Detecting Web Application DAST Attacks with Machine Learning

Pojan Shahrivar
Rapid7 LLC
Boston, USA
pojan_shahrivar@rapid7.com

Stuart Millar
Rapid7 LLC
Boston, USA
stuart_millar@rapid7.com

Ezzeldin Shereen
KTH Royal Institute of Technology
Stockholm, Sweden
eshereen@kth.se

***Abstract*—Dynamic application security testing (DAST) scanning consists of automated requests to web applications with the goal of uncovering exploitable vulnerabilities. While the legitimate use of scanners aids development teams in improving security postures, they are often used by malicious actors in a brute-force manner for attack reconnaissance with a view to eventual compromise. Despite this threat from misuse of DAST to web applications and the critical data they handle, security mechanisms are lacking, with threshold-based classifiers suffering from being overly sensitive, causing excessive false positives. To tackle the problem, this paper demonstrates the first application of machine learning to specifically detect DAST attacks that augments a next-generation web application firewall implementing OWASP's AppSensor framework. Avoiding the brittle threshold approach and using tumbling windows of time to generate aggregated event features from source IPs, twelve random forest models are trained on millions of real-world events. Results show an optimal window size of 60 seconds achieves an F1 score of 0.94 and a miss rate of 6% on average across three production-grade web applications.**

***Index Terms*—dynamic application security testing, vulnerability scanning, random forest, machine learning, web application firewall, web application security**

I. Introduction

The growth of the cloud has caused the ubiquity of web applications offering rich platform-agnostic experiences across the likes of webmail, e-commerce, banking and social media. The volume of apps and society's dependency on them creates a major incentive for cybercriminals with a significant increase in web application attacks [1]. The OWASP Top Ten [2] is a highly valuable consensus list of the most critical web application vulnerabilities, such as broken access control and injections. In practice, however, protecting application servers is inherently difficult since they have to serve legitimate requests whilst simultaneously denying access to malicious ones. The distinction between a malicious and legitimate request can be hard to make, and the cost of blocking legitimate traffic can be high. With web attacks being the most common form of compromise [3], there is a pressing need for defensive capabilities to protect businesses, consumers and governments.

Dynamic application security testing (DAST) gathers information on a web app's potential vulnerabilities by sending a variety of HTTP requests in a brute-force manner through automated pen-testing techniques, which can then be remediated in the software development lifecycle. There is a valid global commercial industry for these tools, plus some are freely open-sourced [4]–[6]. However, DAST scanners are also used maliciously by both skilled and unskilled attackers. The latter, "script kiddies", may use scanners before selling their findings to a more proficient actor [7]. Hence the failure to detect and block DAST activity could expose an organisation to more severe abuse by sophisticated adversaries. Since the adversary's attack capacity is limited only by the available bandwidth and CPU, defensive systems need to prevent DAST scanning attacks with minimal manual intervention. In a production environment with multiple apps and millions of events, it is not feasible to check each attack alert by hand. With traditional detectors based on statistical thresholds suffering over-optimisation and excess false positives, it is essential to develop new protection mechanisms against malicious DAST activity.

An effective existing approach to securing web apps more generally is to run separate, dedicated security solutions in a layered fashion to filter incoming traffic. One such protection mechanism is a web application firewall (WAF). A WAF operates in the application layer and protects the web app by analyzing each HTTP request and then filtering, monitoring and blocking malicious traffic. WAFs have evolved into next-generation WAFs (NGWAFs) to include sophisticated event analysis engines that detect and prevent other malicious attacks like credential stuffing and distributed denial of service (DDoS). This motivates us to propose an automated machine learning (ML) classifier to augment an NGWAF and block actors performing DAST scanning attacks whilst avoiding the arguably arbitrary adjustments of threshold-based methods.

Our contributions are:

- A method of pre-processing millions of AppSensor events in a temporal fashion using a tumbling window approach to create a proprietary dataset.
- A random forest ML model with an optimal window size $w = 60$ seconds achieving an F1 score of 0.94 and a miss rate of 6% in detecting DAST attacks on average across three production-grade web apps.

The rest of this paper is as follows: Section II contains related work, Section III presents the methodology and Section IV the experimental details. Section V contains results and lastly Section VI outlines conclusions and future work.

II. Related Work

A. Traditional Threshold-based Detection Approaches

Previous threshold-based models for detecting malicious activity were built by modelling client behaviour using statistical models, heuristics and rules for anomaly detection [8]–[11]. For example, a model classifying login attacks could incorporate set thresholds for the hourly number of attempts at inputting credentials and the number of IP addresses in use from an actor. Thresholds could be dynamically generated too, such as a moving average of the number of IPs per user.

