Balancing Computation-Communication Tradeoffs in Scaling Network-Wide Intrusion Detection Systems

Victor Heorhiadi
University of North Carolina
Chapel Hill, NC, USA

Michael K. Reiter
University of North Carolina
Chapel Hill, NC, USA

Vyas Sekar
Intel Labs
Berkeley, CA, USA

Abstract

As traffic volumes and the types of analysis grow, network intrusion detection systems (NIDS) face a continuous scaling challenge. However, management realities limit NIDS upgrades typically to once every 3-5 years. Given that traffic patterns can change dramatically, this leaves a significant scaling challenge in the interim. This motivates the need for solutions that can help administrators better utilize their *existing* NIDS infrastructure. To this end, we design a general architecture for a network-wide NIDS deployment that leverages three scaling opportunities: *on-path* distribution to split responsibilities, *replicating* traffic to NIDS clusters, and *aggregating* intermediate results to split expensive NIDS processing. The challenge here is to balance both the compute load across the network and the total communication cost incurred via replication and aggregation. We implement a backwards-compatible mechanism to enable existing NIDS infrastructure to leverage these benefits. Using emulated and trace-driven evaluations on several real-world network topologies, we show that our proposal can substantially reduce the maximum computation load, provide better resilience under traffic variability, and offer improved detection coverage.

1 Introduction

Network intrusion detection systems play a critical role in keeping network infrastructures safe from attacks. The market for such appliances is estimated to be over one billion dollars [7, 15] and expected to grow substantially over the next few years [8]. The driving forces for increased deployment include increasing regulatory and policy requirements, new application traffic patterns (e.g., cloud, mobile devices), and the ever-increasing complexity of attacks themselves [7, 15, 8]. In conjunction with these forces, the rapid growth in traffic volumes means that NIDS deployments face a continuous scaling challenge to keep up with the increasing complexity of processing and volume of traffic.

The traditional response in the NIDS community to address this scaling challenge has been along three dimensions: efficient algorithms (e.g., [42, 41]); specialized hardware capabilities such as TCAMs (e.g., [53, 32]), FPGAs (e.g., [31, 33]), and graphics processors (e.g. [50, 49]); and parallelism through the use of multicore or cluster-based solutions (e.g., [48, 44, 29, 51]). These have been invaluable in advancing the state-of-the-art in NIDS system design. However, there is a significant delay before these advances are incorporated into production systems. Furthermore, budget constraints and management challenges mean that network administrators upgrade their NIDS infrastructure over a 3-5 year cycle [9]. Even though administrators try to provision the hardware to account for projected growth, disruptive and unforeseen patterns can increase traffic volumes: peer-to-peer, cloud computing, and consumer devices such as smartphones, to name a few. Consequently, it is critical to complement the existing research in building better NIDS systems with more immediately deployable solutions for scaling NIDS deployments.

In this context, recent work in the network management literature demonstrates the benefits of using a *network-wide* approach (e.g., [27, 19, 17, 39]). Specifically, past work has shown that distributing responsibilities across routers on an end-to-end path can offer significant benefits for monitoring applications [18, 39]. Sekar, et al. also showed the benefits of on-path distribution in the context of NIDS/NIPS systems [38]. This class of approaches is promising because it provides a way for administrators to handle higher traffic loads with their *existing* NIDS deployment without requiring a forklift upgrade to deploy new NIDS hardware.

Our premise is that these past proposals for distributing NIDS functions do not push the envelope far enough. Consequently, this not only restricts the scaling opportunities, but also constrains the detection capabilities that a network-wide deployment can provide:

- First, this prior work focuses distributing responsibilities strictly *on-path*. While such on-path processing is viable [21, 4], there is an equally compelling trend towards consolidating computing resources at some locations. This is evidenced by the popularity of "cloud" and datacenter deployments within and outside enterprise networks. Such deployments have natural management and multiplexing benefits that are ideally suited to the compute-intensive and dynamic nature of NIDS workloads.

- Second, this prior work assumes that the NIDS analysis occurring at a network node is *self-contained*. That is, the NIDS elements act as standalone entities and provide equivalent monitoring capabilities without needing to interact with other nodes. This restriction on self-contained analysis means that certain types of aggregated analysis are either infeasible or topologically constrained. For example, in the case of scan detection, Sekar, et al. constrain all traffic to be processed at the ingress gateway for each host [38].

Our vision is a *general* NIDS architecture that can not only exploit *on-path* distribution, but also allows broader scaling opportunities via traffic *replication* and analysis *aggregation*. In doing so, our work generalizes prior work and allows administrators to consider these proposals as specific points in this broader design space. By incorporating *replication*, we avoid the need for strictly on-path offloading, providing the freedom to offload processing to lightly loaded nodes that might be off-path. This also naturally accommodates technology trends toward building consolidated compute clusters. Furthermore, replication enables new detection functionality that would have been previously impossible. For example, our framework enables stateful NIDS analysis even when the two flows in a session (or two sessions as part of a stepping stone attack) do not traverse a common node. At the same time, by allowing *aggregation*, an expensive NIDS task can be split into smaller subtasks that can be combined to provide equivalent analysis capabilities. This enables more fine-grained scaling opportunities for NIDS analyses that would otherwise be topologically constrained (e.g., scan detection).

A key constraint here is to ensure that these new opportunities do not impose significant communication costs on the network. Thus, we need to assign processing responsibilities to balance the tradeoff between the communication cost imposed by replication and aggregation vs. the reduction in computation load. To systematically capture these tradeoffs, we design formal linear programming (LP) based optimization frameworks. We envision a network-wide management module that assigns processing, aggregation, and replication responsibilities across the network using these formulations. In order to execute these management decisions without requiring modifications to existing NIDS implementations, we interpose a lightweight *shim* layer that runs on each hardware platform.[1]

We evaluate our architecture and implementation using a combination of "live" emulation on Emulab [52] and trace-driven simulations on a range of real-world topologies. Our results show that a replication-enabled NIDS architecture can reduce the maximum computation load by up to 10×; is significantly more robust to variability in traffic patterns by reducing the peak load more than 20×; and can lower the detection miss rate from 90% to zero in some scenarios where routes may not be symmetric. These benefits are achieved with little overhead: computing the analysis and replication responsibilities takes $<$ 1.6 seconds with off-the-shelf LP solvers, and our shim layer imposes very low overhead.

Contributions and Roadmap: To summarize, the key contributions of this paper are:

- Identifying previously unexploited replication and aggregation opportunities for NIDS scaling (§2).
- Formal models for a general NIDS framework that subsumes existing on-path models and systematically balances compute-communication tradeoffs for replication and aggregation (§4, §5, §6).
- A backwards-compatible architecture (§3) and implementation (§7) to allow existing NIDS to benefit from these opportunities.
- Extensive evaluation of the potential benefits over a range of real-world network topologies (§8).

We discuss outstanding issues in §9 and related work in §10, before concluding in §11.

2 Background and Motivation

In this section, we begin by describing the prior work for *on-path* distribution [38]. Then, we discuss three motivating scenarios that argue for a general NIDS architecture that can incorporate traffic *replication* and analysis *aggregation*.

2.1 Proposed on-path distribution

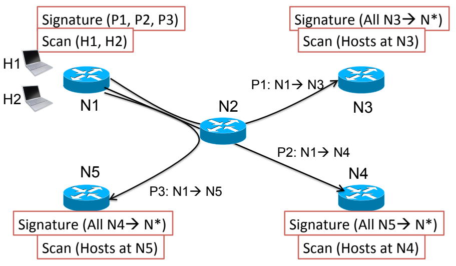


Figure 1: *NIDS deployments today are single-vantage-point solutions where the ingress gateway is responsible for monitoring all traffic*

[1]This shim functionality can also run at an upstream router to which the NIDS is attached.

Suppose there are two types of NIDS analysis: `Signature` for detecting malicious payloads and `Scan` for flagging hosts that contact many destination addresses. Figure 1 shows how today's NIDS deployments operate, wherein all types of analysis occur only at the gateway node. That is, node N1 runs `Scan` and `Signature` detection for all traffic to/from hosts H1-H2 on paths P1–P3 and other nodes run the analysis on hosts for which they are the gateway nodes. A natural limitation with this architecture is that if the load exceeds the provisioned capacity on a node, then that node has to either drop some functionality (e.g., disable expensive modules) or drop packets.

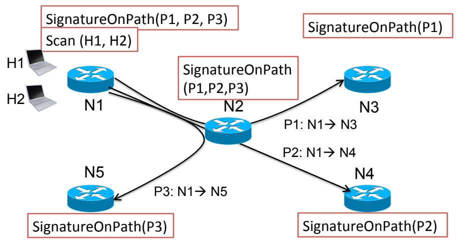


Figure 2: *NIDS with on-path distribution [38]: Any node on the path can run* `Signature` *detection;* `Scan` *detection cannot be distributed.*

One way to extend this is to leverage spare resources elsewhere in the network. For example, even though N1 may be overloaded, nodes N2–N5 may have some spare compute capacity. Building on this insight, Sekar, et al. describe an architecture in which any node on the end-to-end path can run the NIDS analysis if it can perform the analysis in a self-contained fashion without needing any post-processing [38]. For example, NIDS analysis such as `Signature` detection occur at a per-session granularity. Thus, the signature detection responsibilities can be split across the nodes on each end-to-end path by assigning each session to some node on that path, as shown in Figure 2. In the above example, the `Signature` analysis on path P1 is split between N1, N2, and N3; on path P2 between N1, N2, and N4; and between N1, N2, and N5 on P3. This can reduce the load on node N1 by leveraging spare compute resources on N2–N5. Note, however, that the `Scan` module cannot be distributed. `Scan` detection involves counting the number of unique destinations a source contacts which requires a complete view of all traffic to/from a given host. Thus, the ingress node alone is capable of running the analysis in a self-contained manner.

