balancing gone wild. In: Proc. Hot-ICE (2011)
25. Reitblatt, M., Foster, N., Rexford, J., Schlesinger, C., Walker, D.: Abstractions for network update. In: ACM SIGCOMM (2012)
26. Roughan, M.: Simplifying the synthesis of internet traffic matrices. ACM CCR, 35 (2005)
27. Sekar, V., Krishnaswamy, R., Gupta, A., Reiter, M.K.: Network-wide deployment of intrusion detection and prevention systems. In: ACM CoNEXT (2010)
28. Sekar, V., Reiter, M.K., Willinger, W., Zhang, H., Kompella, R.R., Andersen, D.G.: CSAMP: a system for network-wide flow monitoring. In: Proc. NSDI (2008)
29. Sherry, J., et al.: Making middleboxes someone else's problem: Network processing as a cloud service. In: ACM SIGCOMM (2012)
30. Shin, S., Gu, G.: Attacking Software-Defined Networks: A First Feasibility Study. In: ACM SIGCOMM Workshop on Hot Topics in Software Defined Networking (2013)
31. Shin, S., Porras, P., Yegneswaran, V., Fong, M., Gu, G., Tyson, M.: FRESCO: Modular composable security services for software-defined networks. In: Proc. NDSS (2013)
32. Smith, R., Estan, C., Jha, S.: XFA: Faster signature matching with extended automata. In: IEEE Symposium on Security and Privacy (2008)
33. Spring, N., Mahajan, R., Wetherall, D.: Measuring ISP topologies with rocketfuel. In: ACM SIGCOMM (2002)
34. Vallentin, M., Sommer, R., Lee, J., Leres, C., Paxson, V., Tierney, B.: The NIDS cluster: Scalable, stateful network intrusion detection on commodity hardware. In: Kruegel, C., Lippmann, R., Clark, A. (eds.) RAID 2007. LNCS, vol. 4637, pp. 107–126. Springer, Heidelberg (2007)

35. Vasiliadis, G., Polychronakis, M., Antonatos, S., Markatos, E.P., Ioannidis, S.: Regular expression matching on graphics hardware for intrusion detection. In: Kirda, E., Jha, S., Balzarotti, D. (eds.) RAID 2009. LNCS, vol. 5758, pp. 265–283. Springer, Heidelberg (2009)
36. Vasiliadis, G., Polychronakis, M., Ioannidis, S.: MIDeA: a multi-parallel intrusion detection architecture. In: ACM CCS (2011)
37. Wang, R., Butnariu, D., Rexford, J.: Openflow-based server load balancing gone wild. In: Proc. Hot-ICE (2011)
38. World intrusion detection and prevention markets, http://goo.gl/j3QPX3
39. Yu, F., et al.: SSA: a power and memory efficient scheme to multi-match packet classification. In: ACM ANCS (2005)