# Conditional Network Attack Generation using GANs

Author: Thomas Schmied
TU Wien

Supervisor: Fares Meghdouri
TU Wien

Supervisor: Prof. Tanja Zseby
TU Wien

Supervisor: Prof. Thomas Gärtner
TU Wien

## Abstract

This report addresses the problem of conditional network attack generation using GANs: generating attacks of a specified attack type or with desired properties. We apply four established GAN architectures that originate from the domain of (conditional) image generation to generate synthetic network flows: CGAN, WGAN, AC-GAN, and AC-WGAN. We ensure the conditioning of the attack generation in two different ways. First, as is standard practice, we only condition on the network attack types (e.g., generate flows of attack type 'DDoS'). Our second approach then aims for a more fine-grained control over the generation process (e.g., generate flows of attack type 'DDoS' with a short 'Flow Duration' and high 'Packet Length Mean'). We evaluate the validity of generated network flows on three dimensions: (1) using an external classifier, (2) statistical tests on the distribution of generated features, (3) distance measurements of generated and real attacks. For all our experiments we use the CIC-IDS2017 dataset. GANs have been a focus of prior for network attack generation, but to the best of our knowledge this is the first paper that employs GANs for network attack generation while maintaining control over the generation process.

## 1 Introduction

Even though huge quantities of network data is generated each day due to the ubiquity of the Internet in our everyday lives, the amount of network traffic that can be used for research purposes is comparatively limited. In fact, the few publicly available datasets for intrusion detection have numerous issues, in particular artifacts if the traffic is artificially generated [1]. Nonetheless, the effectiveness on future network anomaly detection research depends on the availability of large and diverse network traffic datasets. Additionally, recent cyber-attacks on critical infrastructures, small/large/medium businesses, non-profit organizations, and even entire governments have demonstrated once again how vulnerable network connected software systems are. This shows that cyber-security has become a central concern in our interconnected global society. Intrusion Detection Systems (IDS) are typically responsible for detecting malicious traffic within a network. Recently, there has been a trend towards Machine Learning (ML) based intrusion detection systems and they rely on large-scale datasets of network-attacks.

Therefore, this paper aims to develop methods to generate malicious network traffic based on Generative Adversarial Networks (GANs) [2] in order to make anomaly detection methods more robust. Overall, we apply four established GAN architectures to generate synthetic network flows: Conditional GAN (CGAN) [3], Wasserstein GAN (WGAN) [4], Auxiliary-Classifier GAN (AC-GAN) [5], and Wasserstein GAN with Auxiliary Classifier (AC-WGAN). We also experimented with Wasserstein GAN with Gradient Penalty (WGAN-GP) [6] but the results were inconclusive with this architecture.

However, in practice we do not only want to generate network attacks, but network attacks with particular characteristics that we specify. Hence, we would also like to have full control over the generation process. In our experiments we control the attack generation in two different ways. First, as is standard practice, we condition on the network attack types. This makes it possible to generate flows of a particular attack type, e.g., flows that look like Distributed Denial of Service (DDoS) attacks. However, this approach only enables coarse control. Therefore, our second approach aims for a more fine-grained control over the generation process, by conditioning on a set of twelve features (e.g., 'Flow Duration' or 'Packet Length Mean') that the generated flow should or should not exhibit. This then not only allows to generate an attack of a particular attack type, but also an attack that should, for example, have a short 'Flow Duration' and high 'Packet Length Mean'. If we, for instance, observe that the ML-based IDS performs poorly on flows with low flow duration, we are able to generate a set of diverse flows that exhibit this feature to augment the training dataset. In general, this approach enables us to generate more diverse data samples to increase the robustness of the IDS. Both approaches are explained in more detail in Section 3.

We also have to ensure that the generated network attacks are as realistic as possible, otherwise their effect could be unfavourable. Therefore, we evaluate the validity of generated network flows on three dimensions: (1) using an external classifier, (2) statistical tests on the distribution of generated features, and (3) distance measurements between generated

and real attacks. This approach is similar to the evaluation procedure of prior work on GANs for network traffic [7–9]. We describe each evaluation method in more detail in Section 3.

For all our experiments we used the CIC-IDS2017 dataset [10], one of the most comprehensive network traffic datasets for intrusion detection and a primary focus of prior work [1, 11–13]. Overall, the dataset contains network packets and the respective network flows of 14 different attack types (15 if we include benign flows). Each flow consists of 79 features that the GANs aim to generate in our subsequent experiments. We give a more comprehensive description of the dataset in Section 3.

Overall, this paper is a first step towards generating realistic network attacks while retaining full control over the generation process.

| Attack Type | # of samples | Proportion |
|---|---|---|
| DoS Hulk | 231073 | 0.41437 |
| PortScan | 158930 | 0.28500 |
| DDoS | 128027 | 0.22958 |
| DoS GoldenEye | 10293 | 0.01846 |
| FTP-Patator | 7938 | 0.01423 |
| SSH-Patator | 5897 | 0.01057 |
| DoS slowloris | 5796 | 0.01039 |
| DoS Slowhttptest | 5499 | 0.00986 |
| Bot | 1966 | 0.00353 |
| Web Attack Brute Force | 1507 | 0.00270 |
| Web Attack XSS | 652 | 0.00117 |
| Infiltration | 36 | 0.00006 |
| Web Attack Sql Injection | 21 | 0.00004 |
| Heartbleed | 11 | 0.00002 |

**Table 1.** Number of samples per attack type in CIC-IDS2017

## 2 Background

In this section, we briefly discuss the theoretical background relevant for this work. First, we discuss ML approaches for intrusion detection. Then GAN architectures for network traffic generation are briefly reviewed.

**ML for Intrusion Detection.** The intrusion detection system is a central component of the cyber security arsenal in any secure network. As the name tells, its primary task is to help discover unauthorized intrusions into the network [14]. We distinguish two broad categories of intrusion detection systems: (1) signature-based IDS and (2) anomaly-based IDS [1]. Signature-based IDS, on the one hand, rely on pattern matching to detect (known) attacks. A variety of commercial, as well as open-source, signature-based IDS are available, the most prominent one being SNORT [15]. Even though signature-based IDS can provide strong protection against known threats, they have their limitation, most notably detecting unknown attacks. Anomaly-based IDS, on the other hand, are an attractive alternative as they have the capability to overcome the limitations of signature-based IDS. These methods learn a detection model from available network datasets using (e.g.) ML techniques. As they do not rely on signatures, they have the ability to detect zero-day attacks [1]. However, as all ML models, they rely on the availability of large and diverse datasets to learn the detection model.