2.2 New Opportunities

Relaxing the on-path requirement: Now, the traffic on P1 might overload all nodes N1–N3 on the path. In this case, it is necessary to look for spare resources that are *off-path*. For example, nodes could locally offload some analysis to one-hop neighbors. Additionally, administrators may want to exploit compute clusters elsewhere in the network rather than upgrade every NIDS device. Such consolidated clusters or datacenters are appealing because they amortize deployment and management costs.

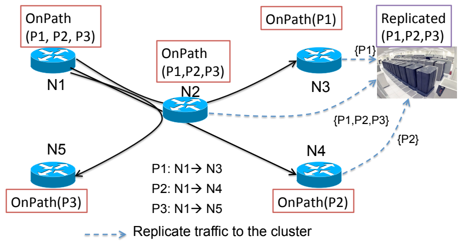


Figure 3: *Redirecting traffic to a compute cluster. With just on-path, the cluster at N3 cannot be used to handle traffic on P2 and P3.*

Consider the scenario in Figure 3. The network has a compute cluster located at node N3. When the processing load on the paths P2 and P3 exceed the provisioned capacity of their on-path nodes, we can potentially replicate (or reroute) traffic from node N2 to node N3 and perform the required analysis using the cluster. This assumes that: (1) there is sufficient network bandwidth to replicate this traffic and (2) the logic to do such replication has low overhead. For (1), we note that the primary bottleneck for many NIDS deployments is typically the number of active connections and the complexity of analysis, and not volume (in bytes) of traffic [22]. As we will show in §7, we can implement a lightweight shim layer to implement (2).

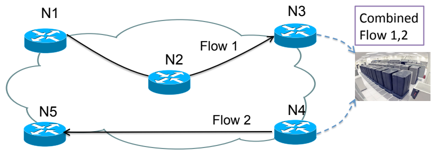


Figure 4: *The analysis needs to combine Flow 1 and Flow 2 (e.g., two directions of a session or two connections in a stepping stone), but they traverse non-intersecting paths. In this case, replication is inherently necessary to avoid detection misses*

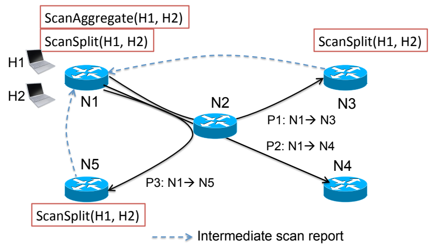


Figure 5: *Aggregating intermediate results lets us distribute analyses that might be topologically constrained.*

Network-wide views: Certain kinds of scenarios and analysis may need to combine traffic from different nodes. For example, "hot-potato" effects may cause the forward and reverse flows for an end-to-end session to traverse non-intersecting routing paths [46]. Thus, stateful NIDS analysis that needs to observe both sides of a session is impossible. A similar scenario occurs for stepping stone detection [54], if the two stages in the stepping stone do not encounter a common NIDS node. In Figure 4, traffic flows Flow 1 and Flow 2 need to be combined, but no single node can observe both flows. Thus, we need to replicate this traffic to a common location to analyze this traffic. Similarly, certain types of anomaly detection [16, 30] inherently require a network-wide view that no single node can provide.

Aggregation for fine-grained splitting: As we saw earlier, prior work requires each type of NIDS analysis to be self-contained. Consequently, analysis such as `Scan` detection are topologically constrained. Allowing the NIDS to communicate intermediate results provides further opportunities for distributing the load. Consider the setup in Figure 5. Each node on the path runs a subset of the `Scan` analysis. The nodes send their intermediate results—a table mapping a src IP to the set/number of destinations the node observed it contact—to an aggregation node that eventually generates alerts. (In this example, the aggregation happens at the ingress, but that is not strictly necessary.) Of course, we need to ensure that the result generated after aggregation is semantically equivalent. We defer to §6 on how we achieve this in practice.

The above scenarios highlight the need to look beyond pure on-path opportunities for distributing NIDS responsibilities in a network. In the next section, we begin with a high-level system overview before delving into the specific formulations for incorporating replication and aggregation opportunities in subsequent sections.

3 System Overview

Our goal is to optimally assign processing, aggregation, and replication responsibilities across the network. Optimality here involves a tradeoff between the compute load on the NIDS elements and the communication costs incurred. Next, we give an overview of the key entities and parameters involved in our framework (Figure 6).

We assume a logically centralized management module that configures the NIDS elements [27, 19, 17, 39]. This module periodically collects information about the current traffic patterns and routing policies. Such data feeds are routinely collected for other network management tasks [24]. Based on these inputs, the module runs the optimization procedures presented in the following sections to assign NIDS responsibilities. This optimization may be run periodically (say every 5 minutes) or

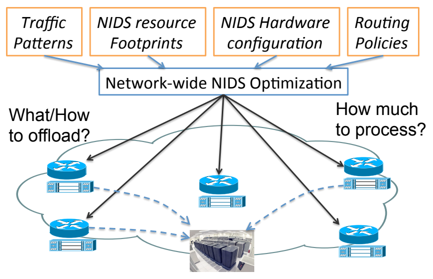


Figure 6: *Network-wide framework for assigning NIDS processing and replicating responsibilities*

triggered by routing or traffic changes (e.g., a specific route's volume increases). This periodic or triggered recomputation allows the system to dynamically adapt to changing traffic patterns. Note that the network administrators need not be bothered with the mathematical formulations. They need to specify high-level policy objectives (e.g., how much link capacity to allow for replication) and set up the optimization module to receive the relevant data feeds. Given these inputs, the configuration is completely automated and needs little or no manual intervention.

As Figure 6 shows, there are four inputs:

1. *Traffic patterns:* This categorizes the traffic along two axes: (1) the type or *class* of traffic (e.g., HTTP, IRC) and (2) the end-to-end *ingress-egress* pair to which the traffic belongs. (Ingress-egress pairs are a natural unit of traffic segmentation for many network management tasks [30].) Let $\mathscr{T}_i$ denote the set of end-to-end sessions of the traffic class i. Let k denote a specific ingress-egress pair and $\mathscr{T}_{ik}$ denote the sessions of class i whose ingress-egress combination is k. $|\mathscr{T}_{ik}|$ denotes the volume of traffic in terms of the number of sessions.

2. *Resource footprints:* Each class i may be subjected to different types of NIDS analyses. For example, HTTP sessions may be analyzed by a basic payload signature engine and through application-specific rules, while all traffic (itself a class) might be subjected to `Scan` analysis. We model the cost of running the NIDS for each class on a specific *resource* r (e.g., CPU cycles, resident memory) in terms of the expected per-session resource footprint Req_i^r (in units suitable for that resource). For a given class i we consider the aggregate effect of running all the NIDS modules relevant to it. We expect these values to be relatively stable over the timescales of few days and can be obtained either via NIDS vendors' datasheets or estimated using offline benchmarks [22]. Our optimization can provide significant benefits even with an approximate estimate of the Req_i^r values.

3. *NIDS hardware:* Each NIDS hardware device R_j is

characterized by its resource capacity Cap_j^r in units suitable for that resource. In the general case, we assume that hardware capabilities may be different across the network, e.g., because of upgraded hardware running alongside legacy equipment.

4. *Routing:* We start by assuming that each class and ingress-egress combination has a unique symmetric routing path P_{ik}, and then subsequently relax this assumption. We use the notation $R_j \in P_{ik}$ to denote that this NIDS node is *on the routing path* for P_{ik}. Note that some nodes (e.g., a dedicated cluster) could be completely off-path; i.e., it does not observe traffic on any end-to-end routing path unless some other node explicitly forwards traffic to it.

Communication Costs: We model communication costs in two ways. First, in the case of replication, we ensure that the additional *link load* imposed by the inter-NIDS communication is bounded. (This ensures that we do not overload network links and thus avoid packet losses.) Similar to the notion of a router being on the path, we use the notation $Link_l \in P_{ik}$ to denote that the network link l is on the path P_{ik}. Second, for aggregation, we count the total *network footprint* imposed by the inter-NIDS communication, measured in *byte-hops*. For example, if NIDS R_1 needs to send a 10KB report to NIDS R_2 four hops away, then the total footprint is $10 \times 4 = 40$ *KB-hops*.

Given this setup, we describe the formal optimization frameworks in the following sections.

4 NIDS with replication

As we saw in Figure 3, we can reduce the NIDS load by replicating the traffic to nodes that are off-path if they have spare resources. In this section, we provide a general framework for combining on-path distribution with off-path replication. For the current discussion, we assume that the NIDS analyses run at a *session-level* granularity. This is typical of most common types of NIDS analyses in use today [1, 34]. We also assume that each ingress-egress pair has a single symmetric routing path. Figure 7 shows the LP formulation for our framework, to which we refer throughout this section.

