PEDDA: Practical and Effective Detection of Distributed Attacks on Enterprise Networks via Progressive Multi-Stage Inference

Minzhao Lyu, Hassan Habibi Gharakheili, and Vijay Sivaraman

***Abstract*—Network attacks on enterprises are distributed in sources and versatile in patterns. However, practical solutions (firewalls) often focus on potential enterprise victims by way of coarse-grained monitoring due to their limited compute resources; thus, *ineffective* in detecting distributed sources and flows of network attacks. In contrast, fine-grained flow-level detection methods are *impractical* in handling millions of flows for large enterprises. We present PEDDA, a progressive multi-stage inference method to detect distributed attacks by leveraging dynamic controls of programmable networks. It flexibly applies inference stages, each with an orchestratable granularity, whereby packet streams are either proactively or reactively partitioned and analysed by specialised functions depending on the evolution of attacks. The granularity of each stage/function is dynamically determined by an optimisation framework subject to computing resource constraints. We prototype a proof-of-concept system consisting of three inference stages that monitors active enterprise hosts, detects and isolates those victims under attacks, and differentiates distributed sources and flows from benign ones, respectively. We evaluate the efficacy of our prototype by applying it to real traffic traces from a large enterprise network injected by DDoS attacks from a public dataset.**

***Index Terms*—DDoS detection; progressive inference; resource orchestration; programmable networks; network security**

I. Introduction

Distributed network attacks (*e.g.,* DDoS), attributed to IoT [1], botnet [2], DNS [3], and amplification [4], continue to target enterprises by large volumes of aggregate traffic, congesting their networks or paralyzing individual hosts [5]. These attacks have reached an alarming rate with high frequency in occurrence, diversity in patterns, and well-distributed sources to bypass countermeasures available on security appliances, as highlighted by academic research [6] and industry reports from Akamai [7], Black Hat Community [8], and Forcepoint reports [9]. A distributed attack may exploit different protocols (*e.g.,* DNS [4], HTTPS [10]) and use large-scale botnet devices, each with specific attack strategies such as low packet rate [11], relatively infrequent HTTP requests [12], and temporal shifts [13]. Fig. 1(a) illustrates how malicious flows sourced from distributed external attackers are mixed (and possibly share subnets) with legitimate traffic from benign external hosts. Therefore, effectively detecting a distributed attack entails determining the victim among enterprise hosts, identifying distributed sources from a large set of external hosts, and detecting malicious flows from numerous concurrent flows, annotated as red entities in Fig. 1(a).

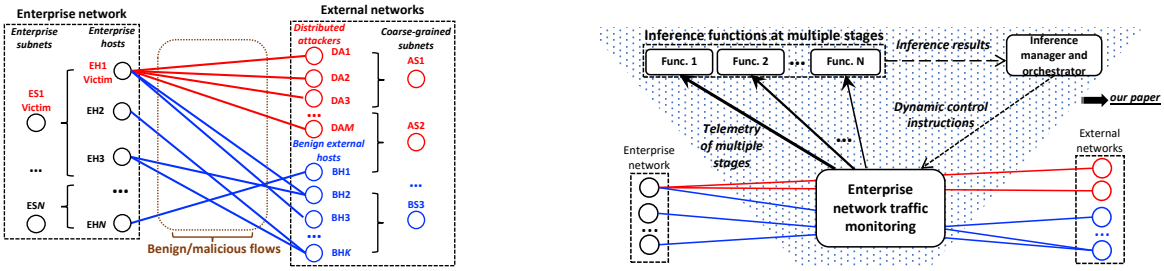
Furthermore, large and federated enterprise networks like universities often allow their sub-departments and users to set up their networked assets (*e.g.,* servers). Some assets can be relatively dynamic and hence hard to manage in practice. For example, a research group may set up/reconfigure their websites and databases without notifying the central IT department. Therefore, continuously monitoring active hosts and dynamically isolating those that may be under distributed attacks are necessary for a robust network operation.

To detect distributed network attacks[1] at the entrance of the enterprise network border (*i.e.,* "**at-destination**"), commercial solutions like firewall appliances and network intrusion detection systems (*e.g.,* Sophos [14], Fortinet [15], Cisco [16], and Palo Alto [17]) employ relatively static policies (*e.g.,* thresholds on the aggregate traffic rate) applied to a predefined list of hosts or subnets. In addition, they demand custom configurations by IT operators given their domain knowledge of the connected hosts and security risks. Such approaches have practically proven to scale to large throughput with the ability to detect typical network attacks on their critical assets (as potential victims) specified by organisational policies. However, they are unable to identify external attackers and malicious flows from benign ones as collecting and maintaining network telemetry for external hosts and flows can be expensive at scale. Thus, attack mitigation often introduces collateral damage to normal communications.

Prior research works like *BLINC* [18], *SpotLight* [19], and *AGM* [20] developed methods to detect distributed attacks. An effective distributed attack detection often requires visibility into activity patterns to the finest flow levels (*i.e.,* a graph of the entire network flows) so that not only victims but also distributed sources and flows could be precisely differentiated from the benign ones. However, such methods consume (computationally) expensive network telemetry for communications between every pair of hosts, allowing for effective detection but at high computing costs. Therefore, they often fall short in practice when employed for real-time detection in large networks of high traffic rates (*e.g.,* with millions of concurrent

M. Lyu, H. Habibi Gharakheili and V. Sivaraman are with the School of Electrical Engineering and Telecommunications, University of New South Wales, Sydney, NSW 2052, Australia (e-mails: minzhao.lyu@unsw.edu.au, h.habibi@unsw.edu.au, vijay@unsw.edu.au).

[1]Methods for detecting distributed network attacks often rely on identifying abnormal volumetric activities (*e.g.,* total packet rates or count of concurrent flows of a target host) instead of inspecting payloads/contents carried by individual packets. Deep packet inspections are often used for information-based attacks or for detecting malware.



(a) Anatomy of distributed attacks.

(b) Our concept of progressive multi-stage inference.

Figure 1: Motivating our work: (a) anatomy of distributed attacks and (b) conceptualised design of our progressive multi-stage inference via dynamic control of programmable networks.

flows and thousands of active enterprise hosts). This is one of the key issues that this paper aims to solve optimally.

In short, proprietary security middleboxes are practical but ineffective for recognising distributed attack sources and flows from benign counterparts; and detection based on flow graphs is effective but impractical for high-throughput enterprise environments. We note that the proven capability of programmable networking techniques can fill this performance gap, as it offers immense potential in making network defence functions elastic and packet routing flexible at run-time. Existing solutions [21]–[23], while demonstrating promises in flexible detection and mitigation of distributed attacks, have inspired our design to address the dilemma of balancing effective detection against practicality in operational enterprise networks.

This paper presents PEDDA, visually conceptualised in Fig. 1(b), a practical and effective system that detects distributed network attacks progressively through multiple inference stages, all dynamically instructed by reactive controls of programmable networks. Each inference stage comes with a certain specialty (*e.g.,* victims, sources, and flows), orchestratable granularity (*i.e.,* from fine-grained host-level to coarse-grained subnet-level), and manageable compute cost. In operation, network traffic is proactively processed by low-cost inference stages. A high-cost stage is reactively employed only for those packet streams partitioned by the prior low-cost stages. In addition, the granularity of each stage is dynamically orchestrated by a real-time optimisation framework subject to computing resource constraints and traffic status. We acknowledge that there exist solutions for detecting DDoS, each with specific objectives, ranging from scalable cloud-based deployment to high-performance P4 hardware to precisely detecting bots. Our paper, instead, focuses on how to optimally use/orchestrate various detection resources (could be standalone, distributed, cloud-based, or via a P4 data plane) available to an enterprise so that the real-time detection power could be maximised with a limited computing budget. We make three key contributions.

First, to motivate our PEDDA design, we model the traffic processing of legacy network security systems by four modules, namely packet dispatching, packet parsing, information gathering, and inferencing. We mathematically formulate per-packet CPU time consumption to highlight computing bottlenecks that hinder the current systems to achieve both operational practicality and detection efficacy.

Second, to combat bottlenecks and robustly handle high rates of data streaming in general, particularly during DDoS attack events, we develop PEDDA, our progressive multi-stage inference. It detects distributed attacks through multiple inference stages (*e.g.,* victims, sources, and flows) progressively – higher stages will incur more computing costs as they need finer-grained network telemetry and more complex data structures. At run-time, network traffic is by default processed by the low-cost inference stage (proactively). Depending on the output of this stage, a selected portion of packets (mostly a minor fraction of total traffic) will be inspected by high-cost inference stages for attack detection (*e.g.,* distributed sources and flows). An appropriate programmable networking technology (*e.g.,* NFV, OpenFlow, or P4) can be employed to orchestrate dynamic controls. An orchestrator is developed to dynamically apply an optimal algorithm to protect PEDDA from being overwhelmed by complex, heavy, and unpredictable traffic mixes, especially for high-cost stages. The algorithm maximises the granularity (the finest host IP level versus aggregate subnet levels) of inference stages subject to available computing resources.

Third, we realise PEDDA as a proof-of-concept prototype using a commodity server and an OpenFlow SDN switch (given its relative ease of use by the current industry) ready to be deployed in a large enterprise for real-time detection. Driven by the insights obtained from an empirical analysis of traffic for a representative enterprise network, we implement three practical inference stages to detect active enterprise hosts, victims, and distributed attackers with flows in a progressive and reactive manner. We evaluate the efficacy of our system by replaying traffic traces of a large university network injected by DDoS attack instances (obtained from a public dataset) and comparing it with state-of-the-art systems. We highlight the superiority of PEDDA in efficient real-time detection and operational practicality under high traffic rates.

The rest of this paper is organised as follows. In §II, we discuss the background technologies and prior work on various detection scenarios with a special focus on detection techniques applied at the destination, which is the primary objective of our work in this paper. §III models the complexity and highlights bottlenecks of legacy solutions for detecting DDoS attacks at the destination, motivating the design of our progressive multi-stage inference method in §IV. We implement and evaluate a proof-of-concept prototype of our proposed method with three inference stages in §V. The paper is concluded in §VI.

II. Background and Related Work

In this section, we begin by discussing three categories of distributed network attack detection, namely "at-source", "in-network", and "at-destination" (§II-A). Focusing on the "at-destination" detection techniques, we highlight their key requirements when deployed in enterprise networks as suggested by both industrial and research communities (§II-B) and discuss related works (§II-C).

A. Categories of Distributed Network Attack Detection

Methods for detecting distributed network attacks can be categorised into three classes, namely "at-source", "in-network", and "at-destination", based on where on the network they are applied [5]. Methods at the source end detect outbound attack traffic generated by hosts inside a subject network. Internet service providers adopt in-network mechanisms to filter malicious traffic before reaching the destination networks. At-destination solutions are often employed close to the border routers of a destination network (*e.g.,* an enterprise), protecting internal hosts and infrastructure from outside attackers. This paper focuses on the latter class of detection (*i.e.,* at-destination), intending to balance the granular detection in real-time against computation costs optimally.

B. Key Requirements of Attack Detection At-Destination

The network security community (both academia and industry) states certain key requirements for an ideal detection of distributed attacks on enterprise networks. These cover a range of factors (will be explained soon in §II-B1 and §II-B2) from telemetry visibility and inference precision to scalability and automaticity to deployability in practical operation in an enterprise environment.

1) Effective Detection: Distributed network attacks such as service probing (*i.e.,* host/port scans [24]–[26]) and denial-of-service (*i.e.,* DDoS [7]) on enterprise internal assets have evolved from a single source and protocol to more sophisticated and complicated forms. An attack may initiate from distributed malicious hosts (*e.g.,* compromised PCs and IoT devices [27]) and use various protocols (*e.g.,* SYN flood via HTTPS protocol or UDP reflection via Memcached port). Therefore, to recognise diversified patterns in malicious network activities, an ideal security solution would need rich **visibility** into the behavioral characteristics of all involved hosts and their network flows [28]. Importantly, malicious traffic generated by attackers is likely to be mixed with legitimate flows from benign external hosts, particularly when traffic is measured closer to victim hosts [29]. To sufficiently eliminate malicious traffic without interrupting benign activities, detection systems are expected to be **precise** in differentiating malicious external hosts and flows of an attack from benign ones.

