INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2022, VOL. 68, NO. 4, PP. 889-898
Manuscript received October 10, 2022; revised November 2022. DOI: 10.24425/ijet.2022.143899

On Implementation of Efficient Inline DDoS Detector Based on AATAC Algorithm

Piotr Wiśniewski, Maciej Sosnowski, and Wojciech Burakowski

***Abstract*—Distributed Denial of Service (DDoS) attacks constitute a major threat in the current Internet. These cyber-attacks aim to flood the target system with tailored malicious network traffic overwhelming its service capacity and consequently severely limiting legitimate users from using the service. This paper builds on the state-of-the-art AATAC algorithm (Autonomous Algorithm for Traffic Anomaly Detection) and provides a concept of a dedicated *inline DDoS detector* capable of real-time monitoring of network traffic and near-real-time anomaly detection.**

The *inline DDoS detector* consists of two main elements: 1) *inline probe(s)* responsible for link-rate real-time processing and monitoring of network traffic with custom-built packet feature counters, and 2) an *analyser* that performs the near-real-time statistical analysis of these counters for anomaly detection. These elements communicate asynchronously via the Redis database, facilitating a wide range of deployment scenarios. The *inline probes* are based on COTS servers and utilise the DPDK framework (Data Plane Development Kit) and parallel packet processing on multiple CPU cores to achieve link rate traffic analysis, including tailored DPI analysis.

***Keywords*—DDoS; Distributed Denial of Service; traffic anomaly detection; AATAC; performance; DPDK**

I. Introduction

DISTRIBUTED Denial of Service (DDoS) attacks aim to overwhelm a target system's capacity in order to severely reduce its accessibility to legitimate users. An attacker uses a number of geographically distributed nodes to generate malicious traffic. Usually, the nodes are machines infected with malware that cedes their control to an attacker. A group of these *zombies* under the control of an attacker constitutes a *botnet* that enables the attacker to perform a distributed attack. Nowadays, even DDoS as a service (DaaS) is available [1]. The DaaS providers sell their services at a very low cost (from 1 USD), and directly via websites (called *booters*). The low cost and ease of access make the DDoS accessible to a wide variety of entities, from big companies wanting to hurt their competition to a single student wanting to disrupt his remote exam [2].

The DDoS attacks may be broadly categorised into 1) *extensive* and 2) *intensive* [1]. The extensive attacks are based on generating a massive volume of relatively simple traffic. E.g., an attacker may fill up the capacity of a resource like link bandwidth with malicious traffic. In such a case, an amplification technique[1] may be applied since the type of traffic is irrelevant. Some more sophisticated attacks belong to this category too. E.g. within SynFlood attack, a relatively massive amount of TCP SYN packets from many spoofed IP addresses is generated to exhaust the victim's TCP connection table preventing legitimate connections from setting up.

The intensive attacks utilise knowledge about design flaws of specific protocols and applications to tailor packets' content to exploit these flaws. E.g., the well-known Slowloris, where the attacker initiates subsequent HTTP connections and holds them open as long as possible. Specifically, zombies send packets containing only parts of an HTTP request without ever finishing these requests. The consecutive packets are sent just before a server timeout preventing the connection from being closed. Consequently, a maximum number of concurrent connections is reached in the HTTP server (e.g., 256 in the case of Apache Web server in default configuration [3]), and the legitimate requests are not accepted. Note that the volume of traffic is not noticeably increased. Consequently, the Slowloris attack is resistant to classical detection and mitigation techniques [4]. A survey of DDoS attack tools is provided in [5].

In order to defend against DDoS attacks, they need to be detected quickly. The following types of DDoS detection techniques are commonly considered [6]: 1) *knowledge based*, 2) *statistical*, and 3) *machine learning based*. Knowledge based techniques, e.g. [7], are focused on the detection of a priori known attacks. Traffic information is gathered and compared with attack-related signatures. If traffic fits an attack pattern, an alarm is raised. Note that these techniques need frequent signatures update to follow new types of attacks, which is resource-demanding. Statistical and machine learning based techniques detect unusual traffic patterns. Traffic features are compared with the historical data (or a model), and an alarm is raised when a significant deviation is detected. Machine learning requires a lot of computing power and vast datasets of training data to train the model (supervised learning, e.g. [8]) or to characterising the traffic without any previous knowledge (unsupervised learning, e.g. [9]), while the statistical methods (e.g. [10]) are relatively lightweight. Note that statistical techniques do not identify the type of attack but just detect anomalies.

In this paper, we build on a state-of-the-art statistical algorithm called AATAC by proposing and evaluating an *inline DDoS detector*. A brief overview of AATAC algorithm is

This work was supported by the Polish National Centre for Research and Development under Project CYBERSECIDENT/381651/II/NCBR/2018 (TAMA project). The project was realized by a consortium of EXATEL and Warsaw University of Technology.

Piotr Wiśniewski, Maciej Sosnowski and Wojciech Burakowski are with Warsaw University of Technology, Institute of Telecommunications (e-mail: {piotr.wisniewski2, maciej.sosnowski, wojciech.burakowski} @pw.edu.pl).

[1] Amplification DDoS attacks use the connectionlessness property of UDP. An attacker sends a requests to a server (e.g., DNS) with spoofed source IP to the victim's IP. Then, the server sends a response to the victim. It is called amplification attack since the responses are much bigger than the requests.

provided in Section II. The *inline DDoS detector* implementation concept, covering the primary design principles together with a detailed characterisation of the *inline probe* and *analyser* elements, is introduced in Section III. Sections IV and V present the testing effort, where Section IV focuses on the performance of the *inline probe* and the *analyser* elements, while Section V offers exemplary end-to-end results. Finally, Section VI summarises the paper and Section VII concisely describes future works.

II. AATAC OVERVIEW

The AATAC method [6] detects anomalies in network traffic. An anomaly is understood as the rapid change of the values of some packet stream features. These features are of two types: 1) *global features* associated with the absolute number of packets of a specific type (e.g., TCP / UDP, TCP SYN) and 2) *feature distributions* associated with the relative number of packets within the range of a specific feature (e.g., source port number distribution). The feature values are calculated as so-called *densities* at consecutive time moments. The current value of a feature density is compared with a number of stored historical densities. For this purpose, the *k-nearest neighbours* ($k-NN$) [11] algorithm is used, and the result of it is normalised to the previously observed density changes. This method determines how uncommon the current density change is – the value of this "unusualness" is called *an anomaly level*.

Let us briefly summarise the anomaly level calculation for a global feature. Feature density is calculated in an incremental fashion:

$$D(t_2) = \lambda^{t_2 - t_1} D(t_1) + w(t_1, t_2) \tag{1}$$

where: $D(t)$ is the density at time t, $w(t_1, t_2)$ is the number of the feature occurrences in the period between t_1 and t_2, and λ is the decay factor. Usage of densities instead of consecutive $w(t_1, t_2)$ values reduces the impact of natural fast-changing fluctuations.