**GANs.** A GAN is a type of neural network architecture that was originally developed for synthetic image generation [2]. The quality of images created with original architectures was low, and they were easily distinguishable from real images [16]. However, in recent years GAN-architectures, such as StyleGAN v2, have become more sophisticated and it is almost impossible to tell if they are real or fake [17]. Since their inception GANs have been applied to numerous other tasks, such as generating video, audio, text, tabular data, and even entire video games [18–23].

**GANs for network traffic generation.** The success of GANs has also attracted the attention of the cyber security research community. One application of GANs is to modify real malware in such a way that it mimics legitimate traffic and thus cannot be detected by the IDS [8]. Furthermore, GANs have been applied successfully for generating network traffic on a flow-based as well as packet-based level [7, 9, 24, 25].

## 3 Methodology

Next, we describe the methodology that we apply in this paper. First, we give a brief overview of the CIC-IDS207 dataset, show its aggregate statistics, and highlight a few notable shortcomings. Then we describe the model architecture that we employ in our subsequent experiments, as well as our two conditioning approaches. Finally, we detail our evaluation approach.

### 3.1 Dataset

The CIC-IDS2017 dataset contains network packets and the respective network flows of 14 different attack types: *Bot, DDoS, DoS GoldenEye, DoS Hulk, DoS Slowhttptest, DoS slowloris, FTP-Patator, Heartbleed, Infiltration, PortScan, SSH-Patator, Web Attack Brute Force, Web Attack Sql Injection*, and *Web Attack XSS*. Table 1 shows the number of samples and proportion for each class in the dataset. The dataset is highly imbalanced: three classes, *DoS Hulk*, *PortScan*, and *DDoS* account for more than 90% of all the samples in the dataset. This is also reflected in the results presented in the next section, as the GANs do well on overrepresented classes but poorly on underrepresented classes, such as *Heartbleed*

or *Web Attack Sql Injection*. However, this is not surprising. This characteristic will be addressed in Section 4. Each flow consists of 85 features that the GANs are tasked to generate. Table 9 in Appendix A lists all features in the dataset. We additionally drop `Flow ID`, `Source IP`, `Destination IP`, `Protocol`, and `Timestamp`. This results in 80 features and during training we also drop the `Label` column. Hence, the GANs have to produce 79-dimensional vectors.

### 3.2 Architecture

As already mentioned, we apply four established GAN models: CGAN, WGAN, AC-GAN, and AC-WGAN. Figure 1 illustrates our architecture on a conceptual level. Overall, there are two central components: the generator and the discriminator. The generator has to produce increasingly more realistic attacks. The discriminator must decide whether the given attacks are real or fake. Hence, generator and discriminator are, in a way, competing against each other. As one component improves the other has to keep up, and vice versa. This is the fundamental idea of GANs. Both of them are represented by fully connected neural networks. We keep the generator and discriminator neural network architectures constant across all GAN models, only the optimizers may differ. In Appendix B we list the complete hyperparameters and architecture choices.

To generate attacks, the generator obtains two inputs: a noise vector $z$ and (in our case) an additional condition. As already mentioned, we represent the condition in two ways. This will be discussed shortly. The noise vector $z$ is a 128-dimensional random vector that is generated anew each time an attack is generated.

Similarly, the discriminator obtains two inputs: an attack and (in our case) an additional condition. During training the discriminator has to determine whether the attacks generated by the generator are real of fake. However, to learn to distinguish real from fake it also receives real attacks that are sampled from the dataset. Depending on the GAN model used, there might be small differences. For example, with AC-GAN the discriminator might not only have to predict whether the input is real or fake but also what class it is.

All input features are in different ranges. Therefore, we simply scale all feature columns using the min-max scaler from `scikit-learn` [26], before feeding the flows into the generator and discriminator of the GAN. Min-max scaling is not optimal for all columns. For example, the ports could be better represented as 16-bit vectors (as the port limit is 65535). However, we leave the problem of finding better feature representations for future work. At inference time the generated flows are re-transformed to their original scale using the saved scaler model.

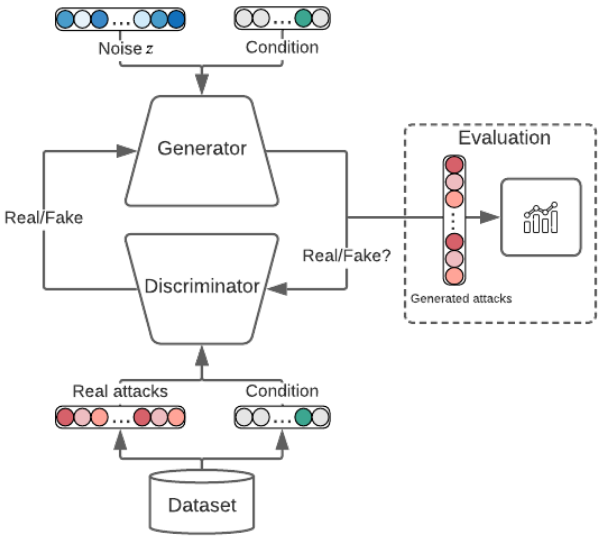



**Figure 1.** Architecture.

#### 3.2.1 Class conditions

In our preliminary experiments, the condition vector in Figure 1 was represented by a one-hot vector that encodes the attack type. As mentioned previously, there are 14 attack types hence, the condition vector has 14 dimensions. For CGAN and WGAN this condition vector can be fed into both the generator and the discriminator. However, the for the models that additionally employ an auxiliary classifier within the discriminator, only the generated/real attack is fed into the discriminator. Otherwise, it would be simple to predict the right class and the additional cross entropy loss on the attack types would be useless. This setup makes it possible to not just generate attacks, but attacks of a particular type.