For each NIDS node, R_j, we introduce the notion of a *mirror set* $MirrorSet_j \subseteq \{R_1 \ldots R_N\}$ that represents a candidate set of nodes to which R_j can offload some processing. This allows us to flexibly capture different replication strategies. For example, in the most general case all nodes can be candidates for replication, i.e., $\forall j : MirrorSet_j = \{R_1 \ldots R_N\} \setminus \{R_j\}$. In case of a single datacenter or a specialized NIDS cluster, we can simply set $\forall j : MirrorSet_j = \{R_d\}$ where R_d is that datacenter/cluster. Alternatively, we can also consider local offload policies and set $MirrorSet_j$ to be R_j's one- or two-hop neighbors. Let $Path_{jj'}$ denote the routing path between R_j and the mirror node $R_{j'}$.

$$\text{Minimize } \max_{r,j}\{ResLoad_j^r\} \text{ subject to}$$

$$\forall i,k: \sum_{j:R_j \in P_{ik}} \left(d_{ikj} + \sum_{\substack{j':R_{j'} \in MirrorSet_j \\ R_{j'} \notin P_{ik}}} c_{ikjj'} \right) = 1 \quad (1)$$

$$\forall r,j: ResLoad_j^r = \frac{1}{Cap_j^r} \sum_{\substack{i,k: \\ R_j \in P_{ik}}} Req_i^r \times |\mathscr{T}_{ik}| \times d_{ikj} + \frac{1}{Cap_j^r} \sum_{\substack{j',i,k: \\ R_j \in MirrorSet_{j'} \\ R_j \notin P_{ik}}} Req_i^r \times |\mathscr{T}_{ik}| \times c_{ikj'j} \quad (2)$$

$$\forall l: LinkLoad_l = Background_l + \sum_{\substack{i,k,j,j': \\ Link_l \in Path_{jj'} \\ R_{j'} \in MirrorSet_j}} \frac{|\mathscr{T}_{ik}| \times c_{ikjj'} \times Size_i}{LinkCap_l} \quad (3)$$

$$\forall l: LinkLoad_l \leq \max\{MaxLinkLoad, Background_l\} \quad (4)$$

$$\forall i,k,j: 0 \leq d_{ikj} \leq 1 \quad (5)$$

$$\forall i,k,j,j': 0 \leq c_{ikjj'} \leq 1 \quad (6)$$

Figure 7: *LP formulation for replication*

At a high-level, we need to decide if a given NIDS node is going to *process* a given session or *replicate* that traffic to one of its candidate mirror nodes (or neither). We capture these determinations with two control variables. First, d_{ikj} specifies the fraction of traffic on the path k of class i that the node R_j processes locally. To capture offloading via replication, we have an additional control variable: $c_{ikjj'}$ which represents the fraction of traffic belonging to P_{ik} that R_j offloads to its "mirror" node $R_{j'}$. Note that there is no need to replicate traffic to elements that are already on-path; formally, if $R_{j'} \in P_{ik}$ then the variables $c_{ikjj'}$ will not appear in the formulation. The bounds on the variables in Eq (5) and (6) ensure that these can only take fractional values between zero and one.

Recall that our objective is to assign processing and offloading responsibilities across the network to balance the tradeoff between the *computation load* and the *communication cost*. Here, we focus on the communication cost as a given constraint on the maximum allowed link load *MaxLinkLoad* imposed by the replicated traffic. For example, network administrators typically want to keep links at around 30–50% utilization in order to absorb sudden bursts of traffic [25].

Our main constraint is a *coverage requirement*; we want to ensure that for each class of traffic and for each ingress-egress pair, the traffic is processed by some node either on- or off-path. Eq (1) captures this constraint by considering the sum of the locally processed fractions d_{ikj} and the offloaded fractions $c_{ikjj'}$ and setting it to 1 for full coverage.

Eq (2) captures the stress on each resource for each node. There are two sources of load on each node: the traffic it needs to process locally from on-path responsibilities (i.e., via d_{ikj}s) and the total traffic it processes as a consequence of other nodes offloading traffic (i.e., via $c_{ikj'j}$s) to it. The inversion in the indices for the c contribution is because the load on R_j is a function of what other $R_{j'}$s offload to it.

Then, Eq (3) models the link load on the link l imposed by the traffic between every pair of R_j and its mirror nodes. Because $|\mathcal{T}_{ik}|$ only captures the number of sessions, we introduce an extra multiplicative factor $Size_i$ to capture the average size (in bytes) of each session of class i. We also have an additive term $Background_l$ to capture the current load on the link due to the normal traffic traversing it (i.e., not induced by our replication). These additive terms can be directly computed given the traffic patterns and routing policy, and as such we treat at as a constant input in the formulation.

As discussed earlier, we bound the communication cost in terms of the maximum link load in Eq (4). The max is needed because the background load may itself exceed the given constraint *MaxLinkLoad*; in this case, the only thing we can ensure that no new traffic is induced on such overloaded links.

Given these constraints, we focus on a specific load balancing objective to minimize the maximum load across all node-resource pairs. We use standard LP solvers to obtain the optimal d_{ikj} and $c_{ikjj'}$ settings which we convert into per-node processing configurations (§7).

Extensions: Even though we present only one objective function, our framework is general to allow administrators to specify other policies. For example, link load costs can be captured using piecewise-linear functions that penalize higher values rather than just focus on the maximum value [25]. Similarly, we can ensure that the compute load never exceeds the capacity or model other load balancing goals beyond the min-max objective.

5 Split traffic analysis

Next, we focus on the scenario from Figure 4 in which we need to replicate traffic because the forward and reverse paths are asymmetric. For simplicity, we assume that there is only one data center node, rather than generalized mirror sets. Thus, we use c_{ikj} instead of $c_{ikjj'}$, implicitly fixing a single mirror node $R_{j'}$ for all R_j.

To model this scenario, we modify how the routing paths for each ingress-egress pair $\mathcal{T}_{ik}$ are specified. In the previous section, we assumed that the forward and reverse paths are symmetric and thus each $\mathcal{T}_{ik}$ has a unique path P_{ik}. In the case where these paths are asymmetric or non-overlapping, instead of defining a single set of eligible NIDS nodes P_{ik}, we define three types of nodes:

1. P_{ik}^{fwd} that can observe the "forward" direction.[2]
2. P_{ik}^{rev} that can observe the "reverse" direction.
3. P_{ik}^{common}, the set of nodes in the (possibly null) intersection of the forward and reverse directions.

We assume here that these types of nodes can be identified from the network's routing policy [40] or inferred from well-known measurement techniques [23]. Having identified these common, forward, and reverse nodes, we split the coverage constraint in Eq (1) into two separate equations as follows:

$$\forall i,k : coverage_{ik}^{fwd} = \sum_{j:R_j \in P_{ik}^{common}} d_{ikj} + \sum_{j:R_j \in P_{ik}^{fwd}} c_{ikj}^{fwd} \quad (7)$$

$$\forall i,k : coverage_{ik}^{rev} = \sum_{j:R_j \in P_{ik}^{common}} d_{ikj} + \sum_{j:R_j \in P_{ik}^{rev}} c_{ikj}^{rev} \quad (8)$$

Now, for stateful analysis, the notion of coverage is meaningful only if both sides of the session have been monitored. Thus, we model the effective coverage as the minimum of the forward and reverse coverages:

$$\forall i,k : coverage_{ik} = \min\{coverage_{ik}^{fwd}, coverage_{ik}^{rev}, 1\} \quad (9)$$

We make three observations with respect to the above equations. First, the d_{ikj} only appear for the nodes in P_{ik}^{common}. Second, we need to separately specify the coverage guarantee for the forward and reverse directions for each $\mathcal{T}_{ik}$ and cap the effective coverage at 1. Third, we also allow the nodes in P_{ik}^{common} the flexibility to offload processing to the datacenter. (Because the nodes in P_{ik}^{common} also appear P_{ik}^{fwd} and P_{ik}^{rev}, they have the corresponding c_{ikj}^{fwd} and c_{ikj}^{rev} variables.)

Now, it may not always be possible to ensure complete coverage for some deployment scenarios. That is, for a particular combination of forward-reverse paths, and a given constraint on the maximum allowable link load, we may not have a feasible solution to ensure that each $coverage_{ik} = 1$. Our goal then is to maximize the traffic coverage or minimize detection misses. To this end, we introduce an additional term in the minimization objective to model the fraction of traffic that suffer detection misses because we cannot monitor both sides of the connection. That is,

$$MissRate = \frac{\sum_{i,k}(1 - coverage_{ik}) \times |\mathcal{T}_{ik}|}{\sum_{i,k} |\mathcal{T}_{ik}|} \quad (10)$$

[2]We assume a well-defined notion of forward and reverse directions, say based on the values of the IP address.

Given the *MissRate*, we update the objective to be:

$$\text{Minimize: } \mathit{LoadCost} + \gamma \mathit{MissRate}$$

with γ set to a large value to have a very low miss rate.

In summary, the formulation to handle such split traffic is as follows. We retain the same structure for the compute load and link load equations as in Eq (2) and Eq (3) respectively. (There are small changes to incorporate the notion of c_{ikj}^{fwd} and c_{ikj}^{fwd}. We do not show these for brevity.) We replace the single coverage equation in Eq (1) with the new coverage models in Eqs (7), (8), and (9). Rather than force each coverage value to be 1 which could be infeasible to achieve, we focus instead on minimizing the effective miss rate by changing the objective function.