2) Practical Operation: An attack detection system becomes practically attractive to enteprise network operators if it can cost-effectively **scale** to high rates of network traffic (*e.g.,* tens gigabits per second throughput and millions of concurrent flows). Besides, operators of large enterprises (*e.g.,* universities) may not always have a complete and up-to-date asset register (full list of hosts/devices connected to their network) [30], [31]. Thus, configuring and enforcing appropriate security policies for individual IP addresses is rather challenging (sometimes impractical). A detection system would be operationally desirable if it could automatically discover the enterprise hosts that need to be protected – we refer to this feature as **automaticity**. Lastly, perimeter security systems are vital to the operation of every enterprise network, especially large ones, protecting them from the outside world. These systems are often deployed on a single (aggregate) Internet link and their configurations infrequently change. We note that operators of large networks often tend to minimise changes to their network topology, network policies, or existing hosts, mainly due to high cost of operations (*e.g.,* offline time, error rate, and labor effort) [32]. Therefore, ease of **deployment** is another impacting factor of attack detection solution.

C. Current Solutions for Detection At-Destination

To the best of our knowledge, there exist three broad groups of solutions[2] for detecting distributed network attacks at the destination end: (i) commercial security middle-boxes, (ii) flow-level attack detection methods, and (iii) experimental prototypes based on programmable networks. In what follows, we describe their individual merits and gaps regarding the key requirements articulated in §II-B.

Practical Security Middleboxes: They are mature products in the market, including next-generation firewalls (NGFW) and intrusion detection systems (IDS). Such systems are built as either proprietary appliances from established vendors [15]–[17] or software tools [33]–[35]. They apply attack detection signatures [36], such as thresholds on packet arrival rate for a specified list of hosts or subnets. These middleboxes primarily aim at scalability and deployability [37], [38] for high-throughput enterprise networks. Thus, they seldom incorporate computationally expensive flow-level telemetry during practice, essential for identifying attackers and their malicious flows mixed with benign traffic. As a result, collateral damages (*e.g.,* unintentionally dropping benign packets [39]) are unavoidable during attack mitigation [22], [29].

Statistical Inference from Flow-level Telemetry: Researchers develop statistical and/or deterministic methods to effectively detect distributed attacks from fine-grained flow-level network telemetry. For example, graph structures that profile network flows exchanged between hosts are widely used in the literature [18]–[20]. Obtaining real-time visibility into every traffic flow between hosts enables operators to precisely detect distributed attackers and malicious flows. However, due to the high computing overheads [40] of flow-level telemetry, such methods can only be employed for relatively low-throughput networks [18] or limited types of traffic such as DNS [41] and HTTP [12] protocols.

Systems leveraging Programmable Networks: Programmable networks (*e.g.,* SDN, NFV, P4) make flexible and elastic traffic management possible by dynamically reconfiguring network functions at run-time. Researchers have

[2]This paper primarily focuses on "network-level" security systems (*e.g.,* border firewalls at the edge of enterprise networks) instead of software tools ("host-level") installed on the individual servers and terminal devices.

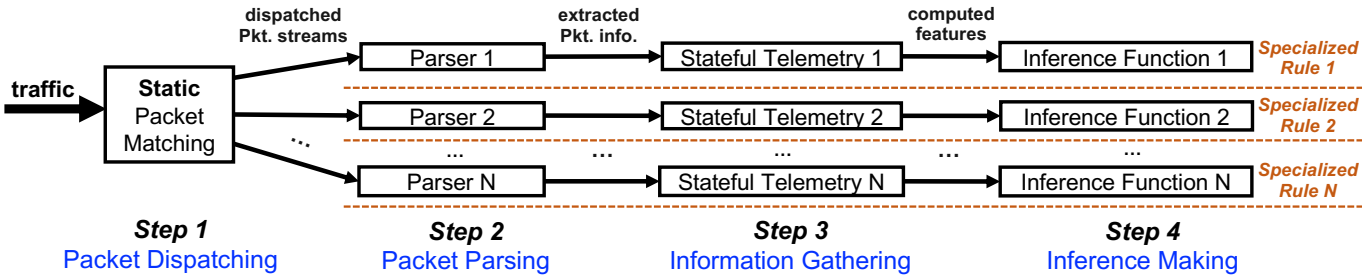


Figure 2: Traffic processing pipelines, responsible for processing received packet streams, in legacy solutions.

developed prototypes leveraging these techniques to achieve security and attack detection objectives [21]–[23], [42]–[44]. These works aimed to address the limitation of proprietary security systems with static detection capability by dynamic and reactive control that programmable networks offer. They inspired the design of our PEDDA, which primarily aims to balance the effectiveness against the practicality of detecting distributed attacks for large enterprise networks.

III. Modelling Complexity of Legacy At-Destination Solutions

In this section, to motivate our design, we analyse legacy detection solutions to highlight computing bottlenecks, hindering them from being effective in attack detection and practical in operation. These highlights and insights motivate our system design in §IV. Specifically, we first model the traffic processing pipeline of legacy at-destination systems (in §III-A), followed by a mathematical formulation of their per-packet CPU consumption (in §III-B) to identify performance bottlenecks that could be optimised in our design.

A. Modelling Traffic Processing Pipelines

Traffic processing of existing attack detection solutions can be modelled as a collection of independent pipelines shown in Fig. 2. Each pipeline is responsible for a specific detection task (*e.g.,* a firewall policy configured by network operators) and consists of four steps to process packet streams, including Dispatching, Parsing, Gathering, and Inference.

1) Four Traffic Processing Steps: Dispatching is the first step that directs packets (based on their metadata) towards a pipeline for further processing. The second step (*i.e.,* Parsing) extracts specific information records (*e.g.,* source and destination IP addresses, transport-layer port number, and protocol) from each packet forwarded to the corresponding pipeline. We note that each parser has its own programmed (and fixed) functionality, such as extracting information from IP header, UDP/TCP header, or certain application-layer headers, thus, in practice, one matching rule may require multiple parsers. In the third step (*i.e.,* Gathering), those extracted records will form the stateful network telemetry each pipeline needs for an intended detection task. In the final step, a corresponding inference is made based on attributes computed from the specific network telemetry of each pipeline.

2) Walkthrough Examples: Let us analyse a couple of walkthrough examples that help us better understand the detection pipelines. Assuming an enterprise network operator plans to protect their main website server (with the IP address `a.b.c.d`) against TCP-SYN DDoS attacks.

A practical detection approach: The network operator may configure a policy on their enterprise border firewall [15], [16], [39] to raise alarms when the target IP address `a.b.c.d` receives more than N TCP-SYN packets during a time interval of T seconds [45]. After installing this rule, all packets received by the firewall will be inspected (*i.e.,* step 1 in Fig. 2) for a SYN flag in their transport-layer header and the destination IPs address of `a.b.c.d`. The matched packet streams are sent to the parser (*i.e.,* step 2) for computing packet count as required for the network telemetry. The telemetry (*i.e.,* step 3) maintains a total number of packets destined to `a.b.c.d` during the current time interval. A corresponding inference function (*i.e.,* step 4) checks packet count (exported by telemetry data every T seconds) and raises alarms if the current value is greater than N.

An effective detection approach: If this task is performed by a flow-level detection method [18]–[20], the inference results can be more precise but demand more complex and expensive processing. In the parsing step, instead of only tracking "aggregate" packet count for the stateful telemetry, a collection of metadata fields, including the transport-layer protocol, TCP flags, IP addresses and source/destination port numbers) are extracted on a "per-flow" basis. In step 3, the extracted information is used to update flow graphs (a relatively complex telemetry) capturing communications between the target IP `a.b.c.d` and external hosts. We note that an operator may set up the granularity of the monitored IP `a.b.c.d` to an aggregated subnet-level (*i.e.,* adding more bits in subnet masks) to reduce the number of monitored entries such as in [41]. In step 4, an external host will be labeled as an attacker if its flow(s) profile is found malicious (*e.g.,* sending an excessive number of flows, most of which are TCP-SYN packets). Although this method can identify individual sources and flows in volumetric (distributed) attacks, as will be discussed next, per-packet costs of computing for such an effective method is too high to be employed for high-throughput enterprise networks.

B. Formulating Per-Packet CPU Consumption

We now mathematically formulate the traffic processing cost, namely CPU consumption per individual packets, which

was identified by prior work like *NitroSketch* [46] as the most critical metric impacting the performance of hardware-based middle-box appliances. Specifically, we model the CPU time spent to process an arriving packet across a four-stage pipeline to quantify its computational complexity (our primary objective). Note that our cost model is agnostic to statistical attributes of traffic. Later in §V (prototype evaluation), we will incorporate representative traffic features such as the number of concurrent flows and packet rate to build simple thresholding models for detection purposes in our proof-of-concept prototype.

Let us use t to denote the CPU time consumption of a packet processed by the entire pipeline. Intuitively, t can be expressed as the sum of time consumption in each of the four steps, given by Eq. 1.

$$t = t_d + t_p + t_g + t_i \tag{1}$$

where t_d, t_p, t_g, and t_i are the time consumed at packet dispatching, parsing, information gathering, and inference making steps, respectively.

We note that the last step (*i.e.,* inference) is executed periodically (*e.g.,* every 10 seconds in commercial firewalls [39]) for distributed attack detection – not triggered by individual packets. It is often treated as an independent process and placed outside the packet processing pipeline. Therefore, we omit t_i from our analysis and focus on t_d, t_p, t_g.

1) Dispatching: Let us start with t_d, the time consumed by each packet in the first step. The detection system has its network interfaces that receive packet streams. An arriving packet is checked against a set of matching rules (or policies) to decide which parser(s) should process it. We assume that there are n_d packet matching rules and n_p parsers in the next step. The best case (*i.e.,* shortest time consumed) $t_{d,min}$ is realised when a packet is matched by the first policy and sent only to one parser in the next step. The worst case $t_{d,max}$, on the other hand, occurs when the packet is checked against all n_d policies and sent to all n_p parsers. Let us denote the time required to: (a) check a packet against a policy by $k_{d,c}$, and (b) direct the packet to a parser by $k_{d,p}$. Therefore, the best and worst cases can be stated as follows in Eq. 2.

$$\begin{aligned} t_{d,min} &= k_{d,c} + k_{d,p} \\ t_{d,max} &= k_{d,c} n_d + k_{d,p} n_p \end{aligned} \tag{2}$$

Assuming $p_{d,i}$ is the hit probability of the i^{th} matching policy, and $p_{p,j}$ is the probability of a packet sent to the j^{th} parser, the average time consumption $t_{d,avg}$ for each arrived packet in this step can be formulated as Eq. 3 below.

$$t_{d,avg} = k_{d,c} \sum_{i=1}^{n_d} p_{d,i} + k_{d,p} \sum_{j=1}^{n_p} p_{p,j} \tag{3}$$

2) Parsing: In this step, parsers extract the required packet metadata (*e.g.,* network layer, transport layer, and application layer) [47]. A simple parser may only processes the network layer, while an expensive one may need to extract data from all header layers of the packet. Therefore, the CPU time consumption for a packet in this step t_p depends on the number of parsers involved (up to n_p) and the number of packet layers to be processed by each parser. Assuming that the i^{th} packet-parser extracts data from a total of $n_{l,i}$ header layers and the time consumption to process each layer is k_l, the total time consumption by the i^{th} parser is expressed as $n_{l,i}k_l$. The best case $t_{p,min}$ is achieved when only the simplest parser is used, while the worst case $t_{p,max}$ occurs when all parsers are applied to a packet. The two cases can be formulated as follows by Eq. 4.

$$\begin{aligned} t_{p,min} &= k_l \min_{i=1}^{n_p}(n_{l,i}) \\ t_{p,max} &= k_l \sum_{i=1}^{n_p}(n_{l,i}) \end{aligned} \tag{4}$$

The average time consumption $t_{p,avg}$ for each packet is obtained by weighting each parser by its hit probabilities ($p_{p,i}$ for the i^{th} parser), which can be mathematically stated below by Eq. 5.

$$t_{p,avg} = k_l \sum_{i=1}^{n_p}(p_{p,i} n_{l,i}) \tag{5}$$

3) Gathering: In this step, stateful network telemetry of each detection task may get updated by packet information obtained from parsers. Telemetry is stored in the data structure of each monitored entity (*i.e.,* host or flow). The data structure can be a simple list of key (*e.g.,* target IP addresses) and value (*e.g.,* packet counts) pairs, or a complex attributed graph [20] tracking statistics of individual flows [48]. Therefore, the CPU time needed for updating a telemetry record depends on its structure type and current size. We use n_g and $N_{g,i}$ to denote the total number of telemetry in this step and current size of the i^{th} telemetry, respectively. The CPU consumption of updating the i^{th} telemetry can be expressed as a fixed function $\theta_i(N_{g,i})$ of time complexity[3]. The best case $t_{g,min}$ is realised when only the simplest telemetry is updated by a processed packet, while the worst case $t_{g,max}$ occurs when all n_g data structures are updated, as given by Eq. 6 below.

$$\begin{aligned} t_{g,min} &= \min_{i=1}^{n_g}(\theta_i(N_{g,i})) \\ t_{g,max} &= \sum_{i=1}^{n_g}(\theta_i(N_{g,i})) \end{aligned} \tag{6}$$

The average time consumption $t_{g,avg}$ by an arrived packet in this step is obtained by weighting all data structures by their probability and expressed by Eq. 7.

$$t_{g,avg} = \sum_{i=1}^{n_g}(p_{t,i}\theta_i(N_{g,i})); \tag{7}$$

where $p_{t,i}$ is the hit probability of the i^{th} telemetry.