When sampling the real attacks from the dataset for the discriminator, we also obtain the attack type. However, during runtime we also have to produce this class condition vector in addition to the noise vector for the generator. Therefore, the attack type of the noise vector is chosen at random. We also experimented with keeping a similar class distribution as shown in Table 1 but led to a more biased generator. Hence, each class is equally likely.

#### 3.2.2 Dynamic condition vectors

While the class condition approach allows to generate attacks of a particular attack type, it only enables coarse control. Therefore, our dynamic condition vector approach aims for a more fine-grained control over the generation process, by conditioning on a set of twelve features present in our dataset that the generated flow should or should not exhibit. The features we selected for this condition vector are: `Destination Port`, `Flow Duration`, `Total Backward`

Packets, Total Fwd Packets, Packet Length Mean, Flow Bytes/s, Flow IAT Min, Flow IAT Max, PSH Flag Count, SYN Flag Count, RST Flag Count, and ACK Flag Count. However, we do not just feed these features directly, but encode them in such a way that they are easily modifiable afterwards. Not all features are treated equally. Overall, they can be categorized into three groups:

1. **Three-level features:** Flow Duration, Total Backward Packets, Total Fwd Packets, Packet Length Mean, Flow Bytes/s, Flow IAT Min, and Flow IAT Max
2. **Flag-features:** PSH Flag Count, SYN Flag Count, RST Flag Count, and ACK Flag Count
3. **Destination Port**: mapped to its 16-bit representation

All three-level features are continuous. To convert them into three levels, each feature is binned into 3 buckets: low, mid and high. To achieve this for each feature in the training dataset the 33rd and the 66th quantiles are computed and hence the respective levels are $\{[0, 0.33], (0.33, 0.66], (0.66, 1]\}$, corresponding the low, mid and high.
The flag-features are binary and determine whether a flag is present or not. Therefore, no modification is required.
Furthermore, the destination port can, in principle, be set to any integer in range $[0 - 65535]$. Hence, it is mapped to its 16-bit representation.

Overall, this results in a 41-dimensional binary condition vector (3 * 7 + 4 + 16 = 41 dimensions) for each training sample. For the dataset the condition vector can simply be computed before training. However, as with the class condition the generator also expects the condition vector input during runtime. With class condition, the attack type could simply be picked at random, but for the dynamic condition vectors random construction is not desirable. Therefore, we collect a maximum of 10K real condition vectors for each attack type from the training dataset. During runtime the condition vectors are sampled uniformly given a random attack class and used in the generator.

This approach enables a more fine-grained control over the generation process. For example, we can specify that an attack with a short 'Flow Duration' and high 'Packet Length Mean' should be generated.

### 3.3 Evaluation

Finally, we describe our evaluation procedure. Evaluating the output of GANs is in general no trivial task [27]. Normally, when evaluating the outputs of GANs the most obvious criterion is visual inspection of the generated images/audio/video. However, in our case this is not an option. Nevertheless, the output has to be as realistic as possible. As already mentioned, we evaluate the validity of generated network flows on three dimensions: (1) using an external classifier, (2) statistical tests on the distribution of generated features, and (3) distance measurements between generated and real attacks. During training we evaluate after every epoch.

#### 3.3.1 External classifier

The first evaluation method is to use an external classifier. The intuition behind this approach is that the generated network flows should yield high detection ratios when fed into an IDS. Therefore, we train a Random Forest (RF) (using scikit-learn) on the original CIC-IDS2017 dataset. We selected this ML-model as it is trains relatively quickly and resulted in the highest classification scores in the original publication. Our classifier obtains similar scores as in the original publication [10]. The trained RF is then used during the training of the GAN models to classify the generated outputs. Overall, we report the accuracy-, precision-, recall-, F1-scores (marco and weighted) computed using the scikit-learn classification report.

#### 3.3.2 Statistical tests

Furthermore, we compute statistical tests between the generated feature distributions and real feature distributions. The intuition behind this evaluation method is that for the generated features to be realistic, they should be similar to the real features. Therefore, we apply the two-sample t-test (with significance level $\alpha = 5\%$) to each feature in a batch of generated attacks and real attacks. In the best case all 79 feature columns of real and generated attacks are not significantly different. If the GAN architecture learns to model the distributions, we should see an increase of the number of "real" feature columns throughout the training process.

#### 3.3.3 Distance measures

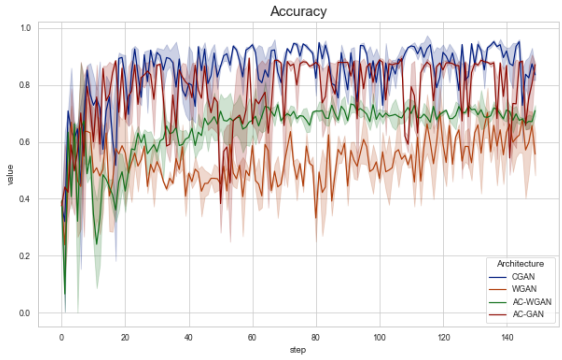
Similar to our second evaluation method we also compute the euclidean distances between real and generated feature columns. During training a decrease in the distances should be observed.

#### 3.3.4 Others

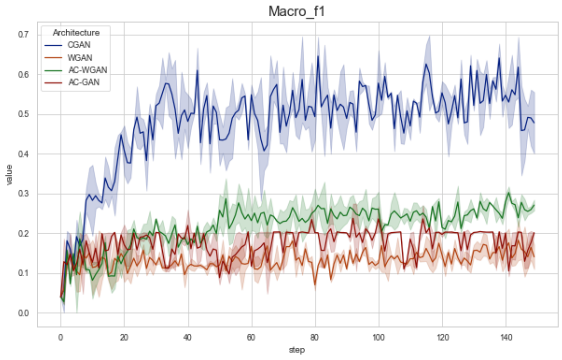
Another evaluation method that we relied on during training is visual inspection of real and generated distributions of a few selected feature columns. Over the course of training, the real and generated distributions should look increasingly similar. Another valuable evaluation method is to train an IDS with the artificial attacks generated by the GANs and to record the classification scores. However, we did not do this in this paper and leave it for future work.