Extensions: We can extend the model to quantify *MissRate* in terms of the i,k combination with the largest fraction of detection misses (i.e., $\mathit{MissRate} = \max_{i,k}(1 - \mathit{coverage}_{ik})$), or consider a general weighted combination of these coverage values to indicate higher priority for some types of traffic.

6 NIDS with aggregation

Next, we proceed to the third opportunity for scaling via aggregation. The high-level idea is to split a NIDS task into multiple sub-tasks that can be distributed across different locations. Each NIDS node generates *intermediate reports* that are sent to an aggregation point to generate the final analysis. As a concrete example, we focus on the `Scan` detection module that counts the number of distinct destination IP addresses to which a given source has initiated a connection in the previous measurement epoch. For clarity, we focus on using aggregation without replication and assume a single symmetric path for each ingress-egress pair. This means that we just need to assign the local processing responsibilities captured by the d_{ikj} variables.

Because the choice of intermediate reports and aggregation points may vary across different detection tasks, we use a general notion of *network distance* between node R_j and the location to which these reports are sent. This is captured by Comm_{ikj}; the indices indicate that the location may depend on the specific class i and path k. For example, in `Scan` detection, we may choose to send the reports back to the ingress for the host because it is in the best position to decide if an alert should be raised, e.g., based on past behavior.

We do, however, need to be careful in choosing the granularity at which we distribute the work across nodes. Consider the `Scan` detection example in Figure 8 where our goal is to count the number of destinations that each source contacts. Suppose there are two sources S_1, S_2 contacting four destinations D_{1-4} as shown and there are

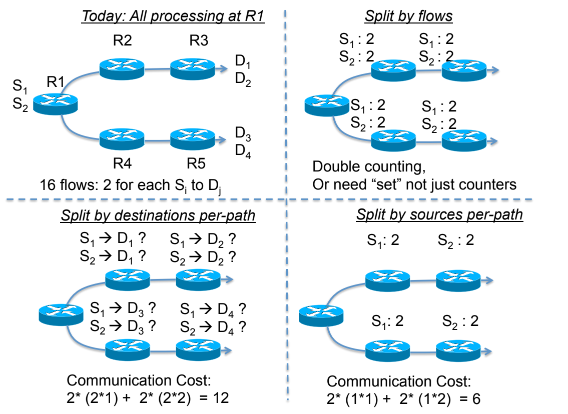


Figure 8: *Different options for splitting the* `Scan` *detection responsibilities*

Minimize $\mathit{LoadCost} + \beta \times \mathit{CommCost}$ subject to

$$\mathit{LoadCost} = \max_{r,j}\{\mathit{ResLoad}_j^r\} \quad (11)$$

$$\mathit{CommCost} = \sum_{i,k,j}(|\mathscr{T}_{ik}| \times d_{ikj}) \times \mathit{Rec}_i \times \mathit{Comm}_{ikj} \quad (12)$$

$$\forall i,k: \sum_{j:R_j \in P_{ik}} d_{ikj} = 1 \quad (13)$$

$$\forall r,j: \mathit{ResLoad}_j^r = \frac{1}{\mathit{Cap}_j^r} \sum_{\substack{i,k:\\ R_j \in P_{ik}}} \mathit{Req}_i^r \times |\mathscr{T}_{ik}| \times d_{ikj} \quad (14)$$

$$\forall i,k,j: 0 \le d_{ikj} \le 1 \quad (15)$$

Figure 9: *LP formulation for aggregation*

two flows for every src-dst pair. The high-level idea here is that each NIDS runs a per-src `Scan` counting module on its assigned subset of the traffic. Then in the second step, each node sends these local per-src counters to the aggregation point, which outputs the final result of suspicious sources. Now, we could choose three different strategies to split the monitoring responsibilities:

1. *Flow-level:* The nodes on a path split the traffic traversing that path on a per-flow basis, run a local `Scan` detection module on the set of observed flows and send intermediate reports back to the ingress.
2. *Destination-level:* Instead of splitting the traffic on a path by flows, we do a split based on destinations for each path. In the example, node R2 checks if each source contacted D_1, node R3 for D_2, and so on.
3. *Source-level:* The other alternative is for each node to focus on a subset of the sources on each path; e.g., R2 and R3 monitor S_1 and S_2, respectively.

Notice that with a flow-based split, if we only report per-src counters, then we could end up overestimating the number of destinations if a particular source-destination pair has multiple flows. In this case, each node must report the full set of $\langle src, dst \rangle$ tuples, thus incurring a larger communication cost. The aggregator then has to compute the logical union of the sets of destinations reported for each source. With a destination-based split, we do not have this double counting problem. The aggregator simply adds up the number of destinations reported from each node on each path. However, in the worst case, the number of entries each node reports will be equal to the number of sources. Thus, the total communication cost could be 12 units, assuming aggregation is done at R1: each node sends a 2-row report (one row per-src), the report from R2, R4 traverses one hop and those from R3, R5 take two hops. The third option of splitting based on the sources provides both a correct result without over counting and also a lower communication cost of 6 units. Each node sends a report consisting of the number of destinations each source contacts and the aggregator can simply add up the number of destinations reported across the different paths for each source. Thus, we choose the source-level split strategy.

Based on such analysis-specific insights, we can determine a suitable reporting and aggregation strategy. In practice, there are a few natural cases that cover most common NIDS modules that can benefit from such aggregation (e.g., per-src, per-destination). Having chosen a suitable granularity of intermediate reports, we need as input the *per-report size* Rec_i (in bytes) for class i.[3]

As in the previous section, we want to balance the tradeoff between the computation cost and the communication cost. Because the size of the reports (at most a few MB) is unlikely to impact the link load adversely, we drop the *MaxLinkLoad* constraint (Eqs (3), (4)). Instead, we introduce a new communication cost term *CommCost* in the objective, with a weight factor β, which is scaled appropriately to ensure that the load and communication cost terms are comparable. We have the familiar coverage constraint in Eq (13), and the resource load model in Eq (14). (Because there is no traffic replication, the c variables do not appear here.) The additional equation required here is to model the total communication cost *CommCost* in Eq (12). For each entry, this is simply the product of the volume of traffic, the per-unit record size, and the network distance as shown.

7 Implementation

We start by describing how the management engine translates the output of the LP optimizations into device configurations. Then, we describe how we implement these management decisions using a *shim* layer that allows us to run off-the-shelf NIDS software.

[3]This also depends on how these reports are encoded, e.g., key-value pairs for a source-split.

7.1 Optimization and configurations

We solve the LP formulations described in the previous sections using off-the-shelf LP solvers such as `CPLEX`. Given the solution to the optimization, we run a simple procedure to convert the solution into a configuration for each shim instance. (For completeness, we provide the pseudocode in Figure 21 in Appendix A.) The key idea is to map the decision variables — i.e., d_{ikj} and $c_{ikjj'}$ values — into a set of non-overlapping *hash ranges* [39]. For each i,k combination, we first run a loop over the d_{ikj} values, mapping each to a hash-range, and extending the range as we move to the next j. We then run a similar loop for the $c_{ikjj'}$. (The specific order of the NIDS indices for each path do not matter; we just need some order to ensure the ranges are non-overlapping.) Because the optimization frameworks ensure that these d_{ikj} and $c_{ikjj'}$ add up to 1 for each i,k pair, we are guaranteed that the union of these hash ranges covers the entire range $[0,1]$.

7.2 Shim layer

To allow network operators to run their existing NIDS software without needing significant changes, we interpose a lightweight *shim* between the network and the NIDS. We implement this using the Click modular software router [28] with a combination of default modules and a custom module (255 lines of C++ code). The shim maintains persistent TCP tunnels with its mirror node(s) to replicate the traffic and uses a virtual TUN/TAP interface [14] to the local NIDS process. This requires a minor change to the way the NIDS process is launched so that it reads from the virtual interface rather than a physical interface. Most NIDS software provide the interface as an input argument and thus this change is minimal and requires no changes to the NIDS internals. We tested two popular NIDS: Bro [34] and Snort [1]; both had no difficulties running on top of the shim layer.

The shim takes as input the configuration output from the procedure described above, which specifies for each class and ingress-egress pair the hash range that needs to be processed locally and what needs to offloaded to the mirror node(s). As a packet arrives, the shim computes a lightweight hash [3] of the IP 5-tuple (src/dst IPs, src/dst ports, and protocol). It looks up the corresponding class and ingress-egress combination (e.g., based on the port numbers and src/dst IPs) to infer the assigned hash range and decides whether to send this packet to the local NIDS process, replicate it to a mirror node, or neither. One subtle issue here is how this hash is computed. In the case of session-level analysis, we need to ensure that this hash is bidirectional [48]. For aggregation, the hash is over the appropriate field used for splitting the analysis.

7.3 Aggregation scripts

Aggregation requires two simple scripts: (1) one at each NIDS that periodically sends reports to the aggregator and (2) one at the aggregator to post-process these reports. For `Scan` detection, we want to report sources that contact $> D$ destinations. Now, an individual NIDS' observation may not exceed D, but the aggregate might. To this end, we apply the threshold D only at the aggregator and configure each individual NIDS to have a reporting threshold of $D = 0$, to retain the same detection semantics as running the scan detector at the ingress node.