4) The entire pipeline: We now summarise the best (t_{min}), worst (t_{max}), and average (t_{avg}) CPU time consumption per packet by aggregating individual components (discussed above) by Equations 8, 9, and 10 below.

$$t_{min} = k_{d,c} + k_{d,p} + k_l \min_{j=1}^{n_p}(n_{l,j}) + \min_{z=1}^{n_g}(\theta_z(N_{g,z})) \tag{8}$$

[3]For instance, if the i^{th} telemetry is stored in a binary search tree, then the complexity of average and worst-case estimate of $\theta_i(N_{g,i})$ will be $O(log(N_{g,i}))$ and $O(N_{g,i})$, respectively.

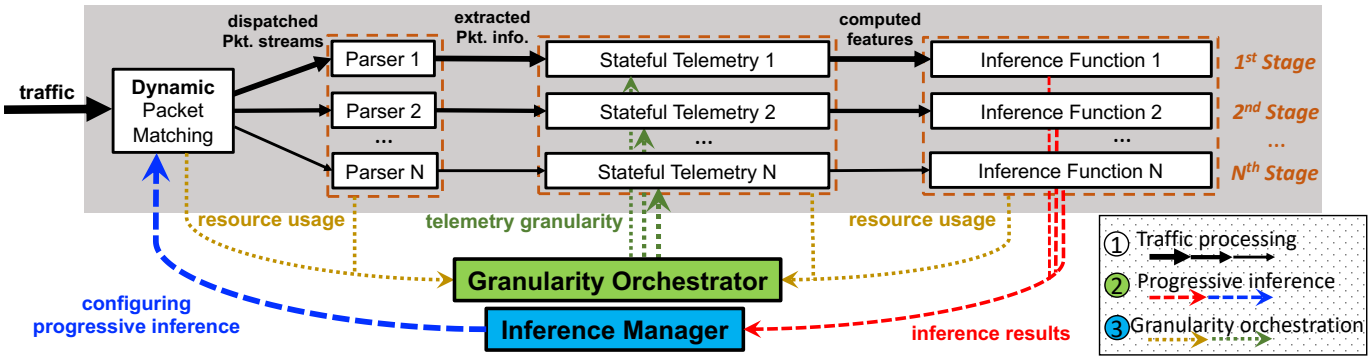


Figure 3: System architecture of PEDDA, our progressive multi-stage inference method.

$$t_{max} = k_{d,c}n_d + k_{d,p}n_p + k_l \sum_{j=1}^{n_p}(n_{l,j}) + \sum_{z=1}^{n_g}(\theta_z(N_{g,z})) \quad (9)$$

$$t_{avg} = k_{d,c}\sum_{i=1}^{n_d}(p_{d,i}) + \sum_{j=1}^{n_p}((k_{d,p} + k_l n_{l,j})p_{p,j}) + \sum_{z=1}^{n_g}(p_{t,z}\theta_z(N_{g,z})) \quad (10)$$

A detection system reaches its maximum scalability if all of its processed packets are mapped to the best case (*i.e.,* t_{min}), while it becomes impractical if most of packets have their processing time as the worst case (*i.e.,* t_{max}).

Summary: Let us revisit the three broad methods (*i.e.,* practical solutions, effective solutions, and PEDDA) we consider in this paper. We now compare their average performance metric (t_{avg}), which is essentially affected by the hit probability of processing steps discussed above.

With practical solutions, a hardware middlebox or appliance is employed to collect relatively lightweight telemetries (often header focused) from a few intended subnets and/or critical IP addresses (those that contribute to the majority of the traffic of interest). We note that updating stateful telemetries (the Gathering step, which is modelled by the third term in Eq. 10) is the main contributor to t_{avg}. Administrators often statically configure their security appliance with computationally inexpensive policies for practical reasons applied to most of their network segments. This means $p_{t,z}$ for the simplest telemetry becomes close to 1, while that of other telemetries (more expensive) tends to 0. This will lead the term $\sum_{z=1}^{n_g}(p_{t,z}\theta_z(N_{g,z}))$ in Eq. 10 tends to $\min_{z=1}^{n_g}(\theta_z(N_{g,z}))$ in Eq. 8. A similar argument can be made for other terms (*i.e.,* packet dispatching and parsing) in Eq. 10, whereby lighter options are often chosen (by administrators) by essentially configuring probabilities $p_{d,i}$ and $p_{p,j}$ for a practical solution – leading t_{avg} to approach t_{min}.

For an effective solution, on the other hand, all supported policies and corresponding telemetries need to be activated for the entire address space (as opposed to certain subnets and/or selected hosts) of the enterprise network. This entails an arriving packet to be processed by a number of parsers, meaning all the probabilities $p_{d,i}$, $p_{p,j}$, and $p_{t,z}$ in Eq. 10 will need to be configured to values close to 1 that lead t_{avg} to approach t_{max} for effective detection. We will see in evaluation results (§V-C) from our prototype applied to real enterprise traffic that the effective detection will result in a t_{avg}, which is 20 times more than that for a practical solution.

Our PEDDA, as opposed to existing static solutions, operates dynamically and hence performs better at scale given constrained computing resources. PEDDA offers this superior performance by dynamically adjusting pipeline modules and their respective hit probabilities. As a result, t_{avg} navigates judiciously between t_{min} and t_{max}, subject to dynamics of inference and available computing resources. We make the following objectives that drive the design of our PEDDA system in §IV: (i) minimising the number of active parallel modules for a received packet in each step (*i.e.,* reducing n_d, n_p, and n_g in Eq. 10); (ii) reducing hit probabilities (*i.e.,* $p_{d,i}$, $p_{p,j}$, and $p_{t,z}$) for expensive modules; and, (iii) selecting stateful telemetry with lighter time complexity $\theta_z()$ in Eq. 10, and maintaining a small number of entries $N_{g,z}$ in the high-cost data structures to reduce the time consumption introduced by the Gathering module (step 3).

Given the above considerations, a detection system is suggested to have complex data structures for effective detection, which are only used to process the necessary fraction of traffic. At the same time, most packet streams are expected to be handled by light modules so that the hit probability and complexity of expensive modules will be significantly reduced for practical uses. Next, we discuss our design of a multi-stage inference architecture based on this idea that progressively detects distributed attacks, whereby only a small but necessary fraction of packet streams are processed by high-cost modules, and low-cost processes handle most traffic.

IV. The Progressive Multi-stage Inference Architecture

Motivated by insights from the previous section, we present our PEDDA system (§IV-A) that employs multiple low-cost and high-cost inference stages to progressively detect a distributed attack. The majority of traffic is proactively processed by low-cost stages, while only a small fraction of traffic (when necessary) will be reactively processed by high-cost stages. High-cost stages are dynamically employed by controls of programmable networks (§IV-B). Depending upon the mix of traffic analysed, high-cost stages may still get overwhelmed if the finest granular telemetry (IP-level) is maintained during the progressive detection. To overcome this challenge, we design an orchestrator that dynamically selects the granularity

(IP-level or certain subnet-level) of each stage by solving a run-time optimisation problem subject to computing resources available at run-time (§IV-C).

A. System Architecture of PEDDA

We now describe the architecture of our progressive multi-stage inference system, including design rationale and choices, schematic of the architecture, and workflows.

1) Rationale and Choices: As discussed earlier in §I and shown in Fig. 1(a), effective detection of a distributed attack can only be achieved by comprehensively monitoring the behavior of potential victims (internal hosts) and potential attackers (external hosts) while considering all network flows, which is practically challenging as the number of external hosts can be very large and the set of flows is unbounded. Therefore, legacy solutions either focus only on victims for practicality at scale or attempt to detect distributed sources and flows that become prohibitively expensive and do not scale cost-effectively. We note that effective detection can be better balanced against practicality at scale by employing multiple inference stages (each activated progressively). Maintaining complex states and telemetry data can be dynamically managed for a certain fraction of traffic analysed at each stage. An early-stage inference monitors a simple aspect of the network activity (*e.g.,* focusing on internal hosts in Fig. 1(a)) via low-cost telemetry, while following stages (with higher-cost telemetry) processes only a manageable fraction of traffic partitioned based on the inference results from the prior stage(s). This process is progressively executed (from high-cost to low-cost stages) till an attack is fully detected by all aspects (*i.e.,* victims, sources, and flows).

Following what was discussed above, three key design choices are made. First, we use multiple inference stages, each offers a specialisation (*e.g.,* victims, sources, or flows) in attack detection. They depend on each other and come in logical orders. Second, an inference stage only processes packet streams partitioned by its prior stage. Therefore, the first stage processes all received packet streams by default, whereas stages with higher costs analyse only a selected (configurable) fraction of traffic determined by their prior cohorts. Third, each stage is expected to make an inference at the finest granularity. However, subject to available computing resources, the granularity of each stage may get reduced (by aggregating into certain subnet levels) to guarantee operational robustness.

Our three key design choices collectively guarantee the practicality and detection capability of PEDDA, particularly at the scale of tens of Gbps traffic rates with millions of concurrent flows (demonstrated in §V-C2).

2) Schematic and Workflows: Driven by our design choices, the PEDDA architecture has three key modules, each highlighted by a colored box in Fig. 3. As shown by the grey region, PEDDA contains **collaborative inference stages** (a pipeline of packet matching, parser, telemetry, and inference functions) logically ordered with costs from low to high.

The fundamental difference between our collaborative inference stages (shown in Fig. 3) and the legacy traffic processing pipelines (shown in Fig. 2) lies in how packets are processed. The packet matching module is often statically configured (by the network administrator) in legacy solutions pre-deployment. In contrast, the packet matching in PEDDA is dynamically configured at runtime based on factors like network traffic, attack type, and computing resources. Also, note that each processing pipeline in Fig. 2 operates independently for a special rule, whereas PEDDA pipelines in Fig. 3 collaboratively serve detection objectives. An **inference manager** (blue box) is designed to collect detection results from all stages and dynamically instructs them to expand or reduce the scope/fraction of traffic they process. A **granularity orchestrator** (green box) takes run-time statistics of system resources as input and gives the granularity of each stage as output. Its specifications and mechanisms will be discussed in §IV-C. Fig. 3 illustrates three workflows, namely packet processing (highlighted by solid arrows), progressive inference (highlighted by dashed arrows), and granularity orchestration (highlighted by dotted arrows).

Progressive traffic processing and inference in real-time: In real-time operation, the packet matching module receives traffic and proactively forwards all packet streams to the first stage, which employs the simplest (low-cost) telemetry. For example, let us assume that the first stage detects only victims by tracking the packet count of each enterprise host. Once a victim is identified, the first stage will notify the inference manager. The second stage will then be instructed to start processing traffic associated with that specific victim for a further inference (*e.g.,* detect distributed attackers). This procedure is progressively executed till all aspects (*e.g.,* victims, attackers, and flows) of a distributed attack are determined. With this approach, only the necessary portion of the network traffic is dynamically and selectively processed by high-cost stages, effectively detecting attacks while ensuring operational practicality. We will discuss in §V-C2 that our approach may still not fully capture all aspects (*e.g.,* missing some attack sources and malicious flows) of certain attacks, especially those that are short-lived and complete rapidly without giving the detection system sufficient time for progressive inference.

Orchestrating granularity of real-time telemetry: We note that network traffic, in general, and especially during DDoS attack events, is dynamic and relatively unpredictable. Therefore, our inference system may face challenges in processing a large volume of data streams in real-time, given the fixed amount of measurement and computing resources. The granularity orchestrator is developed to address this operational challenge. It is continuously fed by run-time load statistics and system utilisation metrics (traffic rate, CPU consumption, memory usage, and the number of monitored entities) across all pipelines. The orchestrator determines how to adjust each stage's granularity (*i.e.,* fine-grained IP levels versus coarse-grained subnet levels) by solving an optimisation problem (in §IV-C). This adjustment is made periodically at a configurable frequency (e.g., near real-time). Following an optimal solution is obtained, instructions are dynamically sent to the inference manager to configure each processing stage at run-time.

B. Reactive Traffic Control via Programmable Networks

The key enabler of our progressive inference method is reactive control of packet forwarding and processing, naturally matched with the paradigm of programmable networking and technology options like NFV, SDN, or P4. In what follows, we discuss three possible choices for reactive controls.