## 4 Experiments & Results

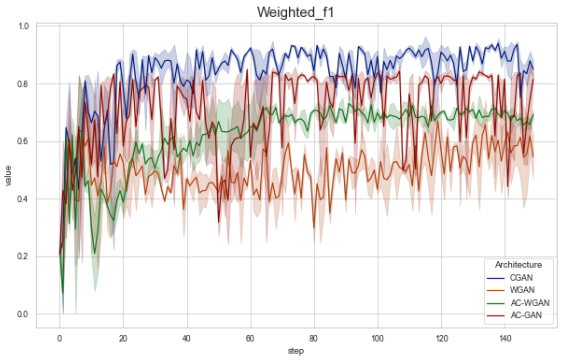
In this section we discuss our experiments and report our results. First, our experiments with the class conditions are shown, then we present the experiments with dynamic condition vectors.




**(a)** Accuracy.




**(b)** Macro F1.




**(c)** Weighted F1.

**Figure 2.** External classifier classification scores of generated flows.

| Metric | AC-GAN | AC-WGAN | CGAN | WGAN |
|---|---|---|---|---|
| Accuracy | **0.872±0.01** | 0.71±0.028 | 0.836±0.026 | 0.557±0.106 |
| Macro_f1 | 0.201±0.0 | 0.271±0.019 | **0.478±0.109** | 0.141±0.044 |
| Macro_precision | 0.19±0.0 | 0.301±0.001 | 0.466±0.109 | 0.141±0.048 |
| Macro_recall | 0.214±0.0 | 0.295±0.011 | 0.568±0.123 | 0.165±0.069 |
| Weighted_f1 | 0.815±0.011 | 0.692±0.022 | **0.85±0.038** | 0.546±0.107 |
| Weighted_precision | 0.767±0.01 | 0.711±0.007 | 0.884±0.057 | 0.562±0.121 |
| Weighted_recall | 0.872±0.01 | 0.71±0.028 | 0.836±0.026 | 0.557±0.106 |

**Table 2.** Mean classifier scores & standard deviations over 2 random seeds measured at final step.

### 4.1 Class conditions

**Classifier scores.** Figure 2 shows the classification scores (accuracy, macro F1, and weighted F1) and learning curves of the four GAN architectures over the course of 150 epochs. The whole evaluation pipeline is run after each epoch and each run is averaged over two random seeds. CGAN is represented by the blue curve, WGAN is in orange, AC-WGAN in green, and AC-GAN in red. Overall, the individual architectures work differently well. The accuracy and weighted F1-scores are relatively high for all architectures while the macro F1-scores are considerably lower. Hence, the GANs are not able to learn to generate attacks of all attack types well. We believe that this is due to the class imbalance in the dataset (Table 1).

Table 2 presents the exact scores and standard deviations for all architectures measured at the final training epoch. We see that AC-GAN achieves the highest accuracy score, while CGAN achieves the highest macro and weighted F1-scores. This shows that CGAN is best able to generate attacks of

| # of real features | AC-GAN | AC-WGAN | CGAN | WGAN |
|---|---|---|---|---|
| Bot | 38.0±5.657 | 27.0±7.071 | 41.5±2.121 | 45.0±4.243 |
| DDoS | 13.5±0.707 | 30.5±13.435 | 12.5±0.707 | 13.0±8.485 |
| DoS_GoldenEye | 11.5±2.121 | 23.0±4.243 | 28.0±2.828 | 15.5±0.707 |
| DoS_Hulk | 8.0±1.414 | 26.5±7.778 | 8.5±0.707 | 9.0±7.071 |
| DoS_Slowhttptest | 38.5±4.95 | 52.0±1.414 | 47.0±4.243 | 37.0±2.828 |
| DoS_slowloris | 18.0±2.828 | 32.0±0.0 | 18.0±4.243 | 38.5±12.021 |
| FTP-Patator | 8.0±1.414 | 16.0±4.243 | 12.0±4.243 | 14.5±6.364 |
| Heartbleed | 0.0±0.0 | 0.0±0.0 | 0.0±0.0 | 0.0±0.0 |
| Infiltration | 23.0±16.971 | 29.5±2.121 | 20.5±19.092 | 29.5±16.263 |
| PortScan | 31.0±1.414 | 40.5±0.707 | 37.0±4.243 | 20.5±9.192 |
| SSH-Patator | 2.0±0.0 | 15.0±4.243 | 5.5±2.121 | 7.5±6.364 |
| Web_Attack___Brute_Force | 20.5±9.192 | 30.5±7.778 | 40.0±11.314 | 47.5±0.707 |
| Web_Attack___Sql_Injection | 79.0±0.0 | 79.0±0.0 | 79.0±0.0 | 79.0±0.0 |
| Web_Attack___XSS | 5.5±2.121 | 4.5±3.536 | 11.5±9.192 | 13.0±11.314 |
| mean_n_real_features | 21.179±2.778 | **29.0±0.0** | 25.786±3.536 | 26.393±0.657 |

**Table 3.** Numbers of real features & standard deviations over 2 random seeds measured at final step.

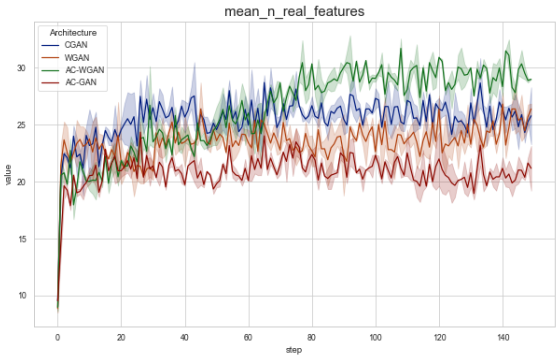



**Figure 3.** Number of real features according to statistical tests.

different attack types.

**Statistical tests.** Next, Figure 4.1 shows the mean number of real features across all 14 attack types averaged across 2 random seeds for the different GAN architectures. In addition, Table 3 shows the respective exact scores. Overall, we observe that AC-WGAN surprisingly achieves the highest mean number of real features across all attack types.