8 Evaluation

We begin by evaluating the performance of the individual components of our system implementation and an end-to-end "live" emulation in §8.1. We then use simulations to evaluate the sensitivity of our system to various parameters with respect to replication in §8.2, split traffic in §8.3, and aggregation in §8.4.

We use real network topologies from educational backbones (Internet2, Geant) and inferred PoP-level topologies from Rocketfuel [45]. For each topology, we construct a traffic matrix for every pair of ingress-egress PoPs using a gravity model based on city populations [35], with shortest-path routing based on hop counts. For ease of presentation, we consider a single aggregate traffic class; i.e., we do not partition the traffic based on port numbers.

8.1 System evaluation

Computation time: Table 1 shows the time to compute the optimal solution for different PoP-level topologies using an off-the-shelf LP solver (`CPLEX`). This result shows that the time to recompute optimal solutions is well within the timescales of network reconfigurations (typically on the order of few minutes).

Topology	# PoPs	Time (s)	
		Replication	Aggregation
Internet2	11	0.05	0.02
Geant	22	0.10	0.02
TiNet (AS3257)	41	0.29	0.02
Telstra (AS1221)	44	0.40	0.03
Sprint (AS1239)	52	1.30	0.05
Level3 (AS3356)	63	1.19	0.04
NTT (AS2914)	70	1.59	0.11

Table 1: *Time to compute the optimal solution for the replication and aggregation formulation.*

Shim overhead: The additional hash computations and lookups impose little overhead over the packet capture that the NIDS has to run natively as well. In our microbenchmarks, the shim implementation does not introduce any (additional) packet drops up to an offered load of 1 Gbps for a single-threaded Bro or Snort process running on a Intel Core i5 2.5GHz machine.

Live emulation in Emulab: To investigate the benefits of off-path replication, we evaluate our system under realistic network conditions using Emulab [52] with an emulated Internet2 topology with 11 nodes. We implemented a realistic traffic generator in Python using `Scapy` [13] that takes as input the topology, traffic matrix, and template traces, and that generates traffic according to these. We used real full-payload packet traces captured from a small enterprise network as the "seed" templates for our generator [6]. One subtle issue is the need to faithfully emulate the ordering of packets within a logical session. To this end, we introduced a stateful "supernode" that is logically connected to every network ingress and that injects packets within each session in order and at the appropriate ingress. We used the `BitTwist` tool for trace injection [2]. Each NIDS node runs on a Pentium III 850 Mhz node with 512 MB of RAM[4] running Snort (version 2.9.1) using the default configuration of rules and signatures.

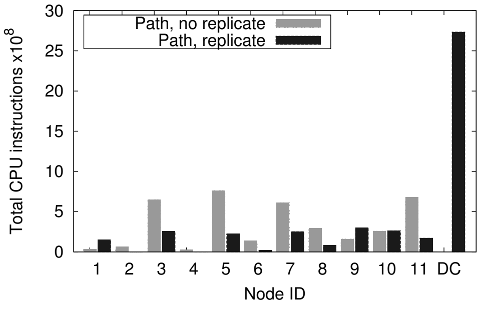


Figure 10: *Maximum absolute CPU usage of each NIDS node in our Emulab experiment*

Figure 10 shows the total number of CPU instructions used by the Snort process (measured using the `PAPI` performance instrumentation library [10]) on each NIDS node for the emulated Internet2 topology with 11 nodes. The result shows the configurations for two NIDS architectures: *Path, No replicate* which emulates the work of Sekar, et al. [38] and *Path, Replicate* which represents our framework from §4. For our setup, we ran the formulation with a single data center (DC) with 8$\times$ the capacity of the other NIDS nodes and assuming *MaxLinkLoad* $=$ 0.4. (We did not explicitly create a datacenter node in our Emulab setup due to constraints in exactly emulating a 8$\times$ larger node.) Figure 10 confirms that replication provides 2$\times$ reduction in resource usage on the maximally loaded node (excepting the DC). This result is identical to that obtained using trace-driven simulations, as will be shown in Fig. 14, allowing us to conclude that sensitivity analysis performed in §8.2 is representative of live performance.

[4]The choice of low-end nodes is not an implementation artifact. We did so to ensure repeatability as it is hard to obtain a large number of high-end nodes for extended periods of time on Emulab.

8.2 Replication: Sensitivity analysis

In this section we study the effect of varying a number of parameters on the performance of our system. Due to the difficulty of scaling our Emulab setup for larger topologies and such sensitivity analysis, we use trace-driven analysis for these evaluations.

Setup: To model the total traffic volume, we start with a baseline of 8 million sessions for the Internet2 network with 11 PoPs, and then scale the total volume for other topologies linearly proportional to the number of PoPs. We model the link capacities $LinkCap_l$ as follows. We compute the traffic volume traversing the maximum congested link (assuming the above shortest path routes). Then, we set the link capacity of each to be $3\times$ this traffic load on the most congested link. As such, $\max_l\{Background_l\} = 0.3$; this reflects typical link utilization levels in networks today [25]. To model the node capacities Cap^r_j, we emulate the Ingress deployment and find the maximum resource requirement across the network, and provision each node with this inferred capacity. Thus, by construction the Ingress deployment has a maximum compute load of one. We model a single data center with $\alpha\times$ the capacity of the other NIDS nodes.

In this discussion, we examine the effects of varying the location and capacity of the data center node (Cap^r_d), the maximum allowed link load with replication (*MaxLinkLoad*), alternative local replication architectures, and the impact of traffic variability.

Choice of datacenter location: The first parameter of interest is the placement of the datacenter. Here, we fix the datacenter capacity to be $10\times$ the single NIDS capacity, but choose different locations based on four natural strategies that network administrators can try:

1. *Max Vol Source:* at the PoP from which most traffic originates.
2. *Max Vol Obs:* at the PoP that observes the most traffic, including traffic for which this is a transit PoP.
3. *Max Paths:* the PoP which lies on the most end-to-end shortest paths.
4. *Min Avg Distance:* the PoP which has the smallest average distance to every other PoP (the medoid).

In general, the chosen location depends on the topology and traffic matrix; in some cases we observe that the strategies result in the same choice of locating the datacenter.

Figure 11 shows the maximum compute load in the network across the topologies for the different choices of datacenter location. The result shows that for most topologies the gap between the different placement strategies is very small and in practice the *Max Vol Obs* strategy works well across all topologies. Thus for the rest of the evaluation, we choose this placement strategy.

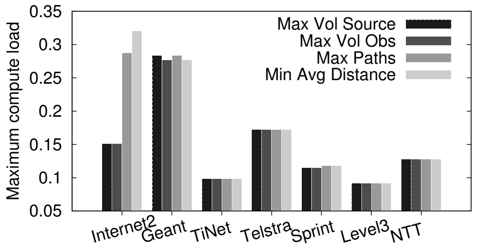


Figure 11: *Exploring strategies for datacenter placement with MaxLinkLoad* $= 0.4$.

Effect of increasing allowed link load: Next, we fix the placement of the datacenter to *Max Vol Obs* and the capacity to $10\times$, and study the impact of increasing *MaxLinkLoad* in Figure 12. For most topologies, we observe diminishing returns beyond *MaxLinkLoad* $= 0.4$, since at that value, the compute load on the datacenter is close to the load on the maximum NIDS node as well.

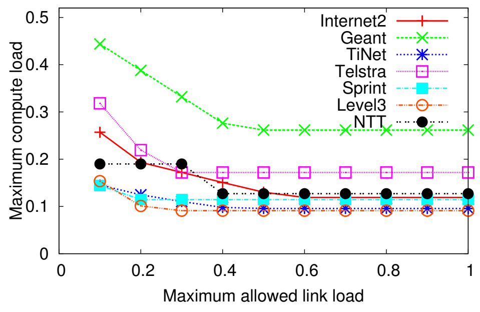


Figure 12: *Varying MaxLinkLoad with datacenter capacity of* $10\times$ *located with the* Max Vol Obs *strategy.*

Increasing the data center capacity: A natural question then is how much to provision the datacenter. To address this, we studied the impact of varying the datacenter capacity. Most topologies show a natural diminishing property as we increase the capacity, with the "knee" of the curve occurring earlier with lower link load. This is expected; with lower *MaxLinkLoad*, there are fewer opportunities for replicating traffic to the datacenter and thus increasing the datacenter capacity beyond 8–10$\times$does not really help (Figure 23, Appendix B).

Visualizing maximum loads: To better understand the previous effects, we visualize a high-level summary of how the optimization allocates the compute and offload responsibilities throughout the network. We consider four configurations here: *MaxLinkLoad* $\in \{0.1, 0.4\}$ and a datacenter capacity Cap^r_d of $2\times$ and $10\times$. Figure 13 shows the difference between the compute load on the datacenter node (*DCLoad*) and the maximum compute load on internal NIDS nodes (*MaxNIDSLoad*) for the different topologies. We see that at low link load and high data center capacity (*MaxLinkLoad* $= 0.1$ and DC=$10\times$), the datacenter is underutilized. With larger link loads or lower link capacity, we find that the load stress on the datacenter is the same as the maximum load across the network (i.e., the gap is zero).