First, our architecture can be fully implemented with virtual network functions (VNF). The packet matching module and packet parser bank in Fig. 3 can be operated within a software switch (*e.g.,* vSwtich) interacting with other components configured as modular services running on general-purpose servers. This choice sounds practical to small enterprise operators who are tasked to manage low traffic rates and may not have hardware programmable switches (*e.g.,* OpenFlow or P4 switch) available in operation. However, fully software-based packet processing pipelines do not seem practical in handling high-rate traffic (*e.g.,* tens of Gbps) of large enterprise networks [23]. Second, one may consider programmable control-plane hardware switches (*e.g.,* OpenFlow) as its packet processing modules, which can handle traffic streams at line rates. Other components, including maintaining telemetry states, can be done in software modules and micro-services. Third, one may choose to embed stateful telemetry into programmable data-plane switches (*e.g.,* P4) for reasons like improving responsiveness and/or reducing loads from generic servers. However, this technology option (while at its early stages of adoption by industry) comes with a key limitation, namely, the voluminosity of its code [49], becoming challenging to develop, debug, and maintain [50].

In operation, each of the choices discussed above will come with certain resource limitations either for software modules running on the generic servers (*i.e.,* available CPU and RAM) or hardware switch resources (*e.g.,* DRAM in OpenFlow switches, SRAM in P4 switches). Exhausting available resources under extreme conditions (*e.g.,* when the most expensive stage processes the majority of packets) can lead to system failure. In what follows, we discuss how the operational robustness of our method is dynamically maintained by optimally tuning the granularity of stages with respect to the available system resources.

C. Optimal Orchestration of Granularity

Intuitively, our PEDDA architecture aims to perform an effective detection while satisfying practicality requirements via progressive inference across multiple stages (described in §IV-A1). However, inference stages, especially those that demand expensive telemetry, may get overwhelmed by heavy traffic load. For example, suppose most of the network traffic is found suspicious and sourced from well-distributed sources with a massive number of concurrent flows. In that case, inferring attackers and their malicious flows at the finest granularity (*i.e.,* host level) can be challenging depending upon available resources and the complexity/intensity of the attack.

The orchestrator module in PEDDA (shown in Fig. 3) is responsible for dynamically choosing the granularity of stages. Ideally, all stages would aim to infer by their finest granularity (*i.e.,* subnet mask "`/32`" for IPv4 or "`/128`" for IPv6), where statistics of network activities are tracked and maintained at the host level. Given total computing resources, network operators may be willing to sacrifice (de-prioritise) the granularity of certain stages that they deem less important. Priority can be specified by configuring a numerical weight (say, between 1 and 100) for each stage, where 1 indicates the lowest priority and 100 indicates the highest priority. For example, suppose an operator wishes to detect victims at the host level (setting the priority of the corresponding stage equal to 100) but is happy to identify external attackers at a subnet level (setting the priority of the corresponding stage equal to 1). The impact of priorities is expected to be relative.

At run-time, with the pre-configured priorities, the orchestrator periodically adjusts the granularity of each stage based on available computing resources. For example, an increase in memory utilisation may lead the orchestrator to replace the granularity "`/32`" in IPv4 scheme with something coarse-grained (say, "`/28`") for stages with low priorities (reducing the granularity by four bits), sustaining the inference but at a lower granularity. Note that the lowest granularity "`/0`" yields a fixed cost regardless of the number of hosts (internal/external) involved since it requires monitoring of one entry (*i.e.,* `0.0.0.0/0`). We note that subnet-based aggregation is often feasible where enterprise subnets are relatively utilised by internal active hosts. Also, distributed attackers often source traffic from compromised (sub)networks (botnet devices) [41] instead of sparsely distributed across the Internet address space.

In order to develop a systematic logic for the orchestrator module, let us mathematically formulate the above logic as a constrained optimisation problem as follows.

Objective function: Our primary objective is to maximise the granularity of individual stages with respect to their priorities configured by the operator. Suppose there are N_s stages, where the i^{th} stage has priority W_i (given as input) that infers at the granularity of subnet mask S_i. Our objective function can be formally stated by Eq. 11.

$$\max \sum_{i=1}^{N_s} (W_i \cdot S_i) \tag{11}$$

where, weighted granularities are maximised.

Constraints: The constraints are threefold: range of granularity, switch memory, and server computing resources.

Granularity: Considering the IPv4 addressing scheme, subnet masks can (theoretically) take integer values between 0 to 32, as given by Eq. 12 below.

$$0 \leq S_i \leq 32 : \forall i \in [\![1, N_s]\!] \tag{12}$$

The network operator may wish to configure some custom ranges for each stage.

We note that though this paper focuses on the IPv4 addressing scheme, one can consider our optimisation for IPv6 addressing by slightly changing the upper bound of S_i to 128. This would expand the search space by a factor of four (128 bits IPV6 addresses versus 32 bits IPv4 addresses). That said, an expanded search space may not necessarily elongate the

convergence time (in sequential search) with the same factor. One can choose to improve the search time by employing non-linear search algorithms. We leave it for future work to quantify and manage the impact of IPv6 addressing on computing resources.

Switch: The packet matching module in Fig. 3 that receives packet streams and dispatches them towards inference stages sees its flow rules updated at run-time by reactive configurations. This module is realised by virtual or hardware programmable switches with limited table sizes (memory).

A programmable control-plane (*e.g.,* OpenFlow) switch has a capacity of accommodating F flow rules along with an upper bound limit on the changing rate of δF rules. The operator may further introduce additional constraints to each inference stage denoted by F_i and δF_i. In our prototype implementation (will be discussed in §V), we will use a commercial OpenFlow switch (*i.e.,* NoviSwitch 2122 [51]) with F equal to 6 million wildcard matches and δF equal to 40K per second. Also, we will partition the switch memory tables, each dedicated to a specific inference stage – stages will monitor their target flows available in their respective table. One may choose a different strategy for allocating resources to various stages.

We use H_i to denote the number of hosts (IPv4 addresses) whose network activity is required to be monitored by the i^{th} stage operating at granularity S_i. We estimate the number of flow rules that can be available to the stage as $\frac{H_i}{\gamma(S_i)}$, where $\gamma()$ is a scaling factor indicating the sparsity of IP addresses at a given subnet. Theoretically speaking, $\gamma(S_i)$ can take a value within the range $[1, 2^{32-S_i}]$. A scaling factor 2^{32-S_i} is when subnet S_i is fully utilised and hence the aggregation well reduces the number of monitored entries (best case). In contrast, a scaling factor 1 is when the aggregation does not reduce the number of monitored entries (worst case) since IP addresses are sparsely distributed (unlikely in typical enterprise settings and distributed attacks). Note that this factor is configured by the operator based on empirical insights into their managed network. Later in prototype implementation for our university network (§V), we will use empirical values such as $\gamma(31) = 1.4$ or $\gamma(16) = 2^{16}$.

Therefore, F_i and δF_i will impose two constraints on the number of entries monitored by each stage, as formally stated by Eq. 13.

$$\begin{aligned} \frac{H_i}{\gamma(S_i)} < F_i : \forall i \in [\![1, N_s]\!]; \\ \delta\frac{H_i}{\gamma(S_i)} < \delta F_i : \forall i \in [\![1, N_s]\!] \end{aligned} \tag{13}$$

Suppose a programmable data-plane switch empowered by the P4 technology (*i.e.,* Intel Tofino 2 [52]) is used for the packet matching module. In that case, two additional constraints are introduced for each stage, including maximum SRAM R_i and stateful ALU A_i allocated to the i^{th} stage. We use r_i and a_i to denote the utilisation of SRAM and ALU per each monitored entity. The two additional constraints are expressed by Eq. 14.

$$\begin{aligned} r_i \cdot \frac{H_i}{\gamma(S_i)} < R_i : \forall i \in [\![1, N_s]\!]; \\ a_i \cdot \frac{H_i}{\gamma(S_i)} < A_i : \forall i \in [\![1, N_s]\!]; \end{aligned} \tag{14}$$

Server: The performance of modules running on commodity servers is constrained by total CPU and memory units allocated to the system, represented by C and M. We use $C_{parser,i}$, $C_{telemetry,i}$, $C_{inference,i}$ to denote the units of CPU available to the parser, telemetry, and inference modules at the i^{th} stage, respectively.

For the i^{th} stage, we assume that packets arrive at rate λ_i, estimated based on the latest measurements. In our implementation (§V), we use the average packet rate during the last minute (our optimisation runs every minute). One may choose to estimate λ_i over a more extended period (say, daily).

The server takes (on average) $c_{p,i}$ and $c_{f,i}$ of the available CPU units to process a packet by parser and inference modules, respectively[4]. Since the telemetry module is stateful, its CPU consumption is estimated as $\theta_{t,i}(n)$ units to process a packet when there are n monitored entities in the data structure. For our prototype (§V), we use $\theta_{t,i}(n) = k \cdot n$, estimating the worst-case time complexity of our telemetry module, which is built based on a hash table[5]. Note that k is a constant coefficient. We empirically measured k by replaying recorded packets onto our server and compute the CPU time consumed per packet in updating the respective entry in the telemetry structure.

Note that unlike parser and telemetry modules that operate on a per-packet basis, the inference module needs to activate periodically, at frequency f_i defined by the network administrator. Therefore, the CPU time consumption of each packet across the three modules can be expressed as $\lambda_i c_{p,i}$, $\theta_{t,i}\left(\frac{H_i}{\gamma(S_i)}\right)\lambda_i$, and $c_{f,i}\left(\frac{H_i}{\gamma(S_i)}\right)f_i$, respectively. Constraints on the units of CPU available to packet parser, telemetry, and inference are expressed by equations 15, 16, and 17 below.

$$\lambda_i c_{p,i} < C_{p,i} : \forall i \in [\![1, N_s]\!] \tag{15}$$

$$\theta_{t,i}\left(\frac{H_i}{\gamma(S_i)}\right)\lambda_i < C_{t,i} : \forall i \in [\![1, N_s]\!] \tag{16}$$

$$c_{f,i}\left(\frac{H_i}{\gamma(S_i)}\right)f_i < C_{f,i} : \forall i \in [\![1, N_s]\!] \tag{17}$$

Now we look at the memory consumption. Both parser and inference modules are stateless functions that consume a roughly constant and negligible amount of server memory. For the stateful telemetry module, we use $m_{t,i}$ to denote the run-time memory usage per monitored entity (host or subnet) of the i^{th} stage. This static coefficient could be benchmarked in the same way we discussed above for coefficients of CPU

[4]We measured these two constant parameters by replaying recorded packets onto our server.

[5]Other data structures like binary search tree would come with their own time complexity function.

Table I: A summary of our optimisation parameters.

Para.	Description	Type	Value in our prototype
W_i	stage priority	input configs	$W_1 = 100$ $W_2 = 10$ $W_3 = 1$
N_s	# stages	input configs	3
f_i	infer. frequency	input configs	0.1
F_i	max. # entries	system constraint	$F_1 = 3e4$ $F_2 = 3e4$ $F_2 = 5e4$
δF_i	max. # δ entries	system constraint	$\delta F_1 = 1e3$ $\delta F_2 = 1e3$ $\delta F_3 = 1e3$
R_i	max. SRAM	system constraint	admin-defined
A_i	max. ALU	system constraint	admin-defined
$C_{p,i}$	max. CPU	system constraint	admin-defined
$C_{t,i}$	max. CPU	system constraint	admin-defined
$C_{f,i}$	max. CPU	system constraint	admin-defined
$M_{t,i}$	max. RAM	system constraint	admin-defined
H_i	# hosts	network stats	runtime measure
λ_i	packet rate	network stats	runtime measure
$\gamma()$	address sparsity	static coefficient	[1.4, 1.8, ..., 2^{16}]
$c_{p,i}$	CPU usage	static coefficient	$c_{p,1} = 2e(-8)$ $c_{p,2} = 2e(-8)$ $c_{p,3} = 2e(-8)$
$\theta_{t,i}$	CPU usage	static coefficient	$\theta_{t,1} = 1e(-4)$ $\theta_{t,2} = 1e(-4)$ $\theta_{t,3} = 2e(-4)$
$c_{f,i}$	CPU usage	static coefficient	$c_{f,1} = 2e(-8)$ $c_{f,2} = 2e(-7)$ $c_{f,3} = 2e(-6)$
$m_{t,i}$	RAM usage	static coefficient	$m_{t,1} = 50$ $m_{t,2} = 1e3$ $m_{t,3} = 1e4$
S_i	subnet mask	optimal output	algorithm-generated

usage. Therefore, the constraint for the memory consumption of the telemetry module can be expressed as Eq. 18.

$$m_{t,i}\left(\frac{H_i}{\gamma(S_i)}\right) < M_{t,i} : \forall i \in [\![1, N_s]\!] \tag{18}$$

Table I summarises all parameters we use in our optimisation. Three parameters (W_i, N_s, f_i) are configurations provided as input by the network administrator. There are eight parameters that specify constraints of switch (F_i, δF_i, R_i, A_i) and server ($C_{p,i}$, $C_{t,i}$, $C_{f,i}$, $M_{t,i}$). Two parameters (H_i, λ_i), pertinent to network statistics, are dynamically measured at runtime. We have static coefficients, namely $\gamma()$, $c_{p,i}$, $\theta_{t,i}$, $c_{f,i}$, $m_{t,i}$) that are obtained via bench-marking with the target environment and/or compute resources. Finally, S_i will be computed as optimal output. We will demonstrate in §V-C2 how the run-time optimisation orchestrates the granularity of each stage to achieve system robustness under various resource constraints.