Seminal work from [12] presented a correlation between anomalous activity and misuse, using statistical techniques to describe and evaluate user behaviour against both fixed and dynamic measures of normality. This highlighted the importance of wider network security, encouraging other researchers to investigate the topic further. [13] proposed an intrusion detection system (IDS) that detected attacks against web servers and web apps by analyzing the statistical characteristics of HTTP traffic and parameters contained in queries between clients and servers. [8] applied Markov chains to create an intrusion detection mechanism for the web, with [9] presenting an anomaly detection model for HTTP traffic to and from a web application, where the app's normal behaviour was described using statistical properties.

These threshold-based systems may be straightforward to implement however actually choosing appropriate thresholds is often not clear, as they will almost certainly differ from one web app to the next. Moreover, there are challenging cases requiring multivariate thresholds. For example, if a username is utilised across different countries, but with the same browser fingerprint, one may draw the conclusion a VPN has been employed and the activity is not suspicious. However to remain effective the threshold-based system then additionally needs to take the number of unique browser fingerprints into account. So it can be seen as complexity increases, heuristics become harder to implement and more manual interventions may be needed.

The selection of an appropriate threshold is further influenced by resultant noise. If a threshold is too low, the detection rate is increased at the cost of time-consuming false positives, whilst setting a threshold too high risks missing some malicious activity altogether. These factors combined make it difficult for both individual users and WAF/NGWAF vendors to create rules and thresholds that generalize well enough. By contrast, our proposed ML classifier does not use thresholds, but rather temporal features indicative of DAST scanning attacks based on tumbling time windows. Moreover, there have been numerous works on detection attacks at the network layer, but work related specifically to the application layer is limited and not as mature [14].

B. Machine Learning Detection approaches

The popularisation of machine learning techniques led researchers to create unsupervised anomaly-based network intrusion detection models [15]–[18]. [15] used self-taught learning, a deep learning-based technique, on NSL-KDD [19], a dataset suggested to solve some of the inherent problems of the KDD'99 [20] dataset. [16] proposed a non-symmetric deep autoencoder (NDAE) for unsupervised feature learning and also proposed a model constructed using stacked NDAEs that was evaluated using the benchmark KDD'99 and NSL-KDD datasets. [17] performed a study on the effectiveness of various ML methods in detecting future cyberattacks, with experiments performed using both classic ML algorithms and neural networks, again using benchmark datasets such as KDD'99 and NSL-KDD. [21] sought to evaluate the feasibility of unsupervised and semi-supervised approaches for web attack detection [21], [22]. The data used was based on a monitoring tool characterizing the behaviour of the web application by extracting traces of program execution from running software. The monitoring model was created by using supervised training with the test suites created for an application as part of the software development process. [23] proposed using an ML approach to model the normal behaviour of applications and to detect cyberattacks, however unfortunately it was not clear to what degree ML was leveraged in the solution, with regular expressions and dynamic programming being used.

We feel leveraging ML in an enterprise application security setting is under-researched with plenty of scope. To the best of our knowledge, there is no work directly addressing the detection of DAST vulnerability scanning attacks, likely due to the absence of relevant web application datasets and the expensive domain knowledge required. This creates an opportunity for us to contribute to the space by taking advantage of a large, real-world dataset to address the research gap in investigating the specific detection of DAST attacks at the application layer using ML.

III. Methodology

This work presents an ML classifier to detect malicious DAST activity that augments an NGWAF as part of a wider end-to-end system. In practice, source IPs can then be blocked to prevent further scanning. IP activity is classified as either positive, deemed to originate from a DAST scanner, or negative, where it does not come from a scanner. The number of false positive detections should be minimised to prevent legitimate requests from being refused. This section discusses curation of raw event data, an overview of the NGWAF plus the concept of tumbling windows, data analysis and feature engineering. Further, to define the research parameters the following assumptions are made:

- the adversary is an unskilled individual who lacks the ability to write their own custom payloads; hence they choose to utilise a DAST scanner maliciously.
- the attacks are not distributed and stem from the same source IP.
- the DAST scanner is running the default off-the-shelf configurations and is not using any custom APIs, nor is it loaded with any extensions.