**Distance measures**. Finally, Table 4 shows the average distances across all attack types for all architectures. Here we observe that CGAN again obtains the best scores.

## 4.2 Dynamic condition vectors

Next, we present the results we obtained for our experiments with dynamic condition vectors. For this scenario we only report our results with CGAN, as it obtained the highest scores in the preliminary experiments.

| Architecture | Score |
|---|---|
| AC-GAN | 0.419±0.087 |
| AC-WGAN | 0.448±0.067 |
| CGAN | **0.377±0.062** |
| WGAN | 0.504±0.122 |

**Table 4.** Distance measures & standard deviations over 2 random seeds measured at final step.

### 4.2.1 General evaluation

Again, all results are averaged over 2 random seeds. First, Table 5 shows the classification metrics for CGAN with dynamic condition vectors. Furthermore, Table 11 and Table 12 in Appendix D show the number of real features and distances measures with dynamic condition vectors, respectively.

In general, the results with CGAN and dynamic condition vectors are similar to the results for CGAN with class condition. However, the mean number of real features is considerably higher with dynamic condition vectors than with the simple class condition.

### 4.2.2 Evaluation of generation control

Finally, we evaluate how well we can control the generation process with the dynamic approach. To achieve this, we vary one feature at a time and observe the effect on the features of the generated attacks.

| Metric | Scores |
|---|---|
| Accuracy | 0.836±0.017 |
| Macro_f1 | 0.507±0.046 |
| Macro_precision | 0.556±0.06 |
| Macro_recall | 0.52±0.062 |
| Weighted_f1 | 0.82±0.016 |
| Weighted_precision | 0.815±0.019 |
| Weighted_recall | 0.836±0.017 |

**Table 5.** Classifier scores & standard deviation with dynamic condition vectors.

| Level | Score |
|---|---|
| Low | 0.847 |
| Mid | 0.724 |
| High | 0.816 |
| Overall | 0.795 |

**Table 6.** Success scores by levels.

| Feature | Score |
|---|---|
| Flow Bytes/s | 0.571429 |
| Flow Duration | 0.857143 |
| Flow IAT Max | 0.904762 |
| Flow IAT Min | 0.523810 |
| Packet Length Mean | 0.904762 |
| Total Backward Packets | 0.857143 |
| Total Fwd Packets | 0.952381 |

**Table 7.** Success scores by features.

| Feature | F1 | Accuracy |
|---|---|---|
| ACK Flag Count | 0.762 | 0.775 |
| PSH Flag Count | 0.854 | 0.881 |
| RST Flag Count | 0.0 | 0.5 |
| SYN Flag Count | 0.587 | 0.746 |

**Table 8.** Success scores for flags.

**Three-level features**. First, we examine the three-level features. For each feature we generate 1000 attacks per class and average the feature values to obtain the estimate of how well the generation can be controlled. Figure E in Appendix E shows the average feature values (and 95% confidence intervals) of 'Flow Duration' for each attack type and each of the three specified levels (0=low, 1=mid, 2=high). From the barplots we get the impressions that the GAN is able to control the generation process well. When 'low' is specified, the generated attacks tend to be lowest, and similar behaviour can be observed for the 'mid' and 'high' levels.
However, to quantify this we compute a simple average success rate based on the mean feature values. We add +1 to the overall success score if the mean value of the generated features $\mu$ is in the right position, i.e.:

- if **low** and $\mu_{low} < \mu_{mid}$ and $\mu_{low} < \mu_{high}$, add +1
- if **mid** and $\mu_{low} < \mu_{mid} < \mu_{high}$, add +1
- if **high** and $\mu_{high} > \mu_{low}$ and $\mu_{high} > \mu_{mid}$, add +1

This score is computed for each feature and attack type. We have seven three-level features and 14 attack types. Hence, the max possible score is 7 * 14 * 3 = 294. To obtain the average success rate, we sum all scores and divide by 294. Tables 6 and 7 show the average success scores per level and feature. Overall, the success rate is relatively high (80%) and the GAN is able to control the generation of three-level features.

**Flag features**. Next, we evaluate the generation control of the binary flag features. Here, we simply compute the accuracy and F1 scores for the generated features, where the real labels are represented by the input condition (i.e., flag/no flag condition). Again 1000 attacks are generated for each option (4 features * 2 options * 1000). In Table 8 the scores are shown. Notably, the model achieves a low score for 'PSH Flag Count'. A reason for this is that this flag is never set in the training dataset, so it never learns to generate it. For other flags the scores are higher, and it is able to generate them well.

**Destination Port**. Finally, we evaluate how well the 'Destination Port' feature can be controlled. Different attacks are carried out on different ports. Therefore, for each attack it only makes sense to control for the ports that occur in the dataset. However, for most only a single Port is relevant (e.g., port 80 for DoS and web attacks, port 444 for 'Infiltration' and 'Heartbleed'). Only for 'PortScan' and 'Bot' a variety of ports numbers is used. However, it is likely the GAN is not able to generate the given port number exactly. Therefore, we show the distributions using boxplots for different port number inputs. For DDoS', 'DoS Hulk', 'DoS Slowhttptest', 'DoS GoldenEye', 'DoS slowloris', 'Web Attack Brute Force', 'Web Attack XSS', 'Web Attack Sql Injection' we use three input puts: 80, 20 and 120. For all of them port 80 is the only relevant port in the training set. Figure E in Appendix E shows the boxplots.
Indeed, the network is not able to set the port numbers exactly. However, we can observe that for port 80 the distribution is narrower around the value 80. Similarly, for port 20 it is overall lower and for port 120 the distribution is higher. But they are both biased towards port 80.
For 'Infiltration' and 'Heartbleed' (port 444), 'FTP-Patator' (port 21) and 'SSH-Patator' (port 22) we generate the same

plots, but there the network is not able to control the generation process presumably because there are only a handful of training samples in the dataset.
Finally, we do the same for 'PortScan' and 'Bot'. These attacks do not operate on a single port, so there is more variability in the dataset. We pick 5 of the most often occurring ports for this attack type and there we do see a clear distinction among the generated port distributions, as shown by Figure E in Appendix E. Nevertheless, the utility is not clear.
These results show that overall, it does not make much sense to control this feature or even to generate it. For the attack to be realistic (or for it to work in practice), it is necessary that the port number is set correctly. As for most attacks there is only a handful of relevant ports, it might be better to set them manually.