Each density has a corresponding distance related to the previous density. The distance is the absolute value of the two densities' numerical difference. A number (say N) of consecutive densities and their corresponding distances are stored together with the mean value of the distances, μ, and the corresponding standard deviation, σ. When a new density is calculated, it is added to the densities list, and its distance to the previous density is added to the distances list. The oldest density and its corresponding distance are removed, and the values of μ and σ are updated. Next, the temporary set of distances between the new density and all the stored densities is calculated. Then, the *k*-th smallest value (*k-nearest neighbour*) of the newly obtained distances is selected: *X*. Finally, the anomaly level is calculated by normalising the value of X with respect to μ and σ:

$$A = \begin{cases} 0, & for\ X = \mu \\ abs\left(\frac{X-\mu}{\sigma}\right), & for\ X \neq \mu \end{cases} \tag{2}$$

Anomaly level A says how many standard deviations, σ, the value X is away from the mean value of the density changes, μ. When the anomaly level exceeds the predefined threshold, an anomaly is detected, and an alarm is raised. As the value of A is expressed in standard deviations, different features may be compared even if their densities differ by orders of magnitude. For example, when the traffic rate doubles, the anomaly level is the same regardless of the actual traffic rate (no matter if the traffic rate increases from 10 kbps to 20 kbps or 200 Gbps to 400 Gbps). Note that the last traffic change is compared with its previous behaviour as a reference. Therefore, even a significant density change does not strongly impact the anomaly level if such density changes were previously common. Consequently, the anomaly threshold that raises an alarm is the same for all features.

III. INLINE DDOS DETECTOR IMPLEMENTATION CONCEPT

A. Introduction and design principles

The main idea behind the *inline DDoS detector* is to provide the implementation concept for efficient, scalable, and elastic deployment of real-time AATAC-based method in high-speed networks that constitutes autonomous systems of the current Internet. Specifically, the concept is mainly suited for Tier 3 Internet Service Providers (ISPs) primarily engaged in delivering Internet access to end customers.

The mitigation of the DDoS attacks requires two main elements: 1) the *detector* element responsible for identifying the attacks/anomalies in network traffic, and 2) the *discarder* responsible for dropping the packet flows identified as attacks. This paper provides the concept of the *inline DDoS detector* element with further remarks regarding its integration with the *discarder* element.

The following two main approaches for the realisation of an AATAC-based *inline DDoS detector* may be considered: 1) implementation of dedicated devices (*inline probes*) in the network responsible for network traffic analysis and calculation of real-time flow characteristics and 2) exploitation of statistics provided by inline network devices (usually routers). The former enables a thorough examination of network traffic characteristics tailored for anomaly detection, but it requires efficient devices capable of performing deep packet inspection (DPI) at link rate speed. The latter utilises the capabilities of routers (already deployed in the network), but it provides traffic analysis options limited to flow statistics provided by the router, usually with NetFlow, IPFIX, or J-Flow solutions [12]. Moreover, the latter relay heavily on traffic sampling [13] due to router CPU power constraints. Note that traffic sampling negatively impacts anomaly detection accuracy.

This paper proposes the concept of an AATAC-based *inline DDoS detector* accordingly to the first approach. The following paragraphs present the primary design principles behind our *inline DDoS detector* concept.

1) Service-tailored anomaly detection

We assume that an organisation (usually an ISP) provides the DDoS mitigation service (DDoS shield) for a number of clients. Each client is assigned with a set of monitored entities identified by the couple consisting of IP address together with direction[2]. Each monitored entity is assigned with a monitoring policy defining a set of packet feature filters.

[2] Identification of direction enables to monitor traffic towards the IP address, from the IP address or in both directions simultaneously.

TABLE I
GLOBAL PACKET FILTERS

Global packet feature filter - list of counters	Filter description
Invalid&FragmentedFilter - Invalid, Fragmented	Filter covers invalid and fragmented IP packets. Invalid counter covers packets for which for which the declared TCP/UDP header length in inconsistent with the actual length. Only packets caring TCP or UDP datagrams may increment this counter. Fragmented counter covers packets being a fragment of bigger packet based on IP header (fragment offset field and more fragments (MF) flag in case of IPv4; Fragmentation header in case of IPv6). Only IPv4 and IPv6 packets may increment this counter.
EtherType&ProtocolFilter - EtherType counters: IPv4, IPv6, ARP, OTHER - Protocol counters: ICMP, TCP, UDP, OTHER	Filter covers packets base on the values of EtherType field of Ethernet header and Protocol/NextHeader filed of IPv4/IPv6 header. Each processed packet increments: 1) exactly one of EtherType counters depending on the value of EtherType field of Ethernet header (IPv4, IPv6, ARP or OTHER), and 2) exactly one Protocol counter depending on the value of Protocol/NextHeader filed of IPv4/IPv6 header (ICMP, TCP, UDP or OTHER).
TcpFlagFilter - SYN, SYN&ACK, RST, FIN, FIN&ACK	Filter covers packets carrying TCP datagrams based on TCP flags. Each processed packet carrying TCP datagram increments at most one of the TCP flag counters depending on the active flags in TCP header (SYN, SYN&ACK, RST)
SlowHttp filter - SlowHttp	Filter covers packets characteristic for Slow Http Header attack (Slowloris). It covers HTTP packets that are not ended with the characteristic "\r\n\r\n" end symbol. Specifically, it counts packets that match the following: 1) encapsulation: IPv4 or IPv6, TCP, HTTP (identified by destination port equal to 80), 2) two last bytes equal to \r\n, 3) two penultimate two bytes not equal to \r\n.
SlowHttpGet filter - SlowHttpGet	Filter covers packets characteristic for Slow Http Header attack (Slowloris) similarly to SlowHttp counter. It covers HTTP GET packets that are not ended with the characteristic "\r\n\r\n" end symbol. Specifically, it counts packets that match the following: 1) encapsulation: IPv4 or IPv6, TCP, HTTP (identified by destination port equal to 80), 2) first tree bytes of TCP payload correspond to “GET” keyword, 3) two last bytes equal to \r\n, 4) two penultimate two bytes not equal to \r\n.
SlowHttpPost filter - SlowHttpPost	Filter covers packets characteristic for Slow Http body attack (RUDY). It covers HTTP POST packets that are not ended with the characteristic "\r\n\r\n" end symbol. Consecutively, it counts packets that match the following: 1) encapsulation: IPv4 or IPv6, TCP, HTTP (identified by destination port equal to 80), 2) first three bytes of TCP payload correspond to “POST” keyword, 3) two last bytes equal to \r\n, 4) two penultimate bytes not equal to \r\n.

This approach enables high flexibility of the DDoS shield adoption in the network. Each client of the DDoS shield service may define a set of servers (identified by IP addresses) running services that should be protected. Depending on the service characteristics, a monitoring policy is defined (e.g., in the case of a web service, it is crucial to detect traffic anomalies indicating slow HTTP DDoS attacks, whereas, in the case of a VPN service, such detection is unnecessary). Moreover, a different monitoring policy may be assigned depending on the traffic direction (towards or from the IP address). Consequently, the approach enables to detect of attacks targeted against predefined services with customisable monitoring policies dependent on the service characteristics.