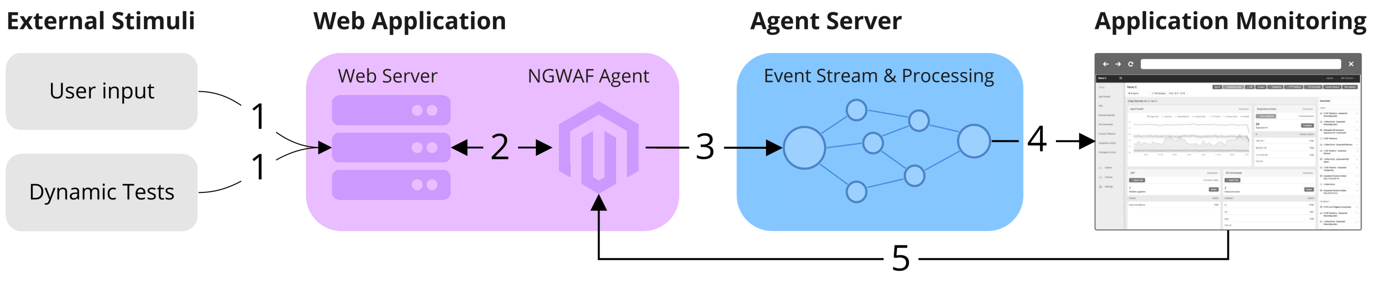

Fig. 1: NGWAF overview

Feature	Description
`ts`	event timestamp
`ip`	source IP
`response-code`	e.g. 200, 404, etc
`payload`	e.g. `OR 1=1`
`method`	e.g. POST, GET, etc
`headers`	headers in the request
`parameter`	parameter set in the url
`sensor`	sensor which was triggered
`pattern`	pattern for the sensor triggered
`location`	body, cookie, header or query
`uri`	e.g. `https://www.space-travel.com/api/saturn/`

TABLE I: AppSensor event features

A. Data Curation

The AppSensor framework is an industry standard from OWASP [24] for implementing application layer intrusion detection and automated response. For the avoidance of doubt, the AppSensor framework is defensive in aiming to detect malicious actors, not offensive with regards to discovering vulnerabilities in a web app. The framework is used in the NGWAF via a series of detection points, or sensors, triggered by suspicious activity. For example, attackers can send an SQLi payload, try to circumnavigate authentication, malform requests with odd encodings or perform credential stuffing, amongst others. The nature of these activities triggers a regular expression pattern within the relevant sensor which creates a timestamped AppSensor event. Table I lists the features of an AppSensor event. There can be millions of such events in a 24-hour period, forming a reliable source of timely real-world raw data to be leveraged.

Per Figure 1, the NGWAF contains a monitoring agent, an event analysis engine and an application monitoring user interface for an overview of threats. After external stimuli in step 1, steps 2-5 depict the process of an event being logged, processed and finally a new security policy emitted. The event analysis engine consumes events generated by the agent and uses policy scripts to automate actions in response to the observed signals. The agent is instrumented into an application or its run-time environment and monitors, detects and prevents attacks in real time.

The policy scripts are designed by administrators, based on information in the application monitoring interface. Dependent on the policy, the engine decides how communications should be handled including allowing, reporting or blocking an IP. In the current configuration, IPs executing DAST activity can be blocked after the fact by the administrator via reviewing the user interface. However, with the potential volume of events, this is time-consuming and expensive, causing cognitive overload and possible alert fatigue.

Sensor	Description
`cmdi`	command injection
`excsrf`	cross-site and anti-request forgery tokens
`exsql`	SQL exceptions
`fpt`	file path traversal
`null`	embedding null code
`reqsz`	unusual request size
`retr`	line-break character
`rspsz`	unusual response size
`s4xx`	all response codes 4xx
`s5xx`	all response codes 5xx
`sqli`	SQL injection
`uaempty`	user agent empty
`xss`	cross-site scripting
`xxe`	XML external entity processing

TABLE II: Sensors implemented in the NGWAF

In line with the AppSensor framework, each category of attack has multiple detection points in the NGWAF acting as sensors, listed in Table II. The NGWAF uses deep packet inspection (DPI) to implement the AppSensor framework, collecting information about activity originating in the application layer. DPI enables the request and response filtering capabilities of suspicious activity in web apps by inspecting the strings present in HTTP requests and matching their patterns against a wide range of attacks.

When a sensor is triggered, an AppSensor event is generated and emitted into the event stream, referred to as the AppSensor stream. The streams of our three busiest production web apps were collected and persisted within a 48-hour time period, detailed further in Section IV-A.

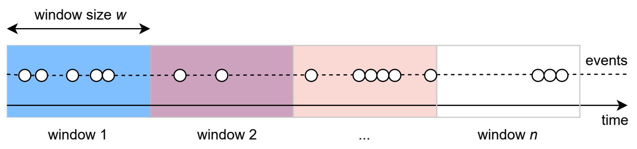

Fig. 2: Dividing time into tumbling windows

B. Tumbling Windows

DAST scanners operate in a conspicuously overt brute-force manner. Some of the most distinct patterns are the request rate and the diversity of requests from a source IP. To take advantage of this and track the rate at which requests are performed, we need to perform aggregation of events over time, forming tumbling windows. By first grouping events together originating from the same source IP address, numerical aggregates can then be created for many features such as intensity of requests, diversity of headers, distribution of response codes and diversity of insertion points.

Per Figure 2, a source IP's events are grouped together by bucketing them into tumbling windows of time that do not overlap. For each IP, features are calculated within each window across the IP's events to form the feature vectors used in the training process. These resulting feature vectors are the individual samples that make up each app's dataset, explained further in Sections III-E and IV-A. The size of the tumbling window impacts storage requirements, processing time, and compute costs in production. Therefore striking a balance between these factors and model performance should be considered in a real-world environment. In this study, a window size $w \in \{1, 30, 60, 300\}$, measured in seconds, is used to create different datasets to be compared and evaluated.