## 5 Discussion

Our experiments suggest that GANs can be an effective tool to generate network attacks with pre-specified properties. The simple CGAN architecture showed to give the best results for class conditions, in particular in terms of the macro F1 score. We consider the macro F1-score as the primary metric to measure the success of the generation process, as it not only indicates that attacks of overrepresented attack types can be generated but also attacks of underrepresented classes. However, the class imbalance in the CIC-IDS2017 dataset is a major obstacle for the selected GAN architectures, as they have difficulties with producing underrepresented attacks. Nevertheless, GANs are able to generate network attacks that yield high detection rates. Furthermore, our experiments with dynamic condition vectors demonstrate that a fine-grained control over the generation process can be achieved. In particular, controlling features with multiple levels has shown to be effective. In contrast, the utility of generating the ports is not clear.

## 6 Conclusion

In this paper, we tackle the problem of conditional network attack generation using GANs. We apply four established GAN architectures, CGAN, WGAN, AC-GAN, and AC-WGAN to the CIC-IDS2017 dataset and explore two kinds of conditions for attack generation. The first condition type only uses the attack types. The second approach aims for a more fine-grained control over the generation process, by conditioning on a set of twelve features. We evaluate the generated outputs in three ways and show that it is possible to control the attack generation process.

## References


[1] A. Khraisat, I. Gondal, P. Vamplew, and J. Kamruzzaman, "Survey of intrusion detection systems: techniques, datasets and challenges," *Cybersecurity*, vol. 2, no. 1, pp. 1–22, 2019.

[2] I. J. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial networks," *arXiv preprint arXiv:1406.2661*, 2014.

[3] M. Mirza and S. Osindero, "Conditional generative adversarial nets," *arXiv preprint arXiv:1411.1784*, 2014.

[4] M. Arjovsky, S. Chintala, and L. Bottou, "Wasserstein generative adversarial networks," in *International conference on machine learning*, pp. 214–223, PMLR, 2017.

[5] A. Odena, C. Olah, and J. Shlens, "Conditional image synthesis with auxiliary classifier gans," in *International conference on machine learning*, pp. 2642–2651, PMLR, 2017.

[6] I. Gulrajani, F. Ahmed, M. Arjovsky, V. Dumoulin, and A. Courville, "Improved training of wasserstein gans," in *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 5769–5779, 2017.

[7] M. Ring, D. Schlör, D. Landes, and A. Hotho, "Flow-based network traffic generation using generative adversarial networks," *Computers & Security*, vol. 82, pp. 156–172, 2019.

[8] M. Rigaki and S. Garcia, "Bringing a gan to a knife-fight: Adapting malware communication to avoid detection," in *2018 IEEE Security and Privacy Workshops (SPW)*, pp. 70–75, IEEE, 2018.

[9] J. Charlier, A. Singh, G. Ormazabal, R. State, and H. Schulzrinne, "Syngan: Towards generating synthetic network attacks using gans," *arXiv preprint arXiv:1908.09899*, 2019.

[10] I. Sharafaldin, A. H. Lashkari, and A. A. Ghorbani, "Toward generating a new intrusion detection dataset and intrusion traffic characterization.," in *ICISSp*, pp. 108–116, 2018.

[11] 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. 41525–41550, 2019.

[12] M. Ring, S. Wunderlich, D. Scheuring, D. Landes, and A. Hotho, "A survey of network-based intrusion detection data sets," *Computers & Security*, vol. 86, pp. 147–167, 2019.

[13] F. Meghdouri, M. Bachl, and T. Zseby, "Eagernet: Early predictions of neural networks for computationally efficient intrusion detection," in *2020 4th Cyber Security in Networking Conference (CSNet)*, pp. 1–7, IEEE, 2020.

[14] A. L. Buczak and E. Guven, "A survey of data mining and machine learning methods for cyber security intrusion detection," *IEEE Communications surveys & tutorials*, vol. 18, no. 2, pp. 1153–1176, 2015.

[15] M. Roesch *et al.*, "Snort: Lightweight intrusion detection for networks.," in *Lisa*, vol. 99, pp. 229–238, 1999.

[16] A. Radford, L. Metz, and S. Chintala, "Unsupervised representation learning with deep convolutional generative adversarial networks," *arXiv preprint arXiv:1511.06434*, 2015.

[17] T. Karras, S. Laine, M. Aittala, J. Hellsten, J. Lehtinen, and T. Aila, "Analyzing and improving the image quality of stylegan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 8110–8119, 2020.

[18] S. Tulyakov, M.-Y. Liu, X. Yang, and J. Kautz, "Mocogan: Decomposing motion and content for video generation," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, pp. 1526–1535, 2018.

[19] C. Donahue, J. McAuley, and M. Puckette, "Adversarial audio synthesis," *arXiv preprint arXiv:1802.04208*, 2018.

[20] J. Engel, K. K. Agrawal, S. Chen, I. Gulrajani, C. Donahue, and A. Roberts, "Gansynth: Adversarial neural audio synthesis," *arXiv preprint arXiv:1902.08710*, 2019.

[21] W. Nie, N. Narodytska, and A. Patel, "Relgan: Relational generative adversarial networks for text generation," in *International conference on learning representations*, 2018.

[22] L. Xu, M. Skoularidou, A. Cuesta-Infante, and K. Veeramachaneni, "Modeling tabular data using conditional gan," *arXiv preprint arXiv:1907.00503*, 2019.

[23] S. W. Kim, Y. Zhou, J. Philion, A. Torralba, and S. Fidler, "Learning to simulate dynamic environments with gamegan," in *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1231–1240, 2020.

[24] Z. Lin, Y. Shi, and Z. Xue, "Idsgan: Generative adversarial networks for attack generation against intrusion detection," *arXiv preprint arXiv:1809.02077*, 2018.

[25] A. Cheng, "Pac-gan: Packet generation of network traffic using generative adversarial networks," in *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pp. 0728–0734, IEEE, 2019.

[26] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, *et al.*, "Scikit-learn: Machine learning in python," *the Journal of machine Learning research*, vol. 12, pp. 2825–2830, 2011.

[27] A. Borji, "Pros and cons of gan evaluation measures," *Computer Vision and Image Understanding*, vol. 179, pp. 41–65, 2019.

## A CIC-IDS2017 Dataset

Table 9 shows all feature column names present in the dataset.

## B Neural network architectures

Table 10 lists all hyperparameters and neural network architectures.

## C Class conditions

Figure 4 also shows the weighted and macro precision and recall scores in addition to the accuracy and F1 scores presented in the main text.

## D Dynamic condition vectors

Table 11 shows the number of 'real' features produces by CGAN with dynamic condition vectors according to statistical tests. Furthermore, Table 12 shows the distance measures for flows generated with CGAN and dynamic condition vectors. Additionally, Figure 5 shows the classifier scores over all 150 epochs for CGAN with dynamic condition vectors.

## E Generation control

Figure E shows the mean feature values (and 95% confidence intervals) of 'Flow Duration' and other features of the generated flows produces with CGAN and dynamic condition vectors for three levels. The subsequent figures show the same plots for 'Flow Bytes/s', 'Flow IAT Max', 'Flow IAT Min', 'Packet Length Mean', 'Total Backward Packets', 'Total Fwd Packets'.

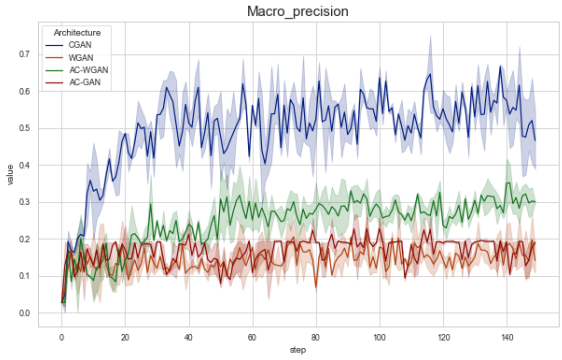
Furthermore, Figures E and E show the distributions of the produces feature values for 'Destination Port' with CGAN and dynamic condition vectors.

| Features | | | | |
|---|---|---|---|---|
| Flow ID | Source IP | Source Port | Destination IP | Destination Port |
| Protocol | Timestamp | Flow Duration | Total Fwd Packets | Total Backward Packets |
| Total Length of Fwd Packets | Total Length of Bwd Packets | Fwd Packet Length Max | Fwd Packet Length Min | Fwd Packet Length Mean |
| Fwd Packet Length Std | Bwd Packet Length Max | Bwd Packet Length Min | Bwd Packet Length Mean | Bwd Packet Length Std |
| Flow Bytes/s | Flow Packets/s | Flow IAT Mean | Flow IAT Std | Flow IAT Max |
| Flow IAT Min | Fwd IAT Total | Fwd IAT Mean | Fwd IAT Std | Fwd IAT Max |
| Fwd IAT Min | Bwd IAT Total | Bwd IAT Mean | Bwd IAT Std | Bwd IAT Max |
| Bwd IAT Min | Fwd PSH Flags | Bwd PSH Flags | Fwd URG Flags | Bwd URG Flags |
| Fwd Header Length | Bwd Header Length | Fwd Packets/s | Bwd Packets/s | Min Packet Length |
| Max Packet Length | Packet Length Mean | Packet Length Std | Packet Length Variance | FIN Flag Count |
| SYN Flag Count | RST Flag Count | PSH Flag Count | ACK Flag Count | URG Flag Count |
| CWE Flag Count | ECE Flag Count | Down/Up Ratio | Average Packet Size | Avg Fwd Segment Size |
| Avg Bwd Segment Size | Fwd Header Length.1 | Fwd Avg Bytes/Bulk | Fwd Avg Packets/Bulk | Fwd Avg Bulk Rate |
| Bwd Avg Bytes/Bulk | Bwd Avg Packets/Bulk | Bwd Avg Bulk Rate | Subflow Fwd Packets | Subflow Fwd Bytes |
| Subflow Bwd Packets | Subflow Bwd Bytes | Init_Win_bytes_forward | Init_Win_bytes_backward | act_data_pkt_fwd |
| min_seg_size_forward | Active Mean | Active Std | Active Max | Active Min |
| Idle Mean | Idle Std | Idle Max | Idle Min | Label |

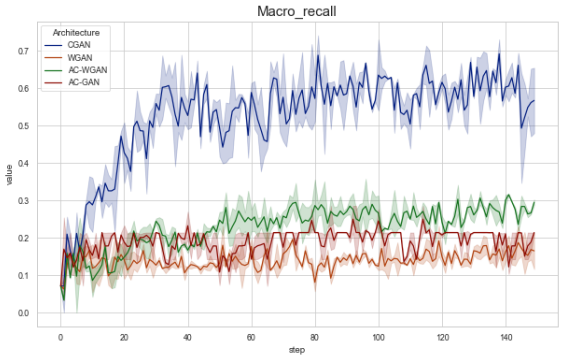
**Table 9.** List of all features

| Parameter | Model | Value |
|---|---|---|
| Batch size | All | 256 |
| G latent dim | All | 128 |
| Epochs | All | 150 |
| Num features | All | 79 |
| Num attack types | All | 14 |
| Discriminator layers | All | 3 × [Linear(256), LeakyReLU(0.2)] |
| Generator layers | All | 3 × [Linear(256), LeakyReLU(0.2)] + BatchNorm1d + Sigmoid |
| Condition size | CGAN w/ condition vectors | 41 |
| Condition latent dim | CGAN w/ condition vectors | 25 |
| Learning rate | CGAN (w/ condition vectors), ACGAN | 0.0002 |
| Learning rate | WGAN | 5e-05 |
| Optimizer | CGAN (w/ condition vectors), ACGAN | Adam |
| Optimizer | WGAN, AC-WGAN | RMSProp |
| Clip val | WGAN, AC-WGAN | 0.1 |

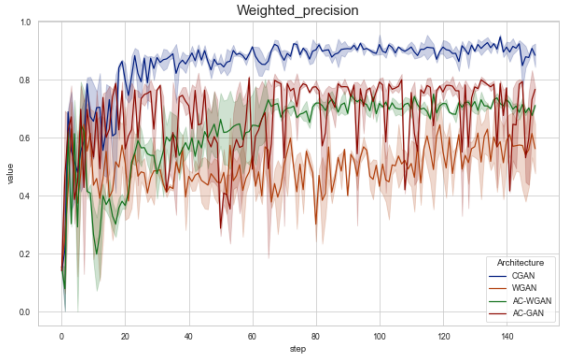
**Table 10.** Hyperparameters.




**(a)** Macro precision.




**(b)** Macro recall.




**(c)** Weighted precision.


**(d)** Weighted recall.

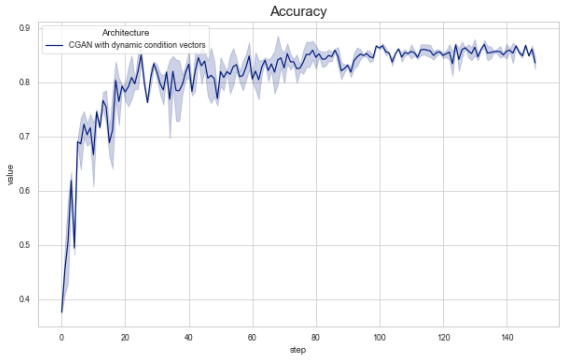
**Figure 4.** External classifier classification scores of generated flows.

| **Attack Type** | **# of real features** |
|---|---|
| Bot | 47.0±2.828 |
| DDoS | 34.5±4.95 |
| DoS_GoldenEye | 53.5±0.707 |
| DoS_Hulk | 30.0±1.414 |
| DoS_Slowhttptest | 50.5±7.778 |
| DoS_slowloris | 36.5±13.435 |
| FTP-Patator | 40.5±3.536 |
| Heartbleed | 0.0±0.0 |
| Infiltration | 55.0±2.828 |
| PortScan | 39.5±2.121 |
| SSH-Patator | 41.0±5.657 |
| Web_Attack___Brute_Force | 42.0±7.071 |
| Web_Attack___Sql_Injection | 79.0±0.0 |
| Web_Attack___XSS | 39.0±31.113 |
| mean_n_real_features | 42.0±3.131 |

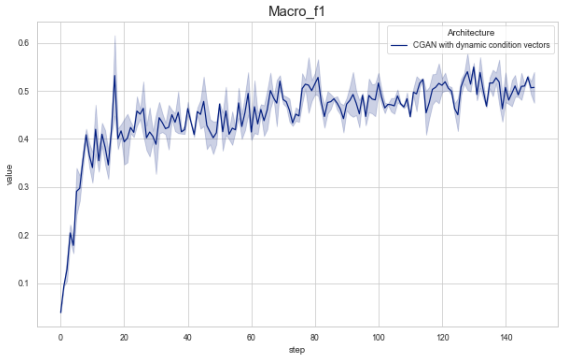
**Table 11.** Number of real features according to statistical tests with dynamic condition vectors.

| Attack Type | Distance |
|---|---|
| Bot | 0.115±0.037 |
| DDoS | 1.166±0.065 |
| DoS_GoldenEye | 0.285±0.027 |
| DoS_Hulk | 1.941±0.055 |
| DoS_Slowhttptest | 0.431±0.116 |
| DoS_slowloris | 0.436±0.05 |
| FTP-Patator | 0.156±0.002 |
| Heartbleed | 0.703±0.296 |
| Infiltration | 0.44±0.128 |
| PortScan | 0.193±0.007 |
| SSH-Patator | 0.106±0.021 |
| Web_Attack___Brute_Force | 0.077±0.06 |
| Web_Attack___Sql_Injection | 0.105±0.011 |
| Web_Attack___XSS | 0.062±0.062 |

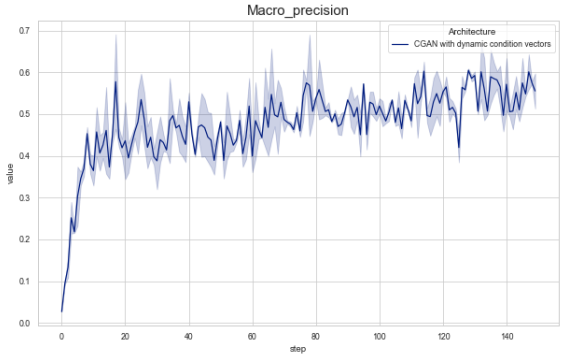
**Table 12.** Distance measures with dynamic condition vectors.



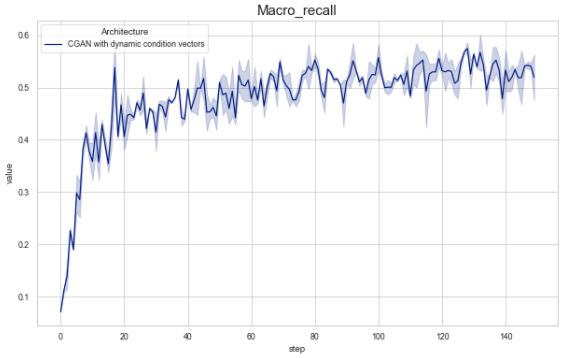

**(a)** Accuracy.



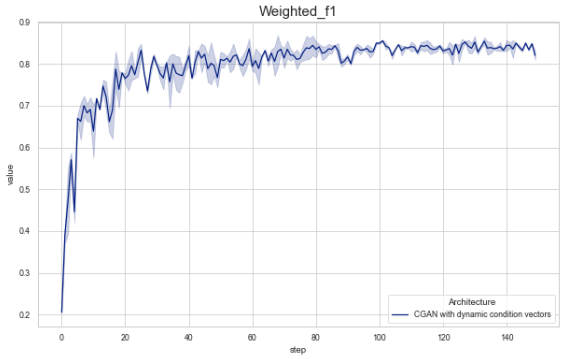

**(b)** Macro F1.