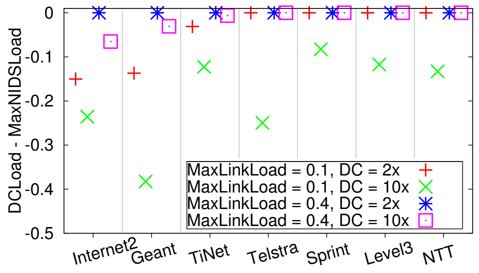


Figure 13: *Comparing the compute load on the datacenter vs. maximum load on interior NIDS nodes.*

Comparison to alternatives: Using the previous results as guidelines, we pick a configuration with the datacenter capacity fixed at 10× the single NIDS capacity located at the *Max Vol Obs* PoP, with *MaxLinkLoad* $= 0.4$. Figure 14 compares this configuration (labeled *Path, Replicate*) against two alternatives: (1) today's *Ingress*-only deployment where NIDS functions run at the ingress of a path; and (2) *Path, No Replicate*, the strictly on-path NIDS distribution proposed by Sekar, et al. [38]. One concern is that our datacenter setup has more aggregate capacity. Thus, we also consider a *Path, Augmented* approach where each of the N NIDS nodes gets a $\frac{1}{N}$ share of the 10× additional resources. The fact that we can consider these alternative designs within our framework further confirms the *generality* of our approach.

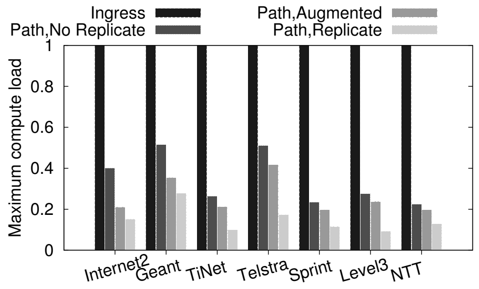


Figure 14: *Maximum compute load across topologies with different NIDS architectures.*

Recall that the current deployments of *Ingress*-only have a maximum compute load of one by construction. The result shows that the datacenter setup has the best overall performance; it can reduce the *MaxLoad* 10× compared to today's deployments and up to 3× compared to the proposed on-path distribution schemes.

Local offload: The above results consider a setup where the network administrator has added a new datacenter. Alternatively, they can use the existing NIDS infrastructure with *local* replication strategies. Specifically, we consider the *MirrorSet* consisting of 1-hop or 2-hop

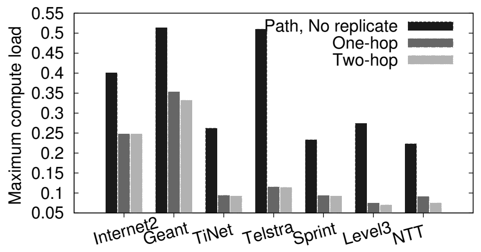


Figure 15: *Local one- and two-hop replication.*

neighbors in addition to the existing on-path distribution. Figure 15 compares the maximum compute load vs. a pure on-path distribution again setting *MaxLinkLoad* $=$ 0.4. Across all topologies, allowing replication within a one-hop radius provides up to 5× reduction in the maximum load. We also see that going to two hops does not add significant value beyond one-hop offload. This suggests a replication-enhanced NIDS architecture can offer significant benefits even without needing to augment the network with additional compute resources.

Performance under traffic variability: The results so far consider a static view with a single traffic matrix. Next, we evaluate the effect of traffic variability. To obtain realistic temporal variability patterns, we use traffic matrices for Internet2 [5]. From this, we compute empirical CDFs of how each element in a traffic matrix varies (e.g., probability that the volume is between 0.6× and 0.8× the mean). Then, using these empirical distributions we generate 100 time-varying traffic matrices using the gravity model for the mean volume.

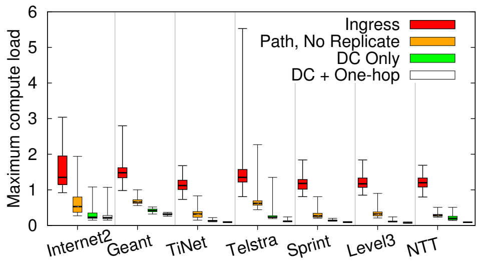


Figure 16: *Comparison between NIDS architectures in the presence of traffic variability.*

Figure 16 summarizes the distribution of the peak load across these 100 runs using a box-and-whiskers plot showing the minimum, 25th %ile, median, 75th %ile, and the maximum observed load. We consider four NIDS architectures: the *Ingress*, the *Path No replicate*, the *Path, replicate* with a datacenter node 10× capacity (labeled `DC Only`), and *Path, replicate* with the flexibility to offload responsibilities to either a datacenter and within a 1-hop radius (labeled `DC + One-hop`). We find that the replication-enabled NIDS architectures outperform the non-replication strategies significantly, with the median values roughly mirroring our earlier results.

The worst-case performance of the no-replication architectures can be quite poor; e.g., for Telstra the worst-case compute load is 5.8 which significantly exceeds the provisioned capacity. (Ideally, we want the maximum compute load to be less than 1.) We also analyzed how the augmentation strategy from Figure 14 performs; the worst-case load with the *Path, Augmented* case is 4× more than the replication enabled architecture (not shown).

8.3 Performance with routing asymmetry

Next, we evaluate the benefits of replication in enabling new analysis capabilities for scenarios where the forward and reverse flows may not traverse the same route as we saw in §2.

We emulate routing asymmetry as follows. For each ingress-egress pair k, we assume the forward traffic traverses the expected shortest path from the ingress to the egress; i.e., P_{ik}^{fwd} is the shortest-path route. However, we set the reverse path P_{ik}^{rev} such that the expected *overlap* (over all ingress-egress pairs) between the forward and reverse paths reaches a target overlap ratio θ. Here, we measure the overlap between two paths P_1 and P_2 in terms of the Jaccard similarity index: $\frac{P_1 \cap P_2}{P_1 \cup P_2}$, which is maximum (= 1) when they are identical and lowest (= 0) if there is no overlap. For each end-to-end path, we precompute its overlap metric with every other path. Then, given a value of θ' (drawn from a Gaussian distribution with mean = θ and standard deviation = $\frac{\theta}{5}$), we find a path from this precomputed set that is *closest* to this target value.[5] For each target θ, we generate 50 random configurations. For each configuration, we run the extended formulation from §5 for the Ingress-only architecture, the *Path, no replicate* architecture, and our proposed framework with a datacenter (with Max Vol Obs). We report the *median* across the 50 runs for two metrics: the detection *miss rate* —the total fraction of traffic that could not be analyzed effectively by any NIDS node, and the compute load as in the previous evaluations.

Figure 17 shows the median miss rate as a function of the overlap factor for the different configurations. We see that the miss rate with an *Ingress*-only setup is greater than 85% even for high values of the overlap metric. The *MaxLoad* curve in Figure 18 is interesting because *Ingress* is lower than the other configurations. The reason is that there is little useful work being done here–It ignores more than 90% of the traffic! Another curious feature is that *MaxLoad* for the replication architecture first increases and then decreases. In this setup with low overlap, the datacenter is the most loaded node. At low θ, however, the *MaxLinkLoad* constraint limits the amount of traffic that can be offloaded and thus the datacenter load is low.

[5]The exact details of how these paths are chosen or the distribution of the θ values are not the key focus of this evaluation. We just need some mechanism to generate paths with a target overlap ratio.

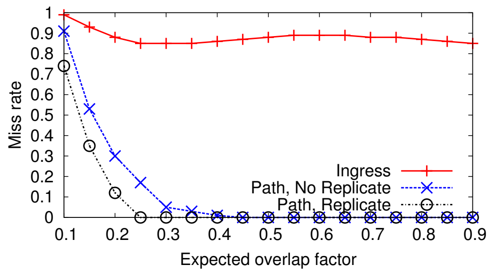


Figure 17: *Detection miss rate vs. degree of overlap*

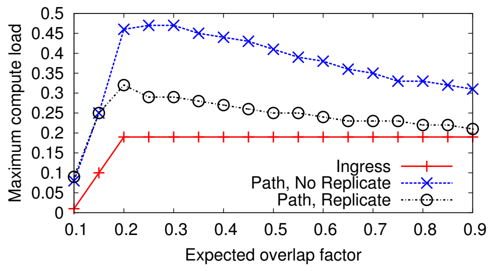


Figure 18: *Maximum load vs. degree of overlap*

8.4 NIDS with aggregation

In this section, we highlight the benefits of aggregation using the framework from §6. As discussed earlier, we focus on `Scan` detection.

Figure 19 shows how varying β trades off the communication cost (*CommCost*) and compute cost (*LoadCost*) in the resulting solution, for the different topologies. Because different topologies differ in size and structure, we normalize the x- and y-axes using the maximum observed *LoadCost* and *CommCost* respectively over the range of β for each topology. As such, the point closest to the origin can be viewed as the best choice of β for the corresponding topology. This figure shows that for many topologies, there are choices of β that yield relatively low *CommCost* and *LoadCost* simultaneously, e.g., both being less than 40% of their maximums.