V. Prototype Implementation and Evaluation

This section presents and evaluates a practical prototype of our *PEDDA* architecture designed in §IV. We demonstrate how PEDDA is ready for deployment in a large enterprise network for real-time DDoS detection. Inspired by the insights from an empirical traffic analysis of a large representative enterprise (§V-A), we will consider three intuitive inference stages. The prototype implementation details are discussed in §V-B. We will evaluate the efficacy of our method (§V-C) using a one-day worth of enterprise traffic injected by DDoS attack traces with ground-truth labels, demonstrating how our prototype outperforms its counterparts in telemetry visibility, detection effectiveness, and operational robustness.

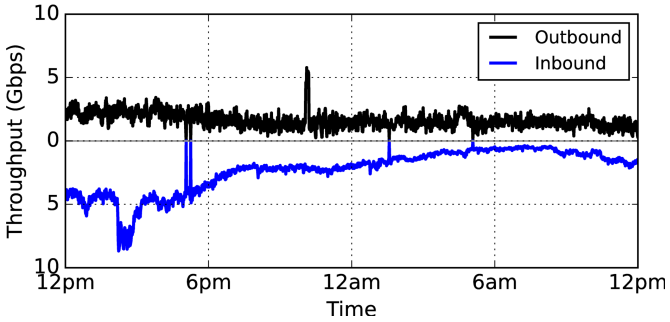


Figure 4: Time-trace of throughput in our enterprise dataset.

A. Three Practical Progressive Inference Stages

Choosing suitable inference stages for enterprise networks is an important task to realise the PEDDA architecture (discussed in §IV) as a practical system. To motivate the selection of our inference stages, let us begin by performing an empirical analysis on traffic traces captured from the network edge of a large enterprise for one day to understand traffic flows and host behavioral profiles.

1) Analysing Traffic from a Representative Enterprise: In our conceptual design (§IV), most of packet streams are expected to be processed by low-cost stages for early inference, while high-cost stages reactively inspect a minority of traffic. To this end, understanding the traffic profile of a typical enterprise is the prerequisite step to choosing a practical and effective approach to progressive inference. In what follows, we draw insights into profiles of traffic flows and host behaviours in an enterprise network that motivate the design of inference stages.

Enterprise dataset: The IT department of our university campus network provisioned a full mirror of its Internet traffic to our data collection system (both inbound and outbound via two separate 10 Gbps fibre links)[6]. We collected a "dataset" on a working day (from 12pm on 31 May 2019 to 12pm on 1 June 2019) with negligible ($< 0.05\%$) drop rates, recording the first 96 bytes of all packets received on mirrored links. Our dataset contains 13.8 billion inbound packets and 21.5 billion outbound packets. The real-time traffic throughput of our dataset is shown in Fig. 4. During busy hours 1-4pm, the inbound link carried about 9 Gbps traffic with a peak packet rate equal to 700K pps, and the outbound link had its throughput as high as about 2 Gbps with 600K pps. For our evaluation experiments in §V-C2, we will replay this dataset

[6]We obtained appropriate ethics clearance (UNSW Human Research Ethics Advisory Panel approval number HC17499) for this study.

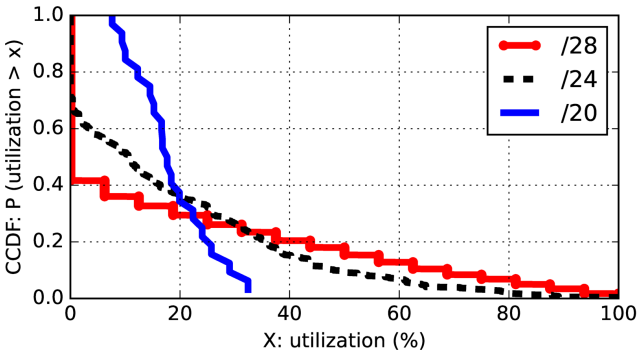


Figure 5: A CDDF plot illustrating how representative internal subnets are utilised in our enterprise dataset.

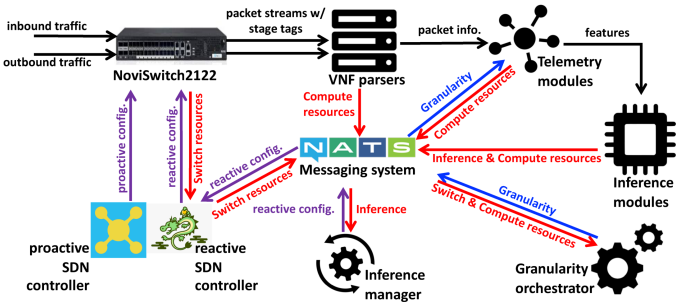


Figure 6: Prototype implementation of PEDDA.

(injected by public DDoS traces with ground-truth labels) onto our detection system.

Network flows: We now analyse the profile of traffic flows crossing the border of our enterprise network. We capture two-sided communications per each flow. The direction of a flow (inbound or outbound) is determined by its first packet. In our dataset, we found 60.8M outbound flows and 559.2M inbound flows. Surprisingly, the majority of (*i.e.,* 73% of outbound and 70% of inbound) flows are one-sided (the other side does not respond). Further investigations revealed that more than a third (36%) of the one-sided flows contain only one packet, while the rest mostly contain repetitive packets. These patterns are found in malicious events like scans [41]. Needless to mention that one-sided flows carry a small fraction of all packets (≈4% of outbound and ≈6% of inbound) in our dataset.

Network hosts: A total of 196K enterprise unique IP addresses appeared in our dataset during the day – this total equals the size of the entire public IPv4 space (*i.e.,* three "`/16`" subnets) of our university network. We first analyse their packet distribution. The majority (92.4%) of those addresses are only found in the destination of inbound packets – no outbound packet originated by those IP addresses. They could be either inactive hosts during the day of our packet capture or completely unassigned IP addresses of the enterprise – probably targets of inbound scans. On the other hand, we find no enterprise IP address that "purely" sends outbound packets without receiving inbound traffic. In other words, internal hosts in our dataset may have one-sided outbound flows but they all have at least one two-sided outbound flow.

Focusing on active hosts (those that send outbound packets), we expect to see a higher variation in the utilisation of smaller subnets (*i.e.,* fine-grained) compared with larger subnets (*i.e.,* coarse-grained). Fig. 5 shows the CCDF of utilisation (active fraction of the subnet block) for three representative subnet sizes, namely "`/20`", "`/24`", and "`/28`" in our enterprise dataset. Unsurprisingly, more than half of "`/28`" subnets (solid red line) are completely idle, about a tenth of them see a utilisation of more than 60%, and certain subnets are fully utilised. This translates to fine-grained subnets being more sparse, demanding specific entries (higher monitoring costs). Reducing the subnet size to "`/24`", the distribution gets slightly smoother – idle portion is reduced (by about 20%) and so does the fraction of highly utilised subnets. It can be seen that the utilisation of "`/20`" subnets is always less than 40% but none of them are idle. This highlights that coarse-grained subnets are less sparse, creating an opportunity to use aggregate entries (reducing monitoring costs).

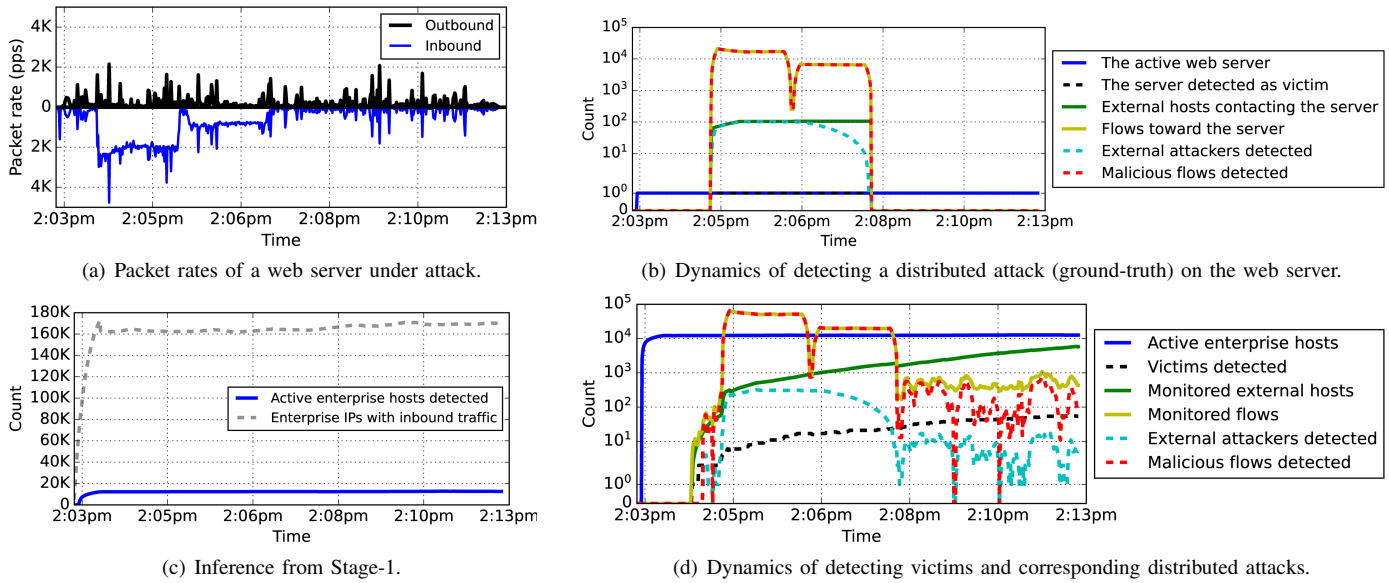
Now, let us analyse the distribution of flows across enterprise hosts. Unsurprisingly, the majority (84.5%) of one-sided inbound flows target unassigned IP addresses or inactive enterprise hosts – they seem to be network scans randomly selecting their targets. Considering one-sided outbound flows, they are all primarily sourced from a negligible fraction (0.4%) of enterprise hosts. We manually investigated those cases and found some exhibit abnormal behaviours like performing port scans on Internet hosts. For example, some of these hosts sweep a relatively large range of port numbers (*e.g.,* more than 2000 TCP ports) on an external host within a short time interval (*e.g.,* 10 seconds).

Focusing on external hosts, we found 751K unique IPv4 addresses were source of inbound packets and of those less than 60% are responded by enterprise internal hosts. Also, we observe that 15K external IP addresses (in our dataset) only received packets from enterprise hosts without sending a response back.

Highlights: We now summarise two key insights obtained from dataset analysis that inspire the design of our practical inference stages. First, unsolicited inbound packets, though they contribute to a small fraction of total traffic, result in a large number of one-sided flows (particularly inbound). Therefore, determining active enterprise hosts (which demand protection against external distributed attacks) is paramount for a detection system since the complexity of expensive telemetry can be significantly reduced. Second, we note that the count of external hosts (750K) and network flows (≈600M) is orders of magnitude larger than that of internal enterprise hosts (≈25K). This means maintaining real-time states for all external hosts and flows is almost impractical. Therefore, for practical reasons, detection of distributed attackers and their malicious flows needs to be progressively achieved by dynamically processing a small fraction of the network traffic.

2) Specifications of Three Inference Stages: Let us now discuss the specifications of three inference stages that collectively detect active enterprise hosts, victims, and distributed attackers with malicious flows.

Stage-1 (active hosts): Our first stage is designed to detect *active enterprise hosts*. The packet matching module sends all outbound packet streams to this stage by default. The parser of this stage extracts source IP addresses of outbound packets,



(a) Packet rates of a web server under attack.

(b) Dynamics of detecting a distributed attack (ground-truth) on the web server.

(c) Inference from Stage-1.

(d) Dynamics of detecting victims and corresponding distributed attacks.