C. Data Labelling

The AppSensor events consist of all matches against the attack sensors and patterns, which are then aggregated into tumbling windows per source IP. However each window needs labelled as to whether it belongs to a scanner or not. This is highly challenging to do by hand given the volume of windows. Therefore the method of labelling the data is based on analysis and comparison between generated data and production data.

We generate data by deploying WebGoat [25], a deliberately insecure web app to demonstrate common server-side application flaws with many vulnerabilities which can be exploited, thus providing the coverage that is needed [2], [25]. WebGoat is then scanned with two DAST tools - OWASP Zed Attack Proxy [6] and Burp Scanner by PortSwigger [5]. Both of these widely used scanners are included in the pen-testing platform Kali Linux, an open-source distro specifically for pen-testing and security research [4]. The data analysis technique employed is EDA [26], used to understand the data, its shape, the types of features and the relationship and patterns between them. The results are then used to verify intuitions about the characteristics of scanning.

The labelling of a source IP's aggregated events within a given window is determined by combining insights gained from the data analysis with manual classification by a human expert. This allows the creation of a set of rules to label activity as originating from a DAST scanner or not. The WebGoat data was manually labelled by a human expert who had access to a dashboard containing the statistics for each source IP session, for example the frequency distribution of sensor activation, and rates of requests being made, plus the label predicted by the ruleset. The expert corrected the labels by hand where needed and the generalised rules were then used to label real-world production data gathered from the AppSensor stream.

D. Data Analysis

Salient input features to the ML model are key in allowing the classifier to detect DAST scanning activity. Given we are adopting a tumbling window approach that aggregates features, before deciding how to engineer these features it is useful to analyse the data in combination with our DAST domain knowledge. Note, where relevant in this subsection, the figures use logarithmic axes and are for a single app. We find similar trends across the other apps in our dataset also, though their figures are not included due to space restrictions.

1) Number of unique URIs per source IP: Figure 3 shows the empirical cumulative distribution function (eCDF) for the number of distinct URIs requested per IP. Scanning attacks with a single distinct URI requested can be attributed to single-page scans. In this case, the URI is constant, with the payload changing for each request made. The large number of distinct URIs for non-scanning attacks is due to web crawlers tending to trigger the `s4xx` sensor rather than containing a malicious payload. With a clear difference between the attack and non-attack classes, it is logical to keep count of the distinct i.e. unique URIs within an aggregated tumbling window.

2) Number of payloads per session: Figure 4 displays the eCDF for the number of distinct payloads for each session. The large proportion of attack sessions that only have a small number of requests can be attributed to a number of requests being made at a very high pace within a very short time. Again since the difference between the attack and non-attack classes is clear, a count of the unique payloads, and total payloads, will be calculated within a tumbling window.

3) Distribution of HTTP methods across events: An interesting observation can be made in Figure 5 where only the attack class has any matches for the HTTP methods `PROPFIND` and `TRACK`. These two methods can be abused for injection and extraction of sensitive information [27], [28]. Given the differences in the plot, we decide a count of each HTTP method in a tumbling window may be a useful feature.

4) Event intensity per class: Figure 6 is a scatter plot to show the joint distributions of intensity in events per second and duration of each session. The longest and most intense sessions are in the top right corner, where the red points for the DAST scanner attack class are concentrated. There are no points in the bottom left corner since duration and intensity have a negative linear relationship. The observation

Feature	Description
`timestamp`	event timestamp
`ip`	source IP
`payload`	extracted payload
`method`	extracted method
`headers.known`	extracted known headers
`headers.unknown`	extracted unknown headers
`sensor`	extracted sensor
`pattern`	extracted pattern
`location`	extracted location
`uri`	extracted URI