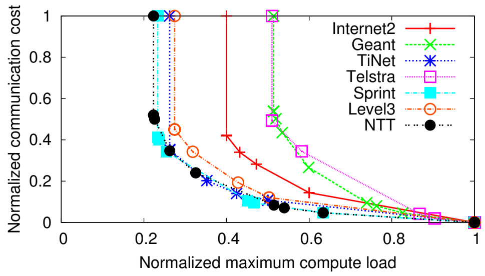


Figure 19: *Tradeoff between the compute load and communication cost with aggregation as we vary β*

To illustrate the load balancing benefits of aggregation, Figure 20 shows the ratio of the compute load of

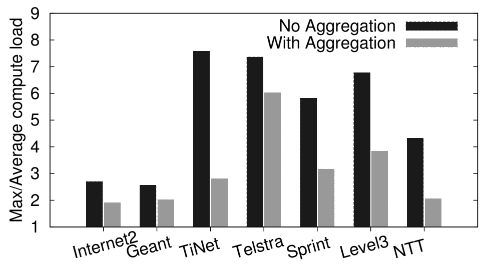


Figure 20: *Ratio between maximum and average compute load with and without aggregation.*

the most loaded node to the average compute load. Here, for each topology, we pick the value of β that yields the point closest to the origin in Figure 19. A higher number represents a larger variance or imbalance in the load. Figure 20 compares this ratio to the same ratio when no aggregation is used. As we can see, aggregation reduces the load imbalance substantially (up to 2.7×) for many topologies.

9 Discussion

Extending to NIPS: Our framework can be extended to the case of intrusion prevention systems (NIPS) as well. Unlike NIDS, however, NIPS are on the critical forwarding path which raises two additional issues that we need to handle. These arise from the fact that we are not actually replicating traffic in this case; rather, we are *rerouting* it. First, we can no longer treat the $Background_l$ as a constant in the formulation. Second, we need to ensure that the latency penalty for legitimate traffic due to rerouting is low. This introduces a new dimension to the optimization to minimize the latency. In practice, the latency impact can be reduced by a more careful choice of the number (e.g., more than 1) and placement (e.g., *Min Avg Distance* may be a better choice) of datacenters.

Shim scalability: Our current shim implementation imposes close to zero overhead for a single-threaded NIDS running on a single core machine for traffic up to 1 Gbps. We plan to extend our implementation using recent advances that further improve packet capture speeds [12] and extend to multi-core architectures [11].

Combining aggregation and replication: Our results show that replication and aggregation independently offer substantial benefits. As future work we plan to explore if a unified formulation that combines both opportunities offers further improvements. For example, we might be able to leverage the existing replication to reduce the communication cost needed for aggregation. One issue, however, is that the analyses benefiting from aggregation may need to split the traffic at a different granularity (e.g., per-src or per-destination) vs. those exploiting replication (e.g., stateful signature matching on a per-session basis). Thus, we need a more careful shim design to avoid duplicating the effort in packet capture across different nodes in order to combine these ideas.

10 Related Work

Scaling NIDS hardware: NIDS involve complex processing and computational intensive tasks (e.g., string and regular-expression matching). There are many proposals for better algorithms for such tasks (e.g. [42]), using specialized hardware such as TCAMs (e.g., [53, 32]), FPGAs (e.g., [31, 33]), or GPUs (e.g., [50, 49]). The dependence on specialized hardware increases deployment costs. To address this cost challenge, there are ongoing efforts to build scalable NIDS on commodity hardware to exploit data-level parallelism in NIDS workloads (e.g., [26, 48, 36, 44, 29, 51]). These efforts focus on scaling *single-vantage-point* implementations and are thus complementary to our work. Our framework allows administrators to optimally use their existing hardware or selectively add NIDS clusters.

Network-wide management: Our use of centralized optimization to assign NIDS responsibilities follows in the spirit of recent work in the networking community (e.g., [27, 19]). The use of hash-based sampling to coordinate monitoring was proposed in previous work (e.g., [23, 39]). The work closest to ours is by Sekar, et al. [38]. While we share their motivation for looking beyond single-vantage-point scaling, our work differs in three key aspects. First, we generalize the on-path distribution to include opportunities for replication and aggregation. Second, their framework cannot handle the types of split-traffic analysis with asymmetric routes as we showed in Figure 17. Third, on a practical note, their approach requires detailed source-level modifications to the NIDS software. By interposing a lightweight shim we allow administrators to run off-the-shelf NIDS software even without access to the source code.

Distributed NIDS: Prior work makes the case for network-wide visibility and distributed views in detecting anomalous behaviors (e.g., [47, 43, 20, 16, 30]). These focus primarily on algorithms for combining observations from multiple vantage points. Furthermore, specific attacks (e.g., DDoS attacks [37], stepping stones [54]) and network scenarios (e.g., the asymmetric routing as in §2) inherently require an aggregate view. Our focus is not on the specific algorithms for combining observations from multiple vantage points. Rather, we build a framework for balancing the computation and communication tradeoffs in enabling such aggregated analysis.

11 Conclusions

While there are many advances in building better NIDS hardware, there is a substantial window before networks can benefit from these in practice. This paper was motivated by the need to complement the work on scaling NIDS hardware with techniques to help network managers better utilize their existing NIDS deployments.

To this end, we propose a general NIDS architecture to leverage three scaling opportunities: on-path distribution to offload processing to other nodes on a packet's routing path, traffic replication to off-path nodes (e.g., to NIDS clusters), and aggregation to split expensive NIDS tasks. We implemented a lightweight shim that allows networks to realize these benefits without any modifications to existing NIDS software beyond simple configuration tweaks. Our results on many real-world topologies shows that this architecture reduces the maximum compute load significantly, provides better resilience under traffic variability, and offers improved detection coverage for scenarios needing network-wide views.

Acknowledgements

This work was supported in part by ONR grant N000141010155, NSF grants 0831245 and 1040626, and the Intel Science and Technology Center for Secure Computing.

References

[1] `http://www.snort.org`.

[2] `http://bittwist.sourceforge.net`.

[3] Bob hash. `http://burtleburtle.net/bob/hash/doobs.html`.

[4] Cisco blade servers. `http://www.cisco.com/en/US/products/ps10280/index.html`.

[5] Internet2 trafficx matrices. `http://www.cs.utexas.edu/~yzhang/research/AbileneTM`.

[6] M57 packet traces. `https://domex.nps.edu/corp/scenarios/2009-m57/net/`.

[7] Magic quadrant for network intrusion prevention systems. `www.stonesoft.com/export/download/pdf/IPS_MQ_2010_208628.pdf`.

[8] Network security spending to soar in the next 5 year. `http://www.v3.co.uk/v3-uk/news/1998293/network-security-spending-soar`.

[9] One-stop security. `http://www.pcmag.com/article2/0,2817,1829582,00.asp#fbid=eHC0T1KbMJp`.

[10] PAPI: Performance Application Programming Interface. `http://icl.cs.utk.edu/papi/`.

[11] Pfq homepage. `http://netserv.iet.unipi.it/software/pfq/`.

[12] Pf_ring. `http://www.ntop.org/products/pf_ring/`.

[13] Scapy packet manipulation toolkit. http://www.secdev.org/projects/scapy/.

[14] Tun/tap. `http://www.kernel.org/doc/Documentation/networking/tuntap.txt`.

[15] World intrusion detection and prevention markets. `http://www-935.ibm.com/services/us/iss/pdf/esr_intrusion-detection-and-prevention-systems-markets.pdf`.

[16] M. Allman, C. Kreibich, V. Paxson, R. Sommer, and N. Weaver. Principles for Developing Comprehensive Network Visibility. In *USENIX Workshop on Hot Topics in Security*, 2008.

[17] M. Caesar, D. Caldwell, N. Feamster, J. Rexford, A. Shaikh, and J. van der Merwe. Design and implementation of a Routing Control Platform. In *Proc. of NSDI*, 2005.

[18] G. R. Cantieni, G. Iannaccone, C. Barakat, C. Diot, and P. Thiran. Reformulating the Monitor Placement problem: Optimal Network-Wide Sampling. In *Proc. of CoNeXT*, 2006.

[19] M. Casado, T. Garfinkel, A. Akella, M. Friedman, D. Boneh, N. Mckeown, and S. Shenker. SANE: A Protection Architecture for Enterprise Networks. In *USENIX Security*, 2006.

[20] F. Cuppens and A. Miege. Alert correlation in a cooperative intrusion detection framework. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[21] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Knies, M. Manesh, and S. Ratnasamy. RouteBricks: Exploiting Parallelism to Scale Software Routers. In *Proc. SOSP*, 2009.

[22] H. Dreger, A. Feldmann, V. Paxson, and R. Sommer. Predicting the Resource Consumption of Network Intrusion Detection Systems. In *Proc. RAID*, 2008.

[23] N. Duffield and M. Grossglauser. Trajectory Sampling for Direct Traffic Observation. In *Proc. of ACM SIGCOMM*, 2001.

[24] A. Feldmann, A. G. Greenberg, C. Lund, N. Reingold, J. Rexford, and F. True. Deriving Traffic Demands for Operational IP Networks: Methodology and Experience. In *Proc. of ACM SIGCOMM*, 2000.

[25] B. Fortz, J. Rexford, and M. Thorup. Traffic Engineering with Traditional IP Routing Protocols. IEEE Communications Magazine, Oct. 2002.

[26] L. Foschini, A. V. Thapliyal, L. Cavallaro, C. Kruegel, and G. Vigna. A Parallel Architecture for Stateful, High-Speed Intrusion Detection. In *Proc. ICISS*, 2008.

[27] A. Greenberg, G. Hjalmtysson, D. A. Maltz, A. Meyers, J. Rexford, G. Xie, H. Yan, J. Zhan, and H. Zhang. A Clean Slate 4D Approach to Network Control and Management. *ACM SIGCOMM CCR*, 35(5), Oct. 2005.