Figure 7: Illustrating how PEDDA detects distributed attacks during a short interval: (a) traffic rates of a web server under a distributed attack, (b) dynamics of detecting a distributed attack on the web server, (c) inference from Stage-1, and (d) dynamics of detecting victims and corresponding distributed attacks.

which are used to update a list of active enterprise IP addresses (or subnets if the granularity is reduced). An IP address will be removed from this telemetry if it has not an outbound packet for a user-defined period. Changes in the list are reported to the inference manager periodically, so that corresponding reactive configurations will be generated for the successive stages.

Stage-2 (victim hosts): Our second stage detects those enterprise hosts that are *victims* of distributed attacks. Reactively instructed by the inference manager, this stage receives inbound and outbound traffic of only active enterprise hosts determined by Stage-1. The telemetry maintained for this stage monitors traffic statistics of each active enterprise host to detect victims from others. In our prototype, inspired by state-of-the-art detection systems [17], we track inbound and outbound packet rates for each monitored entity (IP address or subnet). If the delta of inbound and outbound rates (*i.e.,* $\Delta = |Rate_{in} - Rate_{out}|$) of a monitored entity exceeds a user-defined threshold (500 pps in our implementation), the inference module will report it as a victim[7]. As a result of this detection, the inference manager will instruct the following stage accordingly.

Stage-3 (attack sources and flows): Our third inference stage maintains an expensive telemetry for detecting *distributed attackers* and *malicious flows* in a network attack. This stage only analyses traffic of identified victims. For each packet, the parser of this stage extracts its size, protocol, source and destination IP addresses and port numbers. A streaming graph is used to track the activity of each network flow exchanged between individual (detected) victims and external hosts. Attributes of each external host and flow are computed periodically, so that external sources and malicious flows can be precisely detected at this final stage. In our prototype, we use a threshold on the number of active flows (10 active concurrent flows in our implementation) between an external host and the victim as our detection criteria, which is commonly used by practical security middleboxes [16], [45].

3) Discussion: This paper primarily aimed at setting a foundation to develop practical and effective methods for detecting distributed network attacks. Our work can be improved and built upon in certain ways. First, the inference function could be enhanced by employing more attributes and applying more sophisticated (deterministic and/or machine learning-based) algorithms [41], [53]. Second, we focused on DDoS detection (as an illustrative use-case) to demonstrate the efficacy of our method. Other network attacks like reconnaissance or brute-forcing can be detected by changing the inference criteria. Third, we employed a 3-stage architecture to demonstrate the efficacy of our method when considered a representative loosely-managed enterprise network. One can design a different architecture for further optimisation. Fourth, inference models may require certain tuning to cater for different network settings and traffic characteristics. The above listed improvements and extensions are beyond the scope of this paper.

B. Prototype Implementation

In §IV-B, we discussed technology options, such as pure VNF, OpenFlow, or P4, that can be employed to implement the PEDDA system. In this work, we choose to use an OpenFlow-enabled switch for our prototype. Though OpenFlow capabilities may seem less advanced compared to options like P4, current industry practices find it more intuitive and relatively easier to employ for certain use cases. Also, OpenFlow is currently supported by enterprise-grade switches/routers from top vendors [54], including Cisco [55], Juniper [56], Dell [57], Huawei [58], and HP [59]. It has also been deployed by leading content providers such as Google for managing

[7]One may employ a more sophisticated method to detect victims of distributed attacks.

Table II: A summary of comparative evaluations: our PEDDA versus state-of-the-art solutions.

	Runtime Visibility				Detection Performance			Computing Cost	
	Host	**Victim**	**Attacker**	**Flow**	**Victim**	**Attacker**	**Flow**	**CPU (avg | peak)**	**RAM (avg | peak)**
NGFW	Partial	Partial	None	None	100%	None	None	20% | 42%	immeasurable
IDS	Partial	Partial	None	None	100%	None	None	35% | 62%	21% | 34%
Flow Graph	Complete	Complete	Complete	Complete	100%	95.4%	94.1%	>700% | Full	Full | Full
PEDDA	Complete	Complete	Complete	Complete	100%	93.4%	91.7%	24% | 57%	8% | 11%

WAN [60], Internet peering [61], and datacenter networks [62]. Moreover, the dynamic measurement and control requirements of the PEDDA design are sufficiently fulfilled by programmable control-plane functions offered by OpenFlow. That said, one may improve our implementation by offloading certain telemetry computations from generic servers to a P4 switch, saving costs and/or improving responsiveness.

We realise our PEDDA architecture with the three practical stages by implementing a prototype using an OpenFlow programmable switch and software modules running on a commodity server. Fig. 6 shows functional blocks and their interactions in our prototype. The entire Internet traffic of the enterprise is mirrored to two separate 10 Gbps network interfaces (one for inbound direction and one for outbound direction) of the OpenFlow switch (NoviFlow 2122 [51]).

The programmable switch inserts a specific tag (via the action field of flow entries in the corresponding flow tables) to each arriving packet indicating the stage by which the packet needs to be inferred. Packet parsers (employing the DPDK framework and NFF-Go library), telemetry, and inference modules of our three stages are written in Golang and deployed on a blade server. The server has sixteen 2.10GHz CPUs and 64GB RAM. A publish-subscribe messaging system (NATS) is used to exchange inference results, reactive configurations, available switch and computing resources, and the granularity of each inference stage. The granularity orchestrator (written in Golang) updates the subnet masks of each stage by solving an optimisation problem using the received system statistics, as discussed in §IV-C. The inference manager (written in Python3) publishes run-time flow rules and the granularity of inference stages based on detection results and granularity orchestrating instructions received from the messaging channel. Lastly, our prototype employs two SDN controllers: one is responsible for inserting proactive flow rules (Faucet), and another is responsible for inserting reactive flow rules at run-time (Ryu) to the switch.

C. System Evaluation

In this section, three experiments are conducted. We start by demonstrating how our solution detects distributed attacks on representative hosts by three-stage progressive inference (small-scale evaluation in §V-C1). We then experimentally compare the performance of our method with that of state-of-the-art solutions in terms of telemetry visibility, detection performance, and computing cost when more than 400 hosts become target of attacks during different times of day (large-scale evaluation §V-C2). Finally, we demonstrate how the granularity of stages is optimally adjusted with the utilisation of constrained resources (§V-C3).

1) Small-Scale Detection of Representative Attacks: We demonstrate the dynamics of attack detection by replaying a demo PCAP trace (a subset of our enterprise dataset injected by a short trace of DDoS attacks with ground-truth information) onto our prototype.

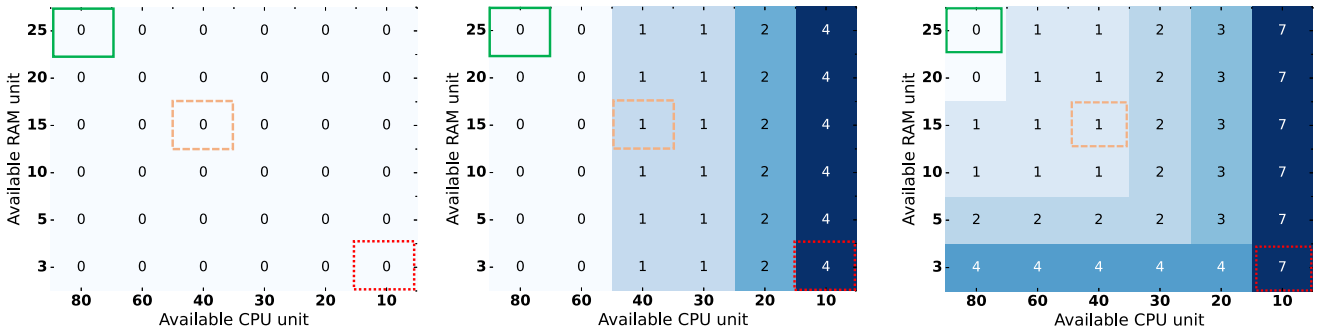
Demo trace: We obtained a public traffic trace from [63] containing 10 minutes' worth of DDoS attacks with ground-truth labels (attack dataset). Next, we modified the destination IPv4 addresses (victims) in the attack dataset to emulate attacks on three representative servers (*i.e.,* a website server, a VPN gateway, and a student portal) in the enterprise dataset (discussed in §V-A1). Lastly, we injected the modified attack trace into a subset (15 minutes' worth) of our enterprise dataset, resulting in our demo trace.

Detection thresholds: As mentioned earlier, we employ a threshold-based method for inference across our three stages. Our threshold values are inspired by the security industry guidelines and best practices [15], [16], [45]. For Stage-1, a detected active enterprise host will get removed if no outbound packet is sent for 10 seconds. Our Stage-2 flags an enterprise host as a victim when the difference of inbound and outbound traffic rates of that host exceeds 500 packets per second. Stage-3 distinguishes malicious external entities and their flows from benign ones that contact enterprise victims. If an external host contacts a victim with more than 10 active flows, the host and its flows will be flagged as malicious.

We note that one may use an extensive set of traffic features, such as statistical measures of packet sizes, contents of certain packets, or temporal activities, to obtain more sophisticated detection models (*e.g.,* machine learning-based). Developing and optimising detection models are beyond the scope of this paper.

Demo evaluation results: We replayed the demo trace onto our prototype. Fig. 7 illustrates how PEDDA progressively detects distributed attacks (victims, attackers, and flows) during a short interval.

As a case study, let us first focus on a web server that is one of the three victims (with ground-truth data). It can be seen in Fig. 7(a) that this server's inbound and outbound packet rates are typically less than 1 Kpps. However, this expected rate is exceeded between 2:04pm and 2:08pm during which we introduced attack traffic of about 10K concurrent flows sourced from 100 distributed sources. Fig. 7(b) shows how the stage make their respective inference. Stage-1 consistently labels (highlighted by solid blue line) this server as an active host for its outbound activity. Stage-2 detects this server as a victim (highlighted by dashed black line) when packets rates exceed a configured threshold due to inbound attacks. Inbound and outbound packet streams of this victim are reactively analysed by Stage-3 (highlighted by solid green and yellow



(a) Stage 1 (active enterprise hosts). (b) Stage 2 (victims). (c) Stage 3 (attack sources and flows).

Figure 8: Optimal adjustment of the inference granularity (distances from the ideal subnet mask) at: (a) stage 1 (active enterprise hosts), (b) stage 2 (victims), and (c) stage 3 (attack sources and flows), during peak network load in our one-day enterprise traffic trace, when 36 scenarios of computing resource constraints are imposed.

lines) to identify external sources and malicious flows. We can observe that the majority of external hosts and flows contacting the victim are flagged as malicious (highlighted by dashed light-green and red lines) – less than 5 of external hosts (look negligible in the plot) are found benign and continue communicating with the server.

Following this initial case study, we analyse the performance of our system across the entire demo trace. Fig. 7(c) illustrates how our Stage-1 detects about 18K active enterprise hosts (highlighted by solid blue line) while close to 170K of IP addresses of our university IP block (highlighted by dashed grey line) are the destination of inbound traffic. Therefore, Stage-2 focuses on traffic of active enterprise hosts aiming for victim detection. The dashed black line in Fig. 7(d) highlights the number of hosts detected as victims by Stage-2. We note that in addition to our three victims (ground-truth), 13 other hosts are detected as victims of distributed attacks. Our manual investigation revealed that they are indeed under (relatively small-scale) distributed attacks. For example, a victim consistently received TCP packets of size 60 to 70 bytes from 32 distributed attackers – 11 of them belong to a subnet"`/24`" subnet, while the other 21 belong to 9 different subnets. Consequently, inbound and outbound traffic of detected victims are processed by Stage-3. This stage correctly detects all ground-truth attackers and their malicious flows. On average, about 1K external hosts, shown by the solid green line in Fig. 7(d)), are monitored at run-time, and only about 1% of them are actual attackers, as shown by the blue dashed line. Furthermore, we observe that the majority ($\geq$90%) of monitored flows are identified as malicious (solid yellow and dashed red lines in Fig. 7(d)). This is not surprising as each external attacker tends to overwhelm the victim by establishing many flows. At the same time, benign users do not often maintain many concurrent connections with an enterprise host.

Lastly, it is important to note that there are about 260K external hosts and 600K concurrent flows in our demo dataset. Of these, less than 4K external hosts and only 1K flows are tracked by the expensive Stage-3 for a fine-grained detection, highlighting the practicality and effectiveness of our detection system.

2) Large-Scale Comparative Evaluations: We now experimentally compare the performance of our system with that of widely-adopted industry solutions and prior academic methods, namely a commercial next-generation firewall (NGFW) appliance Palo-Alto PA-3020 [17] and an open-source intrusion detection system (IDS) Zeek [33] version 3.1.0-dev.280, and a flow graph structure (identical to what we use in our Stage-3 but analyses the entire traffic). The NGFW is evaluated as a standalone hardware appliance, while other systems (our prototype, IDS, and the flow graph) are tested as software tools on the server described in §V-B. All systems are configured by the same set of detection thresholds. Therefore, the inference mechanism is the only variable in our comparative evaluations.