Additionally, two special monitoring entities are defined, so-called *otherIn* and *otherOut.* These cumulative entities cover all traffic that would not be monitored otherwise (the traffic destined to or coming from the IP addresses not-covered by already defined monitoring entities). These entities may be assigned with monitoring policy (as any other). The idea behind the *otherIn* and *otherOut* entities is to enable a cumulative analysis of the background traffic.

2) High scalability and deployment elasticity

Our *inline DDoS detector* consists of two main elements 1) *inline probe(s)* responsible for link-rate real-time forwarding and monitoring of network traffic with custom-built packet feature counters, and 2) an *analyser* that performs the near-real-time statistical analysis for anomaly detection. These elements communicate asynchronously via the Redis database. Note that the database and the *analyser* may be located anywhere in the network, e.g., in the ISP cloud. The decoupling of the AATAC-based *inline DDoS detector* into two main elements facilities high scalability and a wide range of deployment scenarios. The *inline probes* may be located either at: 1) client links (so they process and forward traffic limited to specific clients), 2) at inter-domain ingress links, so they monitor traffic at the ISP network edge, or 3) in other locations following ISP specific requirements. The *analyser* is able to aggregate information (related to monitored entities) from different *inline probes,* see subsection III.C for details. Consequently, additional probes may be deployed in the network if an ISP requires to scale up the solution.

3) Custom packet feature filters

The primary idea behind the *inline probe* is to overcome the limitations of statistics provided by IPFIX-like solutions in routers. The accuracy of these flow statistics is cramped due to the required high traffic sampling rates, and the depth of the analysis is limited to the available standard statistics, e.g. DPI analysis of the HTTP body required to detect the previously mentioned Slowloris attack is not possible. Consequently, such solutions are considered unsatisfactory from the AATAC-based method deployment.

The *inline probes* introduce the concept of a custom *packet feature filter* that is able to analyse the whole packet content in the context of a defined feature, including DPI on the entire packet content if necessary. Specifically, a *packet feature filter* defines a set of (packet feature) counters describing packet features that are monitored. Building on the AATAC method, we distinguish two types of filters: global and distribution filters.

Global packet feature filters constitute a set of counters, each associated with the absolute number of packets of a specific type, e.g., *TcpFlagFilter* contains a set of counters for different flag options in the TCP header. Each counter is of absolute/global type: one traffic feature is described by exactly

TABLE II
DISTRIBUTION PACKET FILTERS

Distribution packet feature filter - number of counters	Filter description
FrameSizeFilter - 9 counters	Filter covers Ethernet frame size distribution. Each packet increments exactly one of 9 counters (bins) depending on the Ethernet frame length. Counters 0, 1, …, 8 cover, accordingly, the following frame size ranges: [0-200), [200-400), [400-600), [600-800), [800-1000), [1000-1200), [1200-1400), [1400-1600), [1600- inf).
SrcIPFilter - 10 counters	Filter covers source IP address distribution. Each IP packet increments exactly one of 10 counters (bins) depending on IP source address. The counter id determined as source IP address (128-byte variable: uint128) modulo 10.
SrcPortFilter - 10 counters	Filter covers source port distribution (TCP or UDP source port). Each packet carrying TCP or UDP datagram increments exactly one of 10 counters (bins). Counters 0, 1, …, 9 cover, accordingly, the following source port ranges: [0-7000), [7000-14000), [14000-21000), [21000-28000), [21000-35000), [35000-42000), [42000-49000), [49000-56000), [56000-63000), [63000- 65535].
DstPortFilter - 10 counters	Filter covers source port distribution (TCP or UDP source port). Each packet carrying TCP or UDP datagram increments exactly one of 10 counters (bins). Counters 0, 1, …, 9 cover, accordingly, the following source port ranges: [0-7000), [7000-14000), [14000-21000), [21000-28000), [21000-35000), [35000-42000), [42000-49000), [49000-56000), [56000-63000), [63000- 65535].
TTLFilter - 10 counters	Filter covers TTL distribution (TTL head field value in the case of IPv4 packets and hop limit header field value in the case of IPv6 packets). Each IP packet increments exactly one of 10 counters (bins). Counters 0, 1, …, 9 cover, accordingly, the following TTL/hop limit ranges: [0-26), [26-52), [52-78), [78-104), [104-130), [130-156), [156-182), [182-208), [208-234), [234- 255].

one counter. The description of the implemented global packet filters (6 filters with 18 counters covering 18 traffic features) is presented in Table I. Note that *SlowHttp*, *SlowHttpGet*, and *SlowHttpPost* filters perform application layer analysis for slow attacks detection [4].

Distribution packet filters constitute a set of counters, each associated with the relative number of packets within the range of a specific feature, e.g., *FrameSizeFilter* contains a set of counters covering the histogram of frame sizes. Each counter is of relative type: one traffic feature is covered by a set of counters. The description of the implemented distribution packet features filters (5 filters with 49 counters covering 5 traffic features) is presented in Table II.

Additionally, each filter is implemented in two versions: 1) standard version, where each matching packet increments a counter by one, and 2) byte version, where each matching packet increments a counter by the packet's length denominated in bytes.

4) Commodity hardware-based solution

In order to provide a cost-effective, readily available, and easily upgradable solution, we designed and implemented our *inline DDoS detector* using COTS (Commercial Off-The-Shelf) hardware and software. Even the performance-critical *inline probe* requires a standard COTS Linux x86 server with a commonly available DPDK-supported NIC (Network Interface Card).

5) Link rate traffic analysis

The efficiency of the *inline probe* handling of the actual network traffic is critical for deploying the *inline DDoS detector*. In order to achieve link rate traffic analysis on COST hardware, the Data Plane Development Kit technology (libraries for the acceleration of packet processing workloads) was utilised together with the application of parallel packet processing on multiple CPU cores [14].

6) Inline DDoS detector and discarder seamless integration

The *inline probe* element of our *inline DDoS detector* is directly forwarding packets, as its name suggests. Note that the

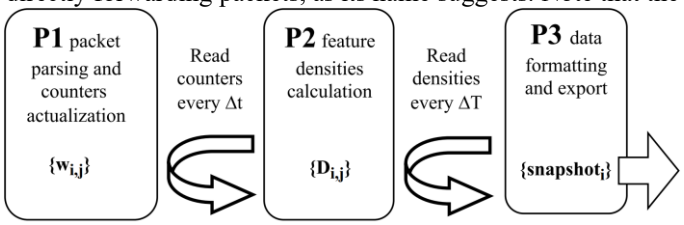

Fig. 1. Inline probe processing phases

discarder device is also of inline nature as it drops the packets identified as attacks.

Our *inline probe* was implemented as an extension of the *Gladdo*s discarder [15], both developed and tested in the TAMA project. Thanks to that, the *inline probe* and *discarder* may run as one software component on the COST server. Depending on the start-up command line argument, this software serves as: 1) *inline probe* solely, 2) *discarder* solely, or 3) *inline probe* and *discarder* simultaneously. Note that the *discarder*'s details are out of this paper's scope.

B. Inline probe

The *inline probe* realises packet processing in three distinctive phases: P1) packet parsing and counter actualisation, P2) feature destinies calculation, and P3) data formatting and export. These phases are depicted in Fig. 1 and further described in the following sections.

1) Phase P1: packet parsing and counter actualisation

Phase P1 is the most computationally demanding phase as it requires link-rate processing of network packet traffic (e.g., 10 Gbps of traffic equals about 1 225 490 frames per second, assuming 1000 B frame size). Consequently, the packet traffic is load balanced on the set of available CPU logical cores. Each NIC physical port is assigned to be served by a number of CPU logical cores (one or more)[3], each bound to a number of

[3] The logical cores assigned to each port are configured via start-up parameter. Each port may be assigned to a different number of cores.

RX queues[4]. The packets are distributed among the assigned RX queues using the RSS (Receive Side Scaling) feature. The RX queue is determined by hashing the appropriate IP addresses and

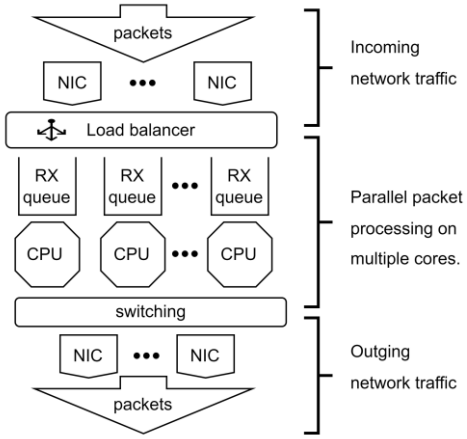

Fig. 2. Parallel packets processing on multiple cores in phase P1 - simplified

TCP/UDP port numbers, so there is no packet reordering in any flow[5]. The simplified logical scheme of this process is depicted in Fig. 2.

Each logical core runs a thread implementing a pipeline of packet feature filters. Depending on the monitoring policy assigned to a given monitored entity, appropriate filters are activated, and associated counters[6] are updated in real-time. Cumulatively, the whole set of counters $\{w_{i,j}: i \in I, j \in J(i)\}$ is updated in real-time, where I denotes a set of monitored entities and $J(i)$ denotes a set of features associated with i-th entity via a monitoring policy[7].

Note that the $\{w_{i,j}\}$ abbreviation for $\{w_{i,j}: i \in I, j \in J(i)\}$ is used in the rest of the paper. Similarly, the abbreviation $\{D_{i,j}\}$ is used for the set of densities $\{D_{i,j}: i \in I, j \in J(i)\}$. A set of densities associated with a monitored entity constitute a *snapshot* describing this entity at a given time moment. The $\{snapshot_i\}$ abbreviation is used for the set of snapshots $\{snapshot_i: i \in I\}$ where each $snapshot_i$ covers all densities associated with the i-th monitored entity.

2) Phase P2: feature destinies calculation

Phases P2 and P3 are implemented in the main thread destined to run on a CPU core different from P1 phase threads. The phase P2 covers the calculation of the set of feature densities $\{D_{i,j}\}$ corresponding to $\{w_{i,j}\}$ counters. While the feature counters are updated in real-time, the densities are calculated at constant intervals Δt, as depicted in Fig. 1. At these time intervals, the main thread fetches all current values of the feature counters. Number of feature occurrences during the Δt interval, $w_{i,j}(t_1, t_2)$, is a difference between the current and the previous counter $w_{i,j}$ values[8].

Each density $D \in \{D_{i,j}\}$ is calculated as follows:

$$D(t_2) = C_1\, D(t_1) + C_2\, w(t_1, t_2), \tag{3}$$

where $C_1$and C_2 are constants that are precomputed as[9]:

$$C_1 = \lambda^{\Delta t}, \quad C_2 = \frac{1-\lambda^{\Delta t}}{\Delta t}. \tag{4}$$

We replaced the original equation (1) with (3) in order to reduce the computation complexity and make the density value easier to interpret. Specifically, having constant Δt allows us to pre-calculate values of C_1 and C_2 and use them as constants during *inline probe* operation. This reduces the computational complexity of *phase P2* as the formula (3) contains only three easy operations: two multiplications and one addition.

In comparison to equation (1), we divided $w(t_1, t_2)$ term by Δt. Consequently, the densities are now denoted in units of [1/s], regardless of the actual Δt interval length. Moreover, we multiplied $w(t_1, t_2)$ term by $(1 - \lambda^{\Delta t})$ to apply EWMA (Exponentially Weighted Moving Average) method to density calculation. Thanks to it, the calculated densities provide information about the current smoothed average of packets per second.

Let us consider a simple example concerning a single feature: constant bitrate traffic with the rate of 400 pps (packets per second), $\Delta t = 0.1$s, and $\lambda = 0.5$. Consequently $w(t_1, t_2) = 40$ and $C_1 = \lambda^{\Delta t} \approx 0.93$, $C_2 \approx 0.7$. In the steady-state, density D calculated accordingly to formula (1) equals:

$$D = 0.93D + 40 \;\rightarrow\; D \approx 571\ [\text{packets}], \tag{5}$$

while density D calculated accordingly to formula (3):

$$D = 0.93D + 0.7 \cdot 40 \;\rightarrow\; D \approx 400\ [\text{pps}]. \tag{6}$$

One can see that the latter provides an easily interpretable value of smoothed packet rate, while the former does not. It should be noted that from the point of view of anomaly detection, both equations (1) and (3) can be used. However, the introduced adjustments provide that the density is not only an intermediate result but also carries information that might be used to detect specific types of DDoS attacks. This issue is further described in Section VI.

3) Phase 3: data formatting and export

The phase P3 covers the preparation and export of snapshots $\{snapshot_i\}$ for all entities, see Fig. 1. Densities $\{D_{i,j}\}$ are read at constant intervals ΔT and the snapshots are prepared and sent to the Redis database. The ΔT satisfies the following:

$$\Delta t \leq \Delta T. \tag{7}$$

This enables to maximise the accuracy of $\{D_{i,j}\}$ calculation while bounding the amount of exported data[10].

Each record sent to the database is a set of key-value pairs where nested structures are allowed. Each record consists of:

[4] By default, the number of queues per logical core equals one, this can be increased up to 12 via start-up parameter.

[5] Packets in RX queue are processed by assigned logical core in FIFO (First In First Out) fashion. Packets may be reordered only if they are processed on different cores.

[6] A monitoring policy defines a set of packet filters each constituting a set of counters.

[7] Note that each counter may be in one of the two versions as described in Section III.3.

[8] The $w(t_1, t_2)$ is the number of the feature occurrences in the period between t_1 and t_2, as described in Section II. Note this period's length is constant. It equals to Δt as we assume that the densities are calculated at constant intervals.

[9] The λ parameter is the decay factor, as described in Section II.

[10] The accuracy of calculation of $\{D_{i,j}\}$ depends on the Δt length. Smaller Δt value corresponds to better accuracy.

1) Unique ID. The ID is concatenated from the *inline probe* ID and monitored entity ID (IP address and direction of traffic: DST – towards the address, or SRC – from the address). An exemplary ID is as follows:
"`PROBE:0:DENSITIES:ENTITY:70.0.0.202:SRC`".

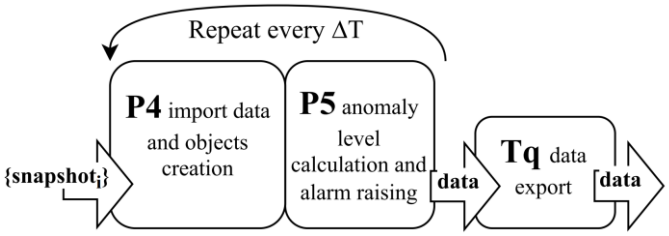

Fig. 3. *Analyser* processing phases

2) Set of densities ($snapshot_i$). Each density has a name and value. In the case of distribution features, there is a number of name-value pairs corresponding to the feature bins, e.g., `TTL_0`, `TTL_1`, …, `TTL_9` (the bins distribution is defined in Table II).
3) Timestamp of the snapshot.
The Unique ID is the key of a record stored in the database, while the set of densities (with a timestamp) is its corresponding value. Note that updating records overrides data, so only the last snapshot is stored for each Unique ID.

C. Analyser

The *analyser* is responsible for the statistical processing of measurement data $\{snapshot_i\}$, detecting attacks (anomalies), and generating alarms. The process is realised in two phases: P4) import data and object creation, and P5) anomaly level calculation and alarm raising. At the end of phase P5, a *data export* job is created and pushed to a *task queue* that is run as another container. These phases are depicted in Fig. 3 and further described in the following sections.

1) Phase P4: data import and objects creation

The *analyser* periodically fetches the records stored in the Redis database (at ΔT interval). Information about entities and their monitored features is extracted from these records, the corresponding objects are created[11]. The *analyser* stores a list of *inline probes* with their entities' data. Specifically, for each entity, the following information is stored: 1) entity ID (`ENTITY_IP:{DST/SRC}`), 2) timestamp of the last snapshot, 3) N most recent snapshots, 4) anomaly levels for all features, and 5) active alarms. If several *inline probes* monitor one entity, an aggregated object that sums up densities is created. Depending on the configuration, the *analyser* calculates the anomaly levels only for the aggregated object or for every single object. In the first case, the processing is faster. In the second case, a higher granularity of information is provided to detect the direction from which the attack originates. Additionally, timestamps are used to determine whether an entity is actively monitored by an *inline probe* (so its timestamps change) and whether *inline probes* work correctly (so its consecutive timestamps intervals do not differ significantly from assumed ΔT), any abnormal result generates a warning.

[11] Each record is related to an *inline probe* and its entities, so if the *analyser* does not have an object for the *probe* and/or entity, it creates such an object.

2) Phase P5: anomaly level calculation and alarm raising

The anomaly level calculation follows the procedure described in Section II for each entity's feature. An alarm is raised when an anomaly level exceeds the client-defined threshold. Information about current densities, anomaly levels, alarms, and warnings is stored in a time series database (e.g., Elastic Search) and might be visualised (e.g., in Grafana). Uploading the data to the time series database is performed by a *task queue* where the *analyser* outsources the job.

Our *analyser* stores not only densities and distances (as the original AATAC algorithm), but also differences between consecutive densities. Summing up all stored differences, we get a positive or negative number, the $(sign)$ equals 1 or -1. This *sign* identifies whether the anomaly is related to an unusual increase or decrease in the observed traffic, allowing to identify the start and the end of the attack accordingly. We enhanced the anomaly level formula with *the sign* of the anomaly to include this information:

$$A = \begin{cases} 0, & for\ X = \mu \\ (sign) \cdot abs\left(\frac{X-\mu}{\sigma}\right), & for\ X \neq \mu \end{cases}. \quad (8)$$

Let us mention that the calculation of the μ and σ from a set of N distances is computationally expensive. This calculation is performed for each new difference value accordingly to the AATAC algorithm (see Section II for details). Thus, we developed and implemented simplified formulas for updating μ and σ^2 (and thus σ) of the set when a new difference value is obtained:

$$\mu_{new} = \mu_{old} + \frac{x_N - x_0}{N}, \quad (9)$$

$$\sigma^2_{new} = \sigma^2_{old} + \frac{(x_N - \mu_{new} + x_0 - \mu_{old})(x_N - x_0)}{N-1}. \quad (10)$$

where x_N is the oldest difference value removed from the set (on N values) and x_0 is a new one.

The *analyser* (as well as a *task queue*) is implemented in python and runs as a docker container. Consequently, it is possible to run multiple instances of this module and specify a range of monitored entities that each instance should process. This enables to scale-up of the solution if needed.

IV. Feasibility Study

To assess the *inline DDoS detector*'s feasibility, we performed several studies of implementation covering the whole solution. The functional tests proved that our implementation is correct. Among others, the packets with a given feature were correctly counted, the corresponding densities and anomaly levels were correctly calculated.

The following subsection briefly describes the aspect of performance studies of our implementation covering both elements: the inline probe and the analyser.

A. Inline probe

The *inline probe's* performance is critical for deploying the *inline DDoS detector*, as the *inline probe* is responsible for handling network traffic. Consequently, some traffic is lost if the incoming traffic overloads its capacity.

In order to qualify the performance of the implementation, we performed a number of measurements experiments based benchmarking methodology defined in the RFC 2544 [16] for the throughput metric defined in the RFC 1242 [17]. The throughput is defined as the fastest rate at which none of

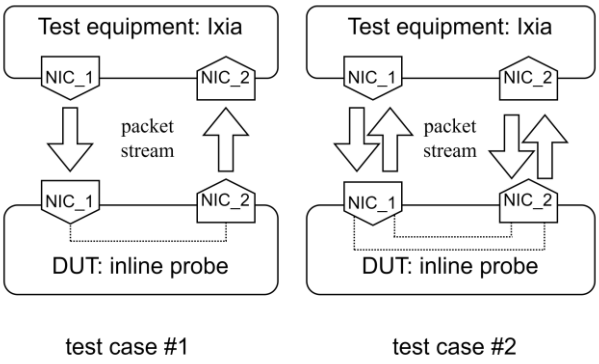

Fig. 4. Test topology

the offered frames is dropped by the device under test (DUT). We assumed that each single measurement iteration lasts 20 seconds with the inter-interaction break equal to 15 s.

The measurements were performed with an *inline probe* running on HP ProLiant DL380 Gen 9 server (DUT) equipped with Intel Xeon CPU E5-2690 v3 @ 2.60GHz, 125,86 GB RAM (including 96 hugepages of 1 GB each), four Intel 82599ES 10-Gigabit SFI/SFP+ NICs. The tests were performed using Ixia hardware XM2 tester equipped with 10 Gbps Ethernet (SFP+) ports.

Here we describe two exemplary test cases, as presented in Fig. 4. The first test case (#1) assumes the 10 standard entities in the DST direction. Each entity is assigned with monitoring policies covering all 11 packet feature filters (67 counters) in the DST direction (towards the monitor object). Additionally, each filter is applied in both configurations[7], effectively doubling the number of active counters. The DUT is offered a traffic stream generated by Ixia tester, constituting of 10 IP flows associated with the monitored objects. Each flow's destination IP address matches IP address of one of the monitored entities. Each NIC physical interface has been assigned to be served by exactly one CPU logical core. Consequently, 3 logical cores are used: one handling the main thread (phases P2 and P3), one handling the packets incoming via NIC_1 (phase P1), and one handling packets incoming via NIC_2 (phase P1). Note that the last core actually does not process any packets in this scenario, as no packets are received by DUT on NIC_2.

The measured throughput for different frame sizes is depicted in Fig. 5, together with maximal theoretical values on 10 Gbps link. The measurement results were not precisely repeatable, the statistical processing was applied. Specifically, the RFC 2544 throughput procedure was repeated 10 times for each frame size, and 95% confidence intervals were calculated. The obtained results show that the *inline probe* can achieve near link-rate traffic handling (at least 95% of link speed) for frames bigger than 758B using just one logical core handling packet processing (P1 phase). More CPU logical cores may be assigned to a physical NIC port if higher performance is required. The incoming traffic would then be load-balanced using the RSS mechanism as described in Section III.B.1.

The second test case (#2) extends the previous scenario with bidirectional traffic. It assumes the 10 standard entities in both DST and SRC directions. All entities are assigned with a policy covering all 11 packet feature filters (as previously).

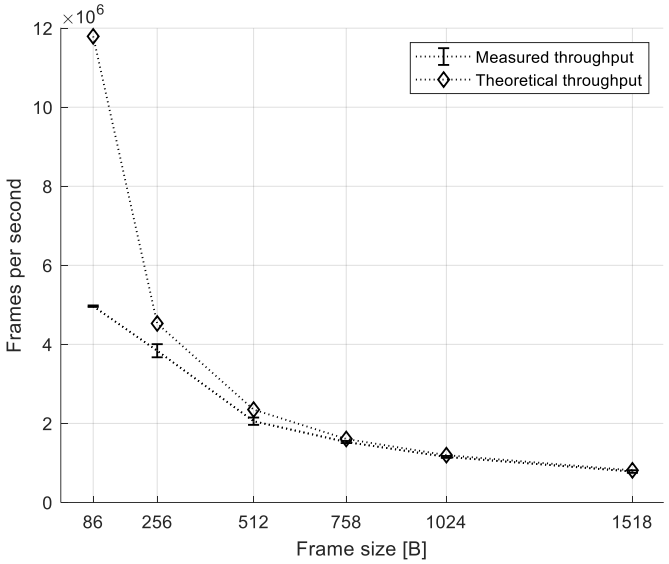

Fig. 5. Throughput achieved on single CPU logical core handling P1 phase

The DUT is offered two traffic streams, one on each interface. Each traffic flow constitutes od 10 IP flows associated with the monitored objects. For flows received by DUT on NIC_1, the flow's destination IP address matches the IP address of one of the monitored entities in the DST direction. For flows received by DUT on NIC_2, the flow's source IP address matches IP address of one of the monitored entities in the SRC direction. Each NIC physical interface has been assigned to be served by exactly one CPU logical core, as previously. Consequently, 3 logical cores are used, one handling the main thread, one handling the packets in the DST direction incoming via NIC_1, and one handling packets in the SRC direction incoming via NIC_2.

Similarly, as in the previous test case, the RFC 2544 throughput procedure was repeated 10 times for each frame size, and 95% confidence intervals were calculated. The obtained results for each direction are statistically equivalent to the results obtained in the first test case[12], thus, they are not plotted in Fig. 5. This derives from the fact that the traffic incoming by each NIC was handled by a different CPU logical core (phase P1). Consequently, the performance of the *inline probe* may be easily scaled up by increasing the number of assigned CPU logical cores.

A. Analyser

The *analyser* performs its primary functions, phases P4 and P5, cyclically every ΔT[13]. Within this time, the module needs to execute the following actions: 1) download data from Redis, 2) update entities' statistics with new snapshots, 3) calculate anomaly levels for each entity's features and compare them with alarm thresholds, 4) add the uploading data job to a *task queue*. These actions are realised by the *main loop* of the *analyser*.

[12] The confidence intervals for all measurement points are overlapping so there is no statistically significant difference between the results.

[13] The default value of ΔT is 1 s.

After executing each round of the main loop, the process sleeps for some time to start a new cycle exactly after ΔT from the previous one.

In order to assess the *analyser's* performance, we measured how long the main loop round takes depending on the number of monitored entities. The *analyser* ran as a container on a virtual machine (VM) with the following characteristic: 4 vCPU @ 2.60GHz cores, 16 GB RAM, Debian Stretch (Release 9.8, kernel 4.9.0) with Docker 18.09.4 installed. The Redis database ran as a container on the same VM to minimise the impact caused by network delays.

In the first scenario, the *analyser* processed data obtained from one *inline probe*. All entities were assigned the same monitoring policy, with all feature filters being active. Results of the measurements are presented in Fig. 6. Measurements were repeated 60 times for each number of monitored entities, and 95% confidence intervals were calculated. When there are no entities' data in the Redis database, the round takes only 2.4 ms as no calculations are performed (only communication with the database is required). When some entities' data is in the database, the *analyser* executes an anomaly detection process for each entity's features. Consequently, the main loop execution time increases proportionally to the number of monitored entities (as all entities are assigned the same monitoring policy). It takes approx. 7-10 ms to process one entity's data.

In the second scenario, a number of *inline probes* are considered[14]. The results are presented in Tab. III. We assumed that all *probes* together are assigned with 32 entities in different configurations. The *analyser* was configured to calculate anomaly level for all single objects (not only for an aggregated object, if it's present). The first three columns cover the cases where 1, 2, and 4 *inline probes* are considered, and each *inline probe* monitors different entities. One can see that the mean time per entity increases with the number of *inline probes*. It derives from the fact that the *analyser* performs some procedures per *inline probe*, e.g., it checks if an entity is also monitored by other *inline probes* (so an aggregated object should be created).

The last two columns cover the cases where 2 and 4 *inline probes* monitor common entities. In the case of 2 *inline probes,* the first monitors entities 1-16, and the second monitors entities 9-24. In the case of 4 *probes*, all monitor the same 8 entities. One can see that the results for these cases do not differ. Comparing different columns, we see that where the *inline probes* monitor common entities, the main loop execution is a little bit faster (as alarms are handled only for the aggregated objects).

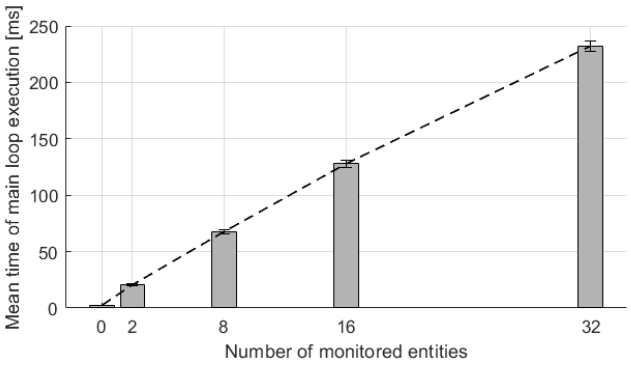

Fig. 6. Analyser efficiency as a function of monitored entities

TABLE III
ANALYSER EFFICIENCY – COMMON MONITORED ENTITIES.

Number of inline probes	1	2	4	2	4
Number of entities per inline probe	32	16	8	16	8
Number of common entities	-	0	0	8	8
Mean time of main loop execution [ms]	232	264	330	284	288
95% confidence [ms]	4,9	9,9	9,8	11,7	6,1
Mean time per entity [ms]	7,3	8,3	10,3	8,9	9,0

We can conclude that the *analyser's* efficiency is sufficient to handle all the presented cases. The main loop execution time for such a number of *inline probes* and entities equals 250-350 ms (7-10 ms per entity). It means that the *analyser* is able to calculate data for approx. 100 entities within assumed $\Delta T = 1\,s$. If there is a need to monitor more entities, as mentioned earlier, many *analysers* may be run and configured to process only a subset of entities, or ΔT may be increased.

V. EXEMPLARY RESULTS

This section presents exemplary end-to-end anomaly detection results performed by our *inline DDoS detector.* We consider two test scenarios, both following a scheme presented in Fig. 7. The traffic is sent by Ixia *BreakingPoint* directly to our *inline DDoS detector.* It consists of 15 minutes of background traffic and an attack conducted between the 6th and the 9th minute (360 – 540 s). We used "BreakingPoint Enterprise 2018" traffic profile consisting of a mix of application traffic.

To focus the attention, we consider a single traffic feature measured by the *inline DDoS detector:* Ipv4 packets (covered by EtherType&ProtocolFilter). Figures 8 and 9 present the densities and anomaly levels measured by the *inline DDoS detector* in the default configuration[15]. These figures were plotted based on the data exported to the time series database by the *analyser.* In the first scenario (Fig. 8), the traffic rate rapidly increases by 60% and stays the same during the attack period.

[14] Since didn't have required number of the *inline probes*, we use a mock imitating several of them.

[15] Parameters values: $\lambda = 0.6, \Delta t = 0.1\ s, \Delta T = 1\ s,\ N = 50$, and $k = 11$.

In the second scenario (Fig. 9), the malicious traffic is significantly more irregular.

The anomaly results appear after 50 seconds, as the algorithm needs to collect $N = 50$ snapshots to start statistical processing. In the first case (Fig. 8), the anomaly level has two outstanding values – at the attack's beginning and end. In these moments, the traffic changes are abnormally rapid and strong.

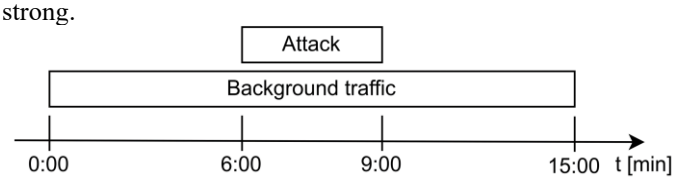

Fig. 7. Timeline of DDoS attack scenarios

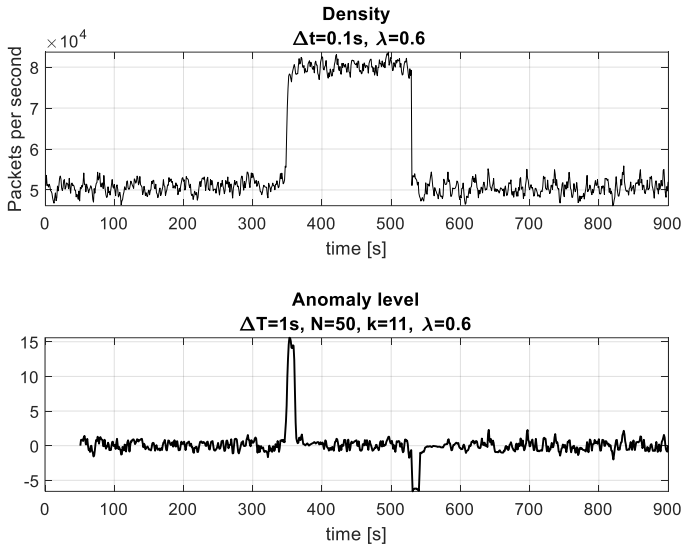

Fig. 8. Visualisation of the anomaly level for exemplary traffic – regular malicious traffic

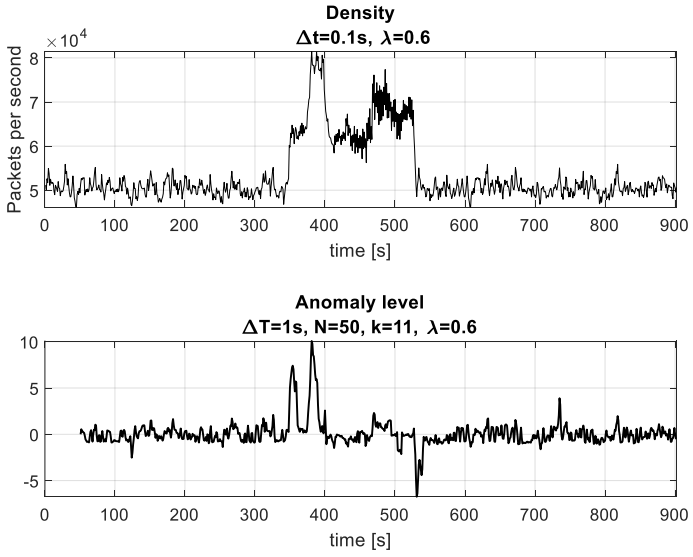

Fig. 9. Visualisation of the anomaly level for exemplary traffic – irregular malicious traffic

In the second case (Fig. 9), there are three outstanding values of anomaly level – approx. in 360, 380, and 540 s. Note that the decrease in the traffic rate in 400 s does not produce a noticeable (negative) anomaly level. This is because 1) 60 000 pps is not an abnormal value considering the last 50 seconds, and 2) another significant change (from 60 000 to 80 000 pps) was observed in the last 50 seconds, so such changes are considered normal. In 540 s, there is a (negative) anomaly level since, during the previous 50 s, there were approximately 65 000 – 75 000 pps that rapidly decreased to 50 000 pps.

Note that real-time monitoring may be performed by reading data stored in from Elastic Search with Graphana reading. A sample visualisation of such monitoring is shown in Fig. 10. The density of total traffic, measured in Mbps (green transparent chart, left y-axis), with its corresponding anomaly level (white plot, right y-axis), is monitored. Additionally, two constant client-defined anomaly levels are marked ($A = 2$ and $A = 3$) to easily identify moments of unusual traffic behaviour.

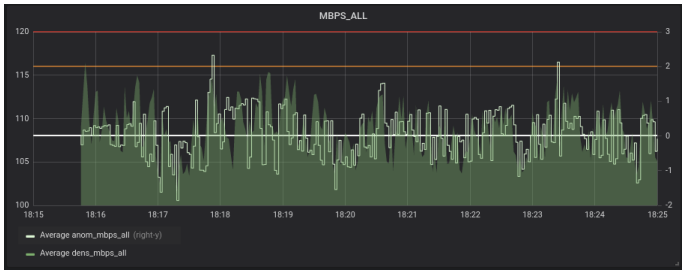

Fig. 10. Visualisation of the anomaly level of exemplary traffic – Graphana

VI. CONCLUSIONS

This paper provides a concept of the *inline DDoS detector* capable of real-time network traffic monitoring for near-real-time anomaly detection. Our *inline DDoS detector* is based on the state-of-the-art AATAC algorithm with a few enhancements reducing computational complexity and improving results utility for DDoS detection. The *inline DDoS detector* provides service-tailored anomaly detection where each monitored entity is assigned with a configurable set of custom packet feature filters, including application layer DPI. The solution is decoupled into *inline probe(s)* and *analyser* elements facilitating high scalability and a wide range of deployment scenarios.

The *inline probes* provide link-rate real-time forwarding and network monitoring on the commodity hardware thanks to the utilisation of the DPDK framework and parallel packet processing on multiple CPU cores. The *inline probes* export per-entity traffic features status in the form of snapshots at regular intervals. The *analyser* aggregates this information from different *inline probes.* Based on a number of the most recent snapshots, it detects attacks (anomalies) using statistical analysis and handles corresponding alarms.

The *inline DDoS detector* has been implemented for the feasibility study, focusing on performance evaluation. Our studies proved that the *inline DDoS detector* is capable of real-time network traffic monitoring and near-real-time anomaly detection.

VII. FUTURE WORKS

This paper proved the feasibility of our *inline DDoS detector* for detecting and alarming about DDoS attacks in near real-time. An alarm informing of a susceptible DDoS attack is raised when an anomaly level exceeds the client-defined threshold. Note that the type of attack is not automatically

determined. This identification must be done by an administrator based on the real-time monitoring statistics of different traffic features. This task is not trivial, as: 1) up to 23 traffic features may be monitored (18 global features and 5 histogram features) in each direction for each client's service, and 2) each feature is described by the density and the corresponding anomaly level. The density carries the information of smothered traffic rate with a specific feature, while the anomaly level allows to detect rapid changes in this rate easily. Both factors should be used simultaneously to improve detection accuracy. Consequently, the *inline DDoS detector* may be further significantly enhanced with automatic identification of DDoS attack type to facilitate automatic attack mitigation.

Moreover, our *inline DDoS detector* enables to assign a different monitoring policy (defining a set of packet feature filters) for each client's service. Nevertheless, it is not obvious how this policy should be constructed for different applications. Specifically, which packet feature filters should be enabled to detect attacks against the most popular Internet services. We plan to address both of the above topics in our future works.

REFERENCES

[1] A. Zand, G. Modelo-Howard, A. Tongaonkar, S. -J. Lee, C. Kruegel and G. Vigna, "Demystifying DDoS as a Service," in IEEE Communications Magazine, vol. 55, no. 7, pp. 14-21, July 2017, https://doi.org/10.1109/MCOM.2017.1600980

[2] J. J. Santanna et al., "Booters — An analysis of DDoS-as-a-service attacks," 2015 IFIP/IEEE International Symposium on Integrated Network Management (IM), 2015, pp. 243-251, https://doi.org/10.1109/INM.2015.7140298

[3] Apache documentation, ServerLimit Directive [online]. Available from: https://httpd.apache.org/docs/2.4/mod/mpm_common.html#serverlimit [Accessed 21.10.2022]

[4] M. Sikora, T. Gerlich and L. Malina, "On Detection and Mitigation of Slow Rate Denial of Service Attacks," 11th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT), 2019, pp. 1-5, https://doi.org/10.1109/ICUMT48472.2019.8970844

[5] H. Kaur, S. Behal and K. Kumar, "Characterisation and comparison of Distributed Denial of Service attack tools," 2015 International Conference on Green Computing and Internet of Things (ICGCIoT), 2015, pp. 1139-1145, https://doi.org/10.1109/ICGCIoT.2015.7380634

[6] G. Roudière and P. Owezarski, "A lightweight snapshot-based DDoS detector," in Proc. of 2017 13th International Conference on Network and Service Management (CNSM), 2017, pp. 1-7, https://doi.org/10.23919/CNSM.2017.8256014

[7] J. Wang, R. C. . -W. Phan, J. N. Whitley and D. J. Parish, "Augmented Attack Tree Modeling of Distributed Denial of Services and Tree Based Attack Detection Method," 2010 10th IEEE International Conference on Computer and Information Technology, 2010, pp. 1009-1014, https://doi.org/10.1109/CIT.2010.185

[8] Y. -C. Wu, H. -R. Tseng, W. Yang and R. -H. Jan, "DDoS Detection and Traceback with Decision Tree and Grey Relational Analysis," 2009 Third International Conference on Multimedia and Ubiquitous Engineering, 2009, pp. 306-314, https://doi.org/10.1109/MUE.2009.60

[9] A. Saied, R. E. Overill, and T. Radzik, "Detection of known and unknown DDoS attacks using Artificial Neural Networks," Neurocomputing, vol. 172, January 2016, pp. 385–393, https://doi.org/10.1016/j.neucom.2015.04.101

[10] X. Qin, T. Xu and C. Wang, "DDoS Attack Detection Using Flow Entropy and Clustering Technique," 2015 11th International Conference on Computational Intelligence and Security (CIS), 2015, pp. 412-415, https://doi.org/10.1109/CIS.2015.105

[11] S. Ramaswamy, R. Rastogi and K. Shim, "Efficient algorithms for mining outliers from large data sets," *ACM SIGMOD Rec.*, vol. 29, no. 2, pp. 427-438, 2000.

[12] R. Hofstede, V. Bartoš, A. Sperotto and A. Pras, "Towards real-time intrusion detection for NetFlow and IPFIX," Proceedings of the 9th International Conference on Network and Service Management (CNSM 2013), 2013, pp. 227-234, https://doi.org/10.1109/CNSM.2013.6727841

[13] G. Roudière and P. Owezarski, "Evaluating the Impact of Traffic Sampling on AATAC's DDoS Detection" in Proc. of the 2018 Workshop on Traffic Measurements for Cybersecurity (WTMC '18). Association for Computing Machinery, New York, NY, USA, 27–32. https://doi.org/10.1145/3229598.3229605

[14] M. Jin, C. Wang, P. Li and Z. Han, "Survey of Load Balancing Method Based on DPDK," 2018 IEEE 4th International Conference on Big Data Security on Cloud (BigDataSecurity), IEEE International Conference on High Performance and Smart Computing, (HPSC) and IEEE International Conference on Intelligent Data and Security (IDS), 2018, pp. 222-224, https://doi.org/10.1109/BDS/HPSC/IDS18.2018.00054

[15] Information about the TAMA project, Exatel webpage [online]. Available from: https://exatel.pl/en/research-and-development/exatel-tama/ [Accessed 21.10.2022]

[16] S. Bradner, and J. McQuaid, "Benchmarking Methodology for Network Interconnect Devices", RFC 2544, https://doi.org/10.17487/RFC2544, March 1999

[17] S. Bradner, "Benchmarking Terminology for Network Interconnection Devices", RFC 1242, https://doi.org/10.17487/RFC1242 , July 1991