[28] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[29] C. Kruegel, F. Valeur, G. Vigna, and R. A. Kemmerer. Stateful Intrusion Detection for High-Speed Networks. In *Proc. IEEE Symposium on Security and Privacy*, 2002.

[30] A. Lakhina, M. Crovella, and C. Diot. Diagnosing Network-Wide Traffic Anomalies. In *Proc. of ACM SIGCOMM*, 2004.

[31] J. Lee, S. H. Hwang, N. Park, S.-W. Lee, S. Jun, and Y. S. Kim. A high performance NIDS using FPGA-based regular expression matching. In *Proc. Symposium on Applied Computing (SAC)*, 2007.

[32] C. R. Meiners, J. Patel, E. Norige, E. Torng, and A. X. Liu. Fast regular expression matching using small tcams for network intrusion detection and prevention systems. In *Proc. USENIX Security Symposium*, 2010.

[33] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating Snort IDS. In *Proc. Architecture for Networking and Communications Systems (ANCS)*, 2007.

[34] V. Paxson. Bro: A System for Detecting Network Intruders in Real-Time. In *Proc. USENIX Security Symposium*, 1998.

[35] M. Roughan. Simplifying the Synthesis of Internet Traffic Matrices. *ACM SIGCOMM CCR*, 35(5), 2005.

[36] D. L. Schuff, Y. R. Choe, and V. S. Pai. Conservative vs. optimistic parallelization of stateful network intrusion detection. In *Proc. Symposium on Principles and Practice of Parallel Programming (PPoPP)*, 2007.

[37] V. Sekar, N. Duffield, K. van der Merwe, O. Spatscheck, and H. Zhang. LADS: Large-scale Automated DDoS Detection System. In *Proc. of USENIX ATC*, 2006.

[38] V. Sekar, R. Krishnaswamy, A. Gupta, and M. K. Reiter. Network-wide deployment of intrusion detection and prevention systems. In *Proc. CoNext*, 2010.

[39] V. Sekar, M. K. Reiter, W. Willinger, H. Zhang, R. Kompella, and D. G. Andersen. cSamp: A System for Network-Wide Flow Monitoring. In *Proc. of NSDI*, 2008.

[40] A. Shaikh and A. Greenberg. OSPF Monitoring: Architecture, Design and Deployment Experience. In *Proc. of NSDI*, 2004.

[41] R. Smith, C. Estan, and S. Jha. XFA: Faster signature matching with extended automata. In *Proc. IEEE Symposium on Security and Privacy*, 2008.

[42] R. Smith, C. Estan, S. Jha, and S. Kong. Deflating the Big Bang: fast and scalable deep packet inspection with variable-extended automata. In *Proc. SIGCOMM*, 2008.

[43] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur. DIDS (distributed intrusion detection system), - motivation, architecture, and an early prototype. In *Proc. National Computer Security Conference*, 1991.

[44] R. Sommer, V. Paxson, and N. Weaver. An Architecture for Exploiting Multi-Core Processors to Parallelize Network Intrusion Prevention. *Concurrency and Computation: Practice and Experience, Wiley*, 21(10):1255–1279, 2009.

[45] N. Spring, R. Mahajan, and D. Wetherall. Measuring ISP Topologies with Rocketfuel. In *Proc. of ACM SIGCOMM*, 2002.

[46] R. Teixeira, A. Shaikh, T. Griffin, and J. Rexford. Dynamics of hot-potato routing in IP networks. In *Proc. SIGMETRICS*, 2004.

[47] A. Valdes and K. Skinner. Probabilistic alert correlation. In *Proc. RAID*, 2001.

[48] M. Vallentin, R. Sommer, J. Lee, C. Leres, V. Paxson, and B. Tierney. The NIDS Cluster: Scalable, Stateful Network Intrusion Detection on Commodity Hardware. In *Proc. of RAID*, 2007.

[49] G. Vasiliadis, S. Antonatos, M. Polychronakis, E. P. Markatos, and S. Ioannidis. Gnort: High Performance Network Intrusion Detection Using Graphics Processors. In *Proc. RAID*, 2008.

[50] G. Vasiliadis, M. Polychronakis, S. Antonatos, E. P. Markatos, and S. Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Proc. RAID*, 2009.

[51] G. Vasiliadis, M. Polychronakis, and S. Ioannidis. MIDeA: A Multi-Parallel Intrusion Detection Architecture. In *Proc. CCS*, 2011.

[52] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An integrated experimental environment for distributed systems and networks. In *Proc. OSDI*, 2002.

[53] F. Yu, T. V. Lakshman, M. A. Motoyama, and R. H. Katz. SSA: A Power and Memory Efficient Scheme to Multi-Match Packet Classification. In *Proc. ANCS*, 2005.

[54] Y. Zhang and V. Paxson. Detecting stepping stones. In *Proc. USENIX Security Symposium*, 2000.

A Mapping to configurations

For brevity, we only show the mapping step for the replication formulation in Figure 21. Solving the optimization formulation using an LP solver gives us the optimal assignments of the d_{ikj} and $c_{ikjj'}$ values. The mapping procedure runs once per class-path combination; i.e., for each i,k. For each such combination, the first loop iterates over non-zero d_{ikj} values and assigns *processing* responsibilities across the respective R_js. At each step it tracks the currently assigned range (i.e., *endrange*) and uses this value as the *startrange* for the next assignment to ensure that the ranges assigned to the different nodes are *non-overlapping*. After completing the processing assignments, it does a similar loop over the *replication* assignments, again ensuring that the assigned ranges do not overlap across the nodes.

```
for all $i,k$ do
    $startrange \leftarrow 0, endrange \leftarrow 0$
    {Start with the local processing assignments}
    for all $R_j \in P_{ik}$ s.t. $d_{ikj} > 0$ do
        $endrange \leftarrow endrange + d_{ikj}$
        $process_{i,k}(R_j) \leftarrow [startrange, endrange]$
        $startrange \leftarrow endrange$
    end for
    {Now, move to the replication assignments}
    for all $R_j \in P_{ik}$ do
        for all $R_{j'} \in MirrorSet_j$ s.t. $c_{ikjj'} > 0$ do
            $endrange \leftarrow endrange + c_{ikjj'}$
            $offload_{i,k}(R_j, R_{j'}) \leftarrow [startrange, endrange]$
            $startrange \leftarrow endrange$
        end for
    end for
end for
```

Figure 21: Mapping the optimization solution into a set of hash ranges for each shim instance

B Additional Results

B.1 Varying datacenter location

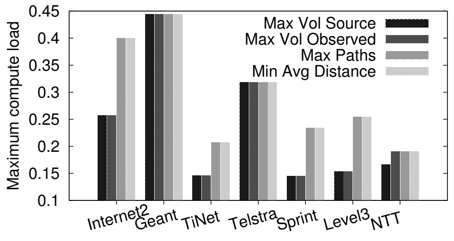
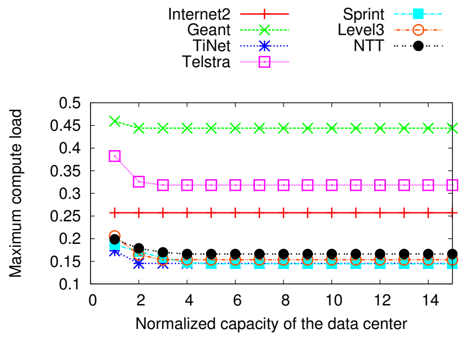


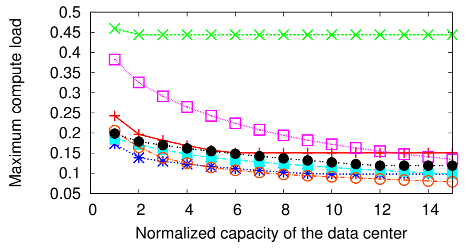
Figure 22: Choosing a data center location with $MaxLinkLoad = 0.1$.

Figure 22 shows the effect of different datacenter placement strategies, but setting the $MaxLinkLoad = 0.1$. The one interesting difference is that there is a more pronounced gap between the placement strategies for TiNet, Sprint, and Level3 at $MaxLinkLoad = 0.1$ compared to Figure 11 with $MaxLinkLoad = 0.4$. This suggests that a more careful placement evaluation might be needed under stricter communication constraints. Even in this case, however, we notice that the *Max Vol Obs* strategy is the best choice across all topologies.

B.2 Varying the datacenter capacity



(a) $\mathit{MaxLinkLoad} = 0.1$



(b) $\mathit{MaxLinkLoad} = 0.4$

Figure 23: Maximum compute load as a function of the provisioned datacenter capacity

Figure 23 shows the effect of increasing the datacenter capacity for two values of *MaxLinkLoad*. For both values, we find a natural diminishing returns property where the slope of the load reduction decreases as the data center capacity increases. The difference is that at a low value, $\mathit{MaxLinkLoad} = 0.1$, we see that the knee of the curve occurs much earlier at $X = 2$ and for some cases (e.g., Internet2) there is little to be gained by increasing the datacenter capacity at low link loads. This is expected—with a tighter bound of *MaxLinkLoad* there are very few opportunities to offload traffic processing. At $\mathit{MaxLinkLoad} = 0.4$, the knee of the curve for most topologies is around 8–10; hence, we chose this value for our evaluation setup.