For this large-scale experiment, we repeatedly injected the DDoS attack traces (discussed in §V-C1) into our enterprise dataset (the full-day trace discussed in §V-A), obtaining an evaluation dataset. Our evaluation dataset contains a total of 432 DDoS attacks targeting 432 enterprise hosts of various roles ranging from a website server and DNS server to NAT gateways, VPN proxy, and even end-hosts. Each attack instance is sourced from 100 external attackers generating 10000 malicious flows. Table II summarises our comparative evaluations (qualitative and quantitative) under three key pillars, namely runtime visibility, detection performance, and computing cost.

Runtime visibility: Due to their static capabilities, the NGFW appliances are often configured to obtain telemetry only for key enterprise servers. IT engineers manually specify the IP addresses or subnets [64] that require monitoring. In other words, NGFW appliances do not provide visibility into external hosts and traffic flows for DDoS detection. Therefore, in order to keep up with high traffic rates during peak hours, they may cause collateral damage when they protect a victim by mitigating inbound volumetric attacks [29]. Software IDS tools come with signatures of known attacks. Zeek also allows users to specify their detection logic or add custom signatures. However, running on generic CPUs, such tools cannot handle high throughput traffic for complex logic [23]. Therefore, it is recommended by the community [65] to reduce the monitoring scope to only those enterprise hosts that are essential (*e.g.,* corporate services). Our PEDDA provides more flexible and fine-grained visibility than its counterparts. It monitors all ac-

tive enterprise hosts without the need for inputs from network administrators. PEDDA reactively obtains/provides visibility into "selected" external hosts and traffic flows, communicating with identified victims of a distributed network attack. Such a level of visibility can only be achieved by flow graph-based solutions, which are computationally expensive (given their static structure and stateful inference) for real-time operation.

Detection performance: We next analyse the performance (*i.e.,* accuracy) of detecting victims, external attackers, and malicious flows. Table II highlights how PEDDA achieved decent rates of detection across victims, sources, and malicious flows (100%, 93.4%, and 91.7% respectively). On the other side, state-of-the-art solutions like NGFW and IDS equipped with detection rules and signature scripts can only raise alarms for all victims (with 100% accuracy) but are unable (protecting themselves from being overwhelmed) to differentiate distributed attackers and malicious flows. The fine-grained detection method based on flow graphs achieved the best results (100%, 95.4%, and 94.1% for victims, attackers, and flows, respectively). We observe that the flow graph method is able to detect distributed sources and malicious flows that are missed by PEDDA but not in real-time. It will need more than seven days to process our enterprise dataset (one day's worth of traffic), given the total computing resources in our server. This is because the flow graph-based method (as opposed to PEDDA) maintains states for more than a half million flow records, hence requiring a significantly large amount of computing resources.

We found that some of the enterprise victims sent more outbound than inbound packets during certain periods, thus, going undetected by our simple thresholding method. As a result, less than a tenth of (ground-truth) external attackers (6.4%) and malicious flows (8.3%) are missed. As discussed in §V-A3, an enhanced inference method more sophisticated [41], [53] than thresholding can improve detection rates but developing those methods is beyond the scope of this paper.

Computing cost: As shown in the rightmost region of Table II, our PEDDA displays a reasonable behavior in terms of computing cost (CPU/RAM consumption), yielding a better (or equivalent) ranking with respect to runtime visibility and detection performance aspects. As mentioned earlier, the flow graph requires seven times more computing resources in order to keep up with our traffic in real-time – that's why the average CPU usage is marked as ">700%". To guarantee operational robustness given any traffic rates and compositions as well as total computing resources available, our PEDDA uses an orchestrator (described in §IV-C) that adjusts the granularity of each inference stage to regulate its resource consumption. In what follows next, we unpack this aspect with numerical metrics in Figures 8 and 9.

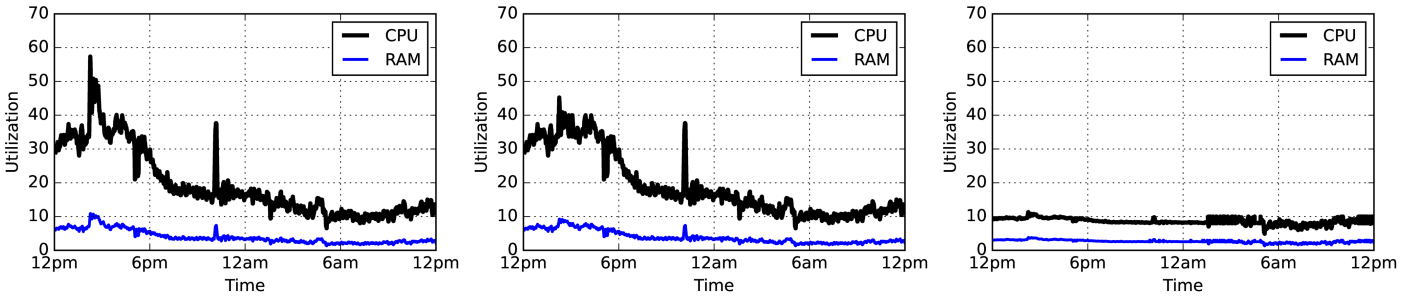
3) Optimal Resource Utilisation: In order to demonstrate the dynamics of optimal granularity adjustment across inference stages, we evaluate the impact of various resource constraints. When replaying the enterprise dataset, we imposed 36 scenarios (six choices of available CPU units and six choices of available RAM units) to our prototype.

Fig. 8 shows how the three stages operate (in terms of granularity) across these constraint scenarios. Note that each cell shows the distance from the ideal subnet mask (/32). We observe in Fig. 8(a) that Stage-1 always realises its ideal granularity (distance 0 in all scenarios). This means active enterprise hosts is determined at granularity /32. Note that we set the highest priority (W_1) to this stage in our prototype. Stage-2 performs perfectly when the available CPU is more than 60 units, but it loses one bit of granularity when the available unit of CPU drops to 40, as shown in Fig. 8(b). The loss of granularity is doubled when only 20 units of CPUs are available. In the worst-case scenario (10 units of CPU available), victims cannot be inferred better than /28 (distance of 4 for Stage-2). Note that the performance of Stage-1 and Stage-2 does not change by varying RAM units at a given value of available CPU value. Moving to Stage-3, the most expensive inference, we see more dynamics in the impact of available resources on the inference granularity, as shown in Fig. 8(c). Attackers and their malicious flows can be determined by the finest granularity /32 when at least 20 units of RAM and 80 units of CPU are available. Reducing the available RAM and/or CPU will gradually decrease the inference granularity. The lowest granularity /25 (distance of 7 for Stage-3) is obtained when only 10 units of CPU are available. Overall, it can be seen that all stages of our prototype performed well without experiencing a compromise in their inference granularity (*i.e.,* no reduction in the subnet mask) when we had at least 20 units of RAM and 80 units of CPU.

Finally, we highlight the real-time utilisation of computing resources for three representative scenarios, namely "high resource" with 25 units of RAM and 80 units of CPU (top-left cells with solid green borders in Fig. 8), "medium resources" with 15 units of RAM and 40 units of CPU (near-center cells with dashed orange borders in Fig. 8), "low resource" with 3 units of RAM and 10 units of CPU (bottom-right cells with dotted red borders in Fig. 8). These constrains are imposed by employing certain caps (on resource utilisation) applied in real-time by applications we run for our prototype. Fig. 9(a) shows that with a sufficient amount of computing resources available (80 units of CPU and 25 units of RAM), all stages yield their inference with an ideal granularity while the resource utilisation of our system is fairly healthy (well below the total capacity). Moving to medium resources available in Fig. 9(b), we see times when the CPU utilisation reaches its limit (40 units), but the RAM utilisation is still healthy (consistently below ten units when the total RAM available is 15 units). Lastly, Fig. 9(b) shows how the utilisation of both CPU and RAM consistently aims to consume all resources available (10 units of CPU and 3 units of RAM) in a scenario when low resources are available. For this very tight situation, as we saw in Fig. 8, Stage-2 and Stage-3 found their granularity slightly dropped, but the system remains robust.

VI. Conclusion And Future Work

In this paper, we proposed PEDDA, a progressive inference method that achieves both effective detection and operational practicality via multiple stages orchestrated by the dynamic control of programmable networks. We first highlighted the performance bottlenecks of traffic processing in legacy solutions by mathematically formulating their time complexity. We



(a) 80 CPU units and 25 RAM units. (b) 40 CPU units and 15 RAM units. (c) 10 CPU units and 3 RAM units.

Figure 9: Real-time utilisation of computing resources when: (a) 80 CPU units and 25 RAM units (top-left cells with solid green borders in Fig. 8), (b) 40 CPU units and 15 RAM units (near-center cells with dashed orange borders in Fig. 8), and (c) 10 CPU units and 3 RAM units (bottom-right cells with dotted red borders in Fig. 8) are available for detecting attacks in our one-day enterprise traffic dataset.

then developed our progressive method that detects distributed attacks through multiple inference stages, each with a certain cost subject to an adjustable granularity. With our system, packet streams are dynamically partitioned and processed by stages where their granularity is optimally determined and orchestrated in real-time, depending upon the progression and scale of attacks as well as available computing resources. Lastly, we implemented a prototype of our proposed solution that progressively detects active enterprise hosts, victims, and attack sources/flows. We evaluated and demonstrated the efficacy of our solution by applying it to high data rates traffic of a real enterprise network mixed with publicly available DDoS traces. Our results showed how PEEDA outperforms legacy solutions by robustness and correctly detecting victims and sources of distributed attack at reasonable granularity.

We envisage two specific directions for future work. First, one may want to adapt the PEDDA architecture for other network management and monitoring use-cases, including inferring the health of networked assets/applications (*e.g.,* works in [53], [66]), as well as detecting distributed network attacks beyond DDoS and reconnaissance (*e.g.,* works in [67], [68]). Second, for PEDDA implementation, there is scope for quantifying the advantages of using other programmable networking technologies, such as VNF and/or P4, compared to the OpenFlow-based system we studied in this paper.

REFERENCES

[1] C. Kolias, G. Kambourakis, A. Stavrou, and J. Voas, "DDoS in the IoT: Mirai and Other Botnets," *Computer*, vol. 50, no. 7, pp. 80–84, 2017.

[2] A. Wang, W. Chang, S. Chen, and A. Mohaisen, "Delving Into Internet DDoS Attacks by Botnets: Characterization and Analysis," *IEEE/ACM Trans. Netw.*, vol. 26, no. 6, Dec 2018.

[3] A. Abhishta, R. van Rijswijk-Deij, and L. J. M. Nieuwenhuis, "Measuring the Impact of a Successful DDoS Attack on the Customer Behaviour of Managed DNS Service Providers," *SIGCOMM Comput. Commun. Rev.*, vol. 48, no. 5, Jan 2019.

[4] M. Kührer, T. Hupperich, C. Rossow, and T. Holz, "Exit from Hell? Reducing the Impact of Amplification DDoS Attacks," in *Proc. USENIX Security*, San Diego, CA, USA, Aug 2014.

[5] S. T. Zargar, J. Joshi, and D. Tipper, "A Survey of Defense Mechanisms Against Distributed Denial of Service (DDoS) Flooding Attacks," *IEEE Communications Surveys & Tutorials*, vol. 15, no. 4, 2013.

[6] S. Haas and M. Fischer, "GAC: Graph-Based Alert Correlation for the Detection of Distributed Multi-Step Attacks," in *Proc. ACM SAC*, Pau, France, Apr 2018.

[7] Akamai Technologies, "2019 State of the Internet Security: DDoS and Application Attacks," https://bit.ly/3nLaohl, 2019, accessed: 2019-10-08.

[8] Blackhat, "DDoS Protection Bypass Techniques," https://bit.ly/2ZkZzJs, 2019, accessed: 2019-10-08.

[9] Forcepoint, "Attacking the Internal Network from the Public Internet using a Browser as a Proxy," https://bit.ly/3CQoBh8, 2019, accessed: 2019-10-08.

[10] K. Singh, P. Singh, and K. Kumar, "Application Layer HTTP-GET Flood DDoS Attacks: Research Landscape and Challenges," *Computers & Security*, vol. 65, pp. 344 – 372, 2017.

[11] M. H. Bhuyan, D. Bhattacharyya, and J. Kalita, "An Empirical Evaluation of Information Metrics for Low-Rate and High-Rate DDoS Attack Detection," *Pattern Recognition Letters*, vol. 51, p. 7, 2015.