TABLE III: Makeup of a single event

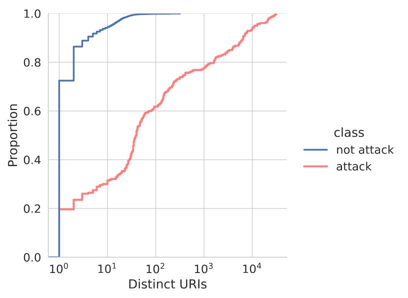

Fig. 3: eCDF for distinct URIs for `app-1`

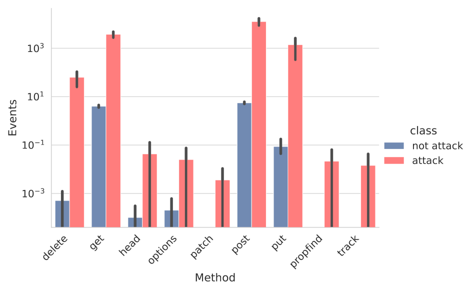

Fig. 5: HTTP method distribution for `app-1`

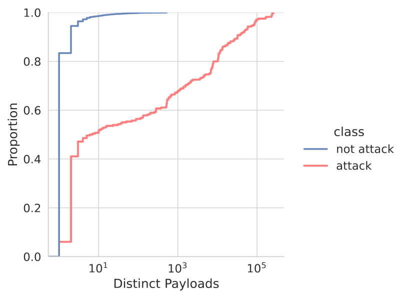

Fig. 4: eCDF for distinct payloads for `app-1`

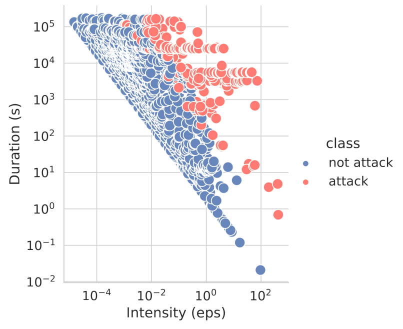

Fig. 6: Intensity and duration distributions for `app-1`

can be made DAST attacks are longer in duration and have a greater intensity, justifying more widely the use of an aggregated time window approach. We assert a DAST attack will generate many more events than non-DAST activity, and thus aggregated features may be more discriminative.

E. Feature Engineering

Each sample in the dataset is an aggregate representation per source IP of individual events within a tumbling window, labelled as either a DAST attack or non-DAST activity. First, a representation is required for a single event, and then a second representation is needed as to how those single events for a given source IP will be aggregated. Table III lists the extracted AppSensor features comprising a single event, with Table IV describing the aggregations to produce the overall feature vector for all events from a given source IP in a window. Note the source IP itself is not part of the final feature vector; it is only used as an identifier.

Numerical features are created using standard summation or counts with min-max scaling [29], though the text features require a different approach. Intuitively, if we can capture a representative set of text features from the training data for DAST attacks and use them as part of the ML training process, the discriminative power in testing may be strengthened. For example, a DAST scanner generates a great number of payloads unique to itself which may aid identification of the scanning attack.

For the corpora of the `payload`, `headers.known`, `headers.unknown` and `URI` text features, the TF-IDF score [30], [31] was calculated to produce a ranking of importance within the training data. It was capped at 100 terms, meaning it builds a vocabulary only containing the 100 most important terms ordered by term frequency across a given corpus. The intent is to highlight unique strings in these features from DAST scanners while ignoring the commonalities. Within a given window, for each of the `payload`, `headers.known`, `headers.unknown` and `URI` features, a count of these top 100 terms can then be calculated.

Feature	Aggregation	Description
ip	N/A	source IP whose events are being aggregated, only used as an identifier, not in model training
payload	value counts	for each of the top 100 training set payloads, a count of their occurrences in the window
payload	nunique	# of unique payloads in the window
method	value counts	for each method, a count of their occurrences in the window
sensor	value counts	for each sensor, a count of how often each is triggered in the window
pattern	value counts	for each pattern, a count of how often each is triggered in the window
location	value counts	for each location, a count of their occurrences in the window
headers	count	total # of headers in the window
headers.known	count	# of known headers in the window
headers.known	nunique	# of unique known headers in the window
headers.known	value counts	for each of the top 100 training set known headers, a count of their occurrences in the window
headers.unknown	count	# of unknown headers in the window
headers.unknown	nunique	# of unique unknown headers in the window
headers.unknown	value counts	for each of the top 100 training set unknown headers, a count of their occurrences in the window
uri	value counts	for each of the top 100 training set URIs, a count of their occurrences in the window
uri	nunique	# of unique URIs in the window
overall	N/A	final feature vector of 469 elements, excluding source IP

TABLE IV: Makeup of an aggregated window of events per source IP

w	App	# train DAST samples	# train non-DAST samples	# test DAST samples	# test non-DAST samples
1	app-1	129,941	129,941	1,265,655	58,605
	app-2	183,333	183,333	409,687	78,334
	app-3	7,704	7,704	3,287	3,303
30	app-1	105,458	105,458	1,144,586	44,805
	app-2	135,662	135,662	244,617	58,200
	app-3	7,714	7,714	4,177	3,303
60	app-1	93,532	93,532	1,139,166	39,810
	app-2	133,283	133,283	230,000	57,042
	app-3	7,707	7,707	4,187	3,303
300	app-1	76,583	76,583	1,138,835	33,348
	app-2	128,479	128,479	155,857	55,110
	app-3	7,704	7,704	3,287	3,303

TABLE V: Train and test splits per window size w across each app

F. *Random Forest ML Model*

The random forest algorithm is used to train each model, inspired by its use outside web application security in areas such as network IDS [32]–[34]. One model is trained per app with a varying window size w. In testing a model makes a binary prediction as to whether the aggregated window activity of a source IP is a DAST attack or not.

IV. Experimental details

A. *Dataset*

The data used in our experiments is collected from the three busiest real-world web applications protected by the NGWAF over a 48-hour period. This raw data covers the range of sensors resulting in a high quality, high volume dataset where app-1 had 49,033,476 events, app-2 had 27,328,196 events and app-3 had 18,113,350 events. Figure 7 shows an example distribution for app-1, with distinctly taller bars representing attacks much more common than the rest, like SQLi and XSS. Note the y-axis is logarithmic meaning the tallest bars are several orders of magnitude greater than others, indicating that as expected considerably more DAST scanning attacks than non-scanning attacks take place in the wild.

The three apps' events are consumed from the AppSensor stream simultaneously, meaning the same version of attack detection patterns is used, ensuring consistency in what is considered an attack. For confidentiality reasons, we cannot reveal details of the functionality or nature of the apps.

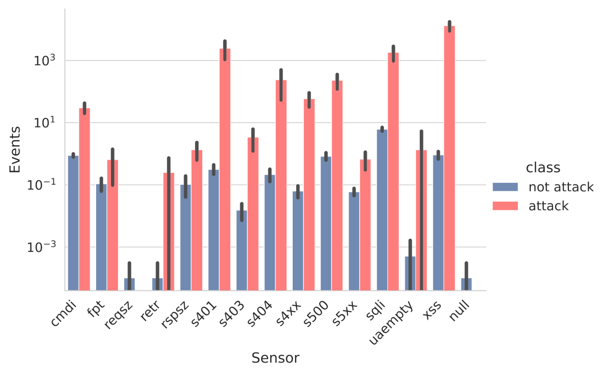

Fig. 7: Event distribution per sensor for app-1

Occasionally duplicate events can be captured in the stream, so the data is also deduplicated. We gather our own proprietary data to give us more confidence in any eventual outcomes, as previous datasets are not AppSensor-orientated.

B. *Dataset Splits*

Table V shows the number of DAST and non-DAST samples per app in training and testing. Recall each sample is an aggregation of a given source IP's activity within a tumbling window, labelled as either a DAST scanning attack or non-DAST activity. To ensure a balanced number of each class per training split, the DAST scan samples are subsampled.

The data for each app was first sorted by timestamp and then split into a training split and a test split, after which the tumbling windows were generated. The training split includes all the data from the first 24-hour collection period, and the testing split all the data from the second 24-hour collection period. This ensures a challenging scenario where a model can be trained using the first 24-hour period and tested on the future second 24-hour period to mirror a real-world deployment. Moreover, depending on the configuration, a vulnerability scan attack can span anything from minutes to an hour or two. Randomly sampling events may create a train and test split containing the same sessions, risking the possibility of a scan being included in both the training and testing splits, leading to artificially inflated performance. After the event data is split, the steps outlined in Section III-E are executed to generate each source IP's aggregations within a tumbling window across each $w \in \{1, 30, 60, 300\}$.

C. Classification Metrics

The metrics precision, recall, F1 and miss rate are calculated, with a focus on the F1 and miss rate, which is the percentage of scanning attack samples missed and mistakenly predicted as non-scanning. The positive case is a DAST scanning attack and the negative case is non-DAST activity. Formally:

$$Precision = \frac{TP}{TP + FP} \tag{1}$$

$$Recall = \frac{TP}{TP + FN} \tag{2}$$

$$F_1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} \tag{3}$$

$$Miss\,rate = 100 \cdot (1 - Recall) \tag{4}$$

The receiver operating characteristic (ROC) curve is also measured for the classification at various probability thresholds. The ROC is a probability curve where the area under the curve (AUC) represents the degree of separability to describe how well the model distinguishes between DAST activity and non-DAST activity.

D. Random Forest Hyperparameters

The random forest model parameters are the defaults per [35]. Additionally, the following software was used: Jupyter, Python 3.9, scikit-learn, pandas [36] and numpy [37].

V. Results

For each window size $w \in \{1, 30, 60, 300\}$, three models were evaluated, one per app, totalling twelve models. Each model was trained with its training data split and tested with its corresponding test split, listed previously in Table V. Having the train and test data fixed as two separate 24-hour periods means any change in model performance can be attributed to varying w. Results in Table VI show strong performance across each window size in the first instance. Specifically, $w = 60$ performs best in detecting DAST activity with an F1 score of 0.94 and a miss rate of 6% on average

w	App	Precision	Recall	F1	Miss rate
1	app-1	0.97	0.95	0.96	5%
	app-2	0.89	0.82	0.84	18%
	app-3	0.96	0.96	0.96	4%
	Average	0.94	0.91	0.92	9%
30	app-1	0.98	0.96	0.97	4%
	app-2	0.92	0.89	0.89	11%
	app-3	0.96	0.96	0.96	4%
	Average	0.95	0.94	0.94	6.33%
60	app-1	0.98	0.97	0.97	3%
	app-2	0.92	0.89	0.90	11%
	app-3	0.96	0.96	0.96	4%
	Average	**0.95**	**0.94**	**0.94**	**6%**
300	app-1	0.97	0.85	0.90	15%
	app-2	0.90	0.86	0.87	14%
	app-3	0.96	0.96	0.96	4%
	Average	0.94	0.89	0.91	11%