[12] K. Hong, Y. Kim, H. Choi, and J. Park, "SDN-Assisted Slow HTTP DDoS Attack Defense Method," *IEEE Communications Letters*, vol. 22, no. 4, pp. 688–691, 2018.

[13] A. Wang, W. Chang, S. Chen, and A. Mohaisen, "A Data-Driven Study of DDoS Attacks and Their Dynamics," *IEEE Transactions on Dependable and Secure Computing*, vol. 17, no. 03, pp. 648–661, May 2020.

[14] Sophos Group, "Sophos XG Firewall: How to prevent DoS and DDoS attacks," https://bit.ly/2tcOPZY, 2018, accessed: 2018-02-11.

[15] Fortinet, "FortiDDoS and Verisign DDoS Protection Service," https://bit.ly/2DsDObH, 2018, accessed: 2018-02-11.

[16] Cisco Systems, "Protection Against Distributed Denial of Service Attacks," https://bit.ly/2WUbvvK, 2018, accessed: 2018-11-2.

[17] Palo Alto Networks, "DoS and Zone Protection Best Practices," https://bit.ly/2HQOMwU, 2018, accessed: 2018-28-1.

[18] T. Karagiannis, K. Papagiannaki, and M. Faloutsos, "BLINC: Multilevel Traffic Classification in the Dark," in *Proc. ACM SIGCOMM*, Oct 2005.

[19] D. Eswaran, C. Faloutsos, S. Guha, and N. Mishra, "SpotLight: Detecting Anomalies in Streaming Graphs," in *Proc. ACM KDD*, London, United Kingdom, Aug 2018.

[20] J. J. Pfeiffer III *et al.*, "Attributed Graph Models: Modeling Network Structure with Correlated Attributes," in *Proc. ACM WWW*, Seoul, Korea, Apr 2014.

[21] S. K. Fayaz *et al.*, "Bohatei: Flexible and Elastic DDoS Defense," in *Proc. USENIX Security*, Washington, D.C., USA, Aug 2015.

[22] T. Yu *et al.*, "PSI: Precise Security Instrumentation for Enterprise Networks," in *Proc. NDSS*, San Diego, CA, USA, Feb 2017.

[23] M. Zhang *et al.*, "Poseidon: Mitigating Volumetric DDoS Attacks with Programmable Switches," in *Proc. NDSS*, San Diego, CA, USA, Feb 2020.

[24] Q. Hu, M. R. Asghar, and N. Brownlee, "Measuring IPv6 DNS Reconnaissance Attacks and Preventing Them Using DNS Guard," in *Proc. IEEE/IFIP DSN*, Luxembourg City, Luxembourg, Jun 2018.

[25] F. Yarochkin, Y. Huang, Y. Hu, and S. Kuo, "Mining Large Network Reconnaissance Data," in *Proc. IEEE PRDC*, Vancouver, BC, Canada, Dec 2013.

[26] Z. Durumeric, M. Bailey, and J. A. Halderman, "An Internet-Wide View of Internet-Wide Scanning," in *Proc. USENIX Security*, San Diego, CA, USA, Aug 2014.

[27] M. Antonakakis *et al.*, "Understanding the Mirai Botnet," in *Proc. USENIX Security*, Vancouver, BC, USA, Aug 2017.

[28] S. Ramanathan, J. Mirkovic, M. Yu, and Y. Zhang, "SENSS Against Volumetric DDoS Attacks," in *Proc. ACSAC*, San Juan, PR, USA, Dec 2018.

[29] C. Dietzel, M. Wichtlhuber, G. Smaragdakis, and A. Feldmann, "Stellar: Network Attack Mitigation Using Advanced Blackholing," in *Proc. ACM CoNEXT*, Dec 2018.

[30] M. Lyu, H. Habibi Gharakheili, C. Russell, and V. Sivaraman, "Mapping an Enterprise Network by Analyzing DNS Traffic," in *Proc. Springer PAM*, Puerto Varas, Chile, Mar 2019.

[31] A. Sivanathan, H. Habibi Gharakheili, F. Loi, A. Radford, C. Wijenayake, A. Vishwanath, and V. Sivaraman, "Classifying IoT Devices in Smart Environments Using Network Traffic Characteristics," *IEEE Transactions on Mobile Computing*, vol. 18, no. 8, Aug 2019.

[32] Z. Liu, H. Jin, Y.-C. Hu, and M. Bailey, "MiddlePolice: Toward Enforcing Destination-Defined Policies in the Middle of the Internet," in *Proc. ACM CCS*, Vienna, Austria, Oct 2016.

[33] V. Paxson, "Bro: A System for Detecting Network Intruders in Real-Time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, Dec. 1999.

[34] B. Caswell, J. C. Foster, R. Russell, J. Beale, and J. Posluns, *Snort 2.0 Intrusion Detection*. Syngress Publishing, 2003.

[35] Suricata, "Community Driven. Always Alert." https://suricata.io/, 2022, accessed: 2022-04-06.

[36] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of Intrusion Detection Systems: Techniques, Datasets and Challenges," *Cybersecur.*, vol. 2, p. 20, 2019.

[37] J. Konstantas, "Enterprise Firewalls' Top Requirement: Scalability," https://bit.ly/3DNLRha, 2012, accessed: 2020-8-9.

[38] E. Damon, J. Mache, R. Weiss, K. Ganz, C. Humbeutel, and M. Crabill, "Chapter 31 - Cyber Security Education: The Merits of Firewall Exercises," in *Emerging Trends in ICT Security*, B. Akhgar and H. R. Arabnia, Eds. Boston: Morgan Kaufmann, 2014, pp. 507 – 516.

[39] Palo Alto Networks, "DoS Protection Profiles," https://bit.ly/3zsMRVW, 2020, accessed: 2020-09-09.

[40] N. Agrawal and S. Tapaswi, "Defense Mechanisms Against DDoS Attacks in a Cloud Computing Environment: State-of-the-Art and Research Challenges," *IEEE Communications Surveys & Tutorials*, vol. 21, no. 4, pp. 3769–3795, 2019.

[41] M. Lyu, H. Habibi Gharakheili, C. Russell, and V. Sivaraman, "Hierarchical Anomaly-Based Detection of Distributed DNS Attacks on Enterprise Networks," *IEEE Transactions on Network and Service Management*, vol. 18, no. 1, pp. 1031–1048, 2021.

[42] A. Gupta, R. Harrison, M. Canini, N. Feamster, J. Rexford, and W. Willinger, "Sonata: Query-driven Streaming Network Telemetry," in *Proc. ACM SIGCOMM*, Budapest, Hungary, Aug 2018.

[43] C. Liu, A. Raghuramu, C.-N. Chuah, and B. Krishnamurthy, "Piggybacking Network Functions on SDN Reactive Routing: A Feasibility Study," in *Proc. ACM SOSR*, Santa Clara, CA, USA, Apr 2017.

[44] H. Li, H. Hu, G. Gu, G.-J. Ahn, and F. Zhang, "vNIDS: Towards Elastic Security with Safe and Efficient Virtualization of Network Intrusion Detection Systems," in *Proc. ACM CCS*, Toronto, Canada, Oct 2018.

[45] Palo Alto Networks, "Reconnaissance Protection," https://bit.ly/3zo1A47, 2020, accessed: 2020-4-2.

[46] Z. Liu, R. Ben-Basat, G. Einziger, Y. Kassner, V. Braverman, R. Friedman, and V. Sekar, "Nitrosketch: Robust and General Sketch-Based Monitoring in Software Switches," in *Proc. ACM SIGCOMM*, Beijing, China, Aug 2019.

[47] A. Panda, S. Han, K. Jang, M. Walls, S. Ratnasamy, and S. Shenker, "NetBricks: Taking the V out of NFV," in *Proc. USENIX OSDI*, Savannah, GA, USA, Nov 2016.

[48] A. Sivanathan, H. Habibi Gharakheili, and V. Sivaraman, "Managing IoT Cyber-Security Using Programmable Telemetry and Machine Learning," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 60–74, 2020.

[49] A. G. Alcoz, C. Busse-Grawitz, E. Marty, and L. Vanbever, "Reducing P4 Language's Voluminosity Using Higher-Level Constructs," in *Proc. EuroP4*, Rome, Italy, Dec 2022.

[50] R. Stoenescu, D. Dumitrescu, M. Popovici, L. Negreanu, and C. Raiciu, "Debugging P4 Programs with Vera," in *Proc. ACM Sigcomm*, Budapest, Hungary, Aug 2018.

[51] NoviFlow, "NoviSwitch 2122 High Performance OpenFlow Switch," https://noviflow.com/wp-content/uploads/NoviSwitch-2122-Datasheet-1.pdf, 2018, accessed: 2018-01-28.

[52] Intel, "Intel Tofino 2," https://www.intel.com.au/content/www/au/en/products/network-io/programmable-ethernet-switch/tofino-2-series.html, 2023, accessed: 2023-03-08.

[53] M. Lyu, H. Habibi Gharakheili, and V. Sivaraman, "Classifying and Tracking Enterprise Assets via Dual-Grained Network Behavioral Analysis," *Computer Networks*, 2022.

[54] Mordor Intelligence, "Enterprise Routers Market – Growth, Trends, Covid-19 Impact, and Forcasts (2023 - 2028)," https://www.mordorintelligence.com/industry-reports/enterprise-routers-market, 2023, accessed: 2023-03-08.

[55] Cisco Systems, "Understand the OpenFlow on Catalyst 9000 Series Switches," https://www.cisco.com/c/en/us/support/docs/switches/catalyst-9300-series-switches/217210-understand-openflow-on-catalyst-9000-ser.html, 2021, accessed: 2023-03-08.

[56] Juniper Networks, "OpenFlow Support on Juniper Networks Devices," https://www.juniper.net/documentation/us/en/software/junos/sdn-openflow/topics/concept/junos-sdn-openflow-supported-platforms.html, 2020, accessed: 2023-03-08.

[57] Dell Technologies, "Hybrid OpenFlow for Dell Networking N-Series Using OpenDaylight," https://www.dell.com/support/kbdoc/en-au/000146824/hybrid-openflow-for-dell-networking-n-series-using-opendaylight, 2016, accessed: 2023-03-08.

[58] Huawei Technologies, "CloudEngine 8800, 7800, 6800, and 5800 V200R019C10 Configuration Guide - Network Management and Monitoring," https://support.huawei.com/enterprise/en/doc/EDOC1100137943/c824b277/openflow-working-mechanism, 2021, accessed: 2023-03-08.

[59] Hewlett Packard Enterprise, "HP ProCurve Switches - SDN/OpenFlow Support," https://support.hpe.com/hpesc/public/docDisplay?docId=mmr_kc-0127767, 2023, accessed: 2023-03-08.

[60] S. Jain *et al.*, "B4: Experience with a globally-deployed software defined wan," *SIGCOMM Comput. Commun. Rev.*, Aug 2013.

[61] K.-K. Yap *et al.*, "Taking the Edge off with Espresso: Scale, Reliability and Programmability for Global Internet Peering," in *Proc. ACM SIGCOMM*, Los Angeles, CA, USA, 2017.

[62] L. Poutievski *et al.*, "Jupiter Evolving: Transforming Google's Datacenter Network via Optical Circuit Switches and Software-Defined Networking," in *Proc. ACM SIGCOMM*, Amsterdam, Netherlands, Aug 2022.

[63] Imapct Cyber Trust, "DARPA 2009 Intrusion Detection Dataset," http://www.darpa2009.netsec.colostate.edu/, 2020, accessed: 2020-8-9.

[64] Palo Alto Networks, "Flood Protection," https://bit.ly/3gEEAqj, 2020, accessed: 2020-09-09.

[65] M. Rahouti, K. Xiong, N. Ghani, and F. Shaikh, "SYNGuard: Dynamic Threshold-based SYN Flood Attack Detection and Mitigation in Software-Defined Networks," *IET Networks*, vol. 10, 01 2021.

[66] M. Lyu, H. Habibi Gharakheili, C. Russell, and V. Sivaraman, "Enterprise DNS Asset Mapping and Cyber-Health Tracking via Passive Traffic Analysis," *IEEE Transactions on Network and Service Management*, 2022.

[67] J. Ahmed, H. Habibi Gharakheili, Q. Raza, C. Russell, and V. Sivaraman, "Monitoring Enterprise DNS Queries for Detecting Data Exfiltration From Internal Hosts," *IEEE Transactions on Network and Service Management*, vol. 17, no. 1, pp. 265–279, 2020.

[68] J. Ahmed, H. Habibi Gharakheili, C. Russell, and V. Sivaraman, "Automatic Detection of DGA-Enabled Malware Using SDN and Traffic Behavioral Modeling," *IEEE Transactions on Network Science and Engineering*, vol. 9, no. 4, pp. 2922–2939, May 2022.