TABLE VI: Performance metrics per w across each app

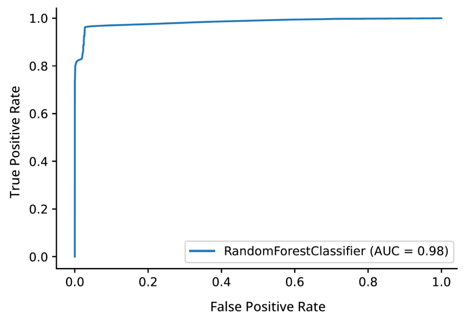

Fig. 8: ROC curve of the model for app-1 with $w = 60$

across all three apps. In practice, $w = 30$ might also be an option in a production setting, though it is recognised w could be varied further, perhaps in 5-second increments, which unfortunately was not possible for this work due to time restrictions. Notwithstanding more exhaustive window tuning, the results are still attractive and it could be said even the smallest window choice of $w = 1$ may be effective in a resource-constrained environment.

Performance can be further demonstrated by the ROC curve, for example from the model for app-1 with $w = 60$ in Figure 8. It plots the true positive rate, i.e. the proportion of correctly predicted DAST attacks, against the false positive rate, i.e. the proportion of DAST attack activity incorrectly predicted as non-DAST. The optimal point is the top left corner with a true positive rate of 1 and a false positive rate of 0. ROC curve steepness is important in maximising the true positive rate while minimizing the false positive rate, and a high AUC implies the model is performant predicting positive and negative samples alike, with a model having an AUC near 1 reaching a maximized measure of separability as the curve approaches the top left corner. Indeed, we can observe in Figure 8 for app-1 the AUC = 0.98 and the curve is almost optimal.

Prior analysis in Section III-D showed DAST attacks last longer, with greater intensity, distinct payloads and URIs, producing a vast number of events. We assert encoding this knowledge via careful feature engineering contributed to the outcome. In practice, an extra step could be added to the

NGWAF to automatically block source IPs of DAST activity without manual intervention, making our proposed end-to-end system appealing in a production setting.

Lastly, we recognise models need updated over time. This is a relatively manageable task of collecting fresh data and retraining at a given cadence, such as once a week. Nevertheless, for these selected three apps our use of a high-quality proprietary dataset gives us strong confidence in the ML model's ability to detect DAST scanning attacks from unskilled individuals operating from a single IP.

VI. Conclusions and Future Work

A random forest ML model detecting DAST vulnerability scanning attacks that augments an existing NGWAF is presented. This solution can form part of an end-to-end streaming data pipeline, classifying DAST activity which can be blocked. Leveraging a large real-world dataset of millions of events, results show an optimal window size of 60 seconds achieves an F1 score of 0.94 and a low miss rate of 6% on average across three selected enterprise-grade production apps. Future work may consider creating a suite of ML models, one per sensor, and explore sequence-orientated deep learning like transformers, recurrent networks and convolutions.

References

[1] "Enisa threat landscape 2020 - web application attacks," Tech. Rep., Feb 2021. [Online]. Available: https://www.enisa.europa.eu/publications/web-application-attacks

[2] "Owasp top ten." [Online]. Available: https://owasp.org/www-project-top-ten/

[3] S. Millar, D. Podgurskii, D. Kuykendall, J. Martínez del Rincón, and P. Miller, "Optimising vulnerability triage in dast with deep learning," in *Proceedings of the 15th ACM Workshop on Artificial Intelligence and Security*, ser. AISec'22. New York, NY, USA: Association for Computing Machinery, 2022, p. 137–147.

[4] "Kali tools: Kali linux tools," March 2023. [Online]. Available: https://www.kali.org/tools/

[5] "Burp scanner - web vulnerability scanner from portswigger." [Online]. Available: https://portswigger.net/burp/vulnerability-scanner

[6] "Owasp zed attack proxy (zap)." [Online]. Available: https://www.zaproxy.org/

[7] T. S. Hyslip, *Cybercrime-as-a-Service Operations*. Cham: Springer International Publishing, 2020, pp. 815–846.

[8] A. Perez-Villegas, C. Torrano-Gimenez, and G. Alvarez, "Applying markov chains to web intrusion detection," *Proceedings of Reunión Espanola sobre Criptología y Seguridad de la Información (RECSI 2010)*, pp. 361–366, 2010.

[9] C. Torrano-Giménez, A. Perez-Villegas, and G. Alvarez Maranón, "An anomaly-based approach for intrusion detection in web traffic," 2010.

[10] C. Torrano-Gimenez, A. Perez-Villegas, and G. Alvarez, "A self-learning anomaly-based web application firewall," in *Computational Intelligence in Security for Information Systems*. Springer, 2009, pp. 85–92.

[11] C. Torrano-Gimenez, A. Perez-Villegas, G. Alvarez, E. Fernández-Medina, M. Malek, and J. Hernando, "An anomaly-based web application firewall." in *SECRYPT*, 2009, pp. 23–28.

[12] D. E. Denning, "An intrusion-detection model," *IEEE Transactions on software engineering*, no. 2, pp. 222–232, 1987.

[13] C. Kruegel and G. Vigna, "Anomaly detection of web-based attacks," in *Proceedings of the 10th ACM conference on Computer and communications security*, 2003, pp. 251–261.

[14] A. Saha and S. Sanyal, "Application layer intrusion detection with combination of explicit-rule- based and machine learning algorithms and deployment in cyber- defence program," *CoRR*, vol. abs/1411.3089, 2014. [Online]. Available: http://arxiv.org/abs/1411.3089

[15] A. Javaid, Q. Niyaz, W. Sun, and M. Alam, "A deep learning approach for network intrusion detection system," *Eai Endorsed Transactions on Security and Safety*, vol. 3, no. 9, p. e2, 2016.

[16] N. Shone, T. N. Ngoc, V. D. Phai, and Q. Shi, "A deep learning approach to network intrusion detection," *IEEE transactions on emerging topics in computational intelligence*, vol. 2, no. 1, pp. 41–50, 2018.

[17] R. Vinayakumar, M. Alazab, K. Soman, P. Poornachandran, A. Al-Nemrat, and S. Venkatraman, "Deep learning approach for intelligent intrusion detection system," *IEEE Access*, vol. 7, pp. 41 525–41 550, 2019.

[18] C. Yin, Y. Zhu, J. Fei, and X. He, "A deep learning approach for intrusion detection using recurrent neural networks," *Ieee Access*, vol. 5, pp. 21 954–21 961, 2017.

[19] M. Tavallaee, E. Bagheri, W. Lu, and A. A. Ghorbani, "A detailed analysis of the kdd cup 99 data set," in *2009 IEEE symposium on computational intelligence for security and defense applications*. IEEE, 2009, pp. 1–6.

[20] "Uci machine learning repository: Kdd cup 1999 dataset." [Online]. Available: https://archive.ics.uci.edu/ml/datasets/kdd+cup+1999+data

[21] Y. Pan, F. Sun, Z. Teng, J. White, D. C. Schmidt, J. Staples, and L. Krause, "Detecting web attacks with end-to-end deep learning," *Journal of Internet Services and Applications*, vol. 10, no. 1, pp. 1–22, 2019.

[22] F. Sun, P. Zhang, J. White, D. Schmidt, J. Staples, and L. Krause, "A feasibility study of autonomically detecting in-process cyber-attacks," in *2017 3rd IEEE International Conference on Cybernetics (CYBCONF)*. IEEE, 2017, pp. 1–8.

[23] M. Choraś and R. Kozik, "Machine learning techniques applied to detect cyber attacks on web applications," *Logic Journal of the IGPL*, vol. 23, no. 1, pp. 45–56, 12 2014.

[24] "Project appsensor." [Online]. Available: https://owasp.org/www-project-appsensor/

[25] "Webgoat 8: A deliberately insecure web application." [Online]. Available: https://github.com/WebGoat/WebGoat

[26] J. W. Tukey *et al.*, *Exploratory data analysis*. Reading, Mass., 1977, vol. 2.

[27] "Webdav propfind method allows web directory browsing." [Online]. Available: https://www.rapid7.com/db/vulnerabilities/http-generic-propfind-dir-browsing/

[28] "Http track." [Online]. Available: https://www.rapid7.com/db/vulnerabilities/http-track-method-enabled/.

[29] "Scikit-learn: Minmaxscaler." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.preprocessing.MinMaxScaler.html

[30] K. Sparck Jones, "A statistical interpretation of term specificity and its application in retrieval," *Journal of documentation*, vol. 28, no. 1, pp. 11–21, 1972.

[31] S. Qaiser and R. Ali, "Text mining: use of tf-idf to examine the relevance of words to documents," *International Journal of Computer Applications*, vol. 181, no. 1, pp. 25–29, 2018.

[32] N. Farnaaz and M. Jabbar, "Random forest modeling for network intrusion detection system," *Procedia Computer Science*, vol. 89, pp. 213–217, 2016.

[33] P. Negandhi, Y. Trivedi, and R. Mangrulkar, "Intrusion detection system using random forest on the nsl-kdd dataset," in *Emerging Research in Computing, Information, Communication and Applications*. Springer, 2019, pp. 519–531.

[34] H. Alqahtani, I. H. Sarker, A. Kalim, S. M. Minhaz Hossain, S. Ikhlaq, and S. Hossain, "Cyber intrusion detection using machine learning classification techniques," in *Computing Science, Communication and Security*, N. Chaubey, S. Parikh, and K. Amin, Eds. Singapore: Springer Singapore, 2020, pp. 121–131.

[35] "Scikit-learn: Random forest classifier." [Online]. Available: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.RandomForestClassifier.html

[36] Pandas, "pandas-dev/pandas: Pandas," 2023. [Online]. Available: https://doi.org/10.5281/zenodo.3509134

[37] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, no. 7825, pp. 357–362, Sep. 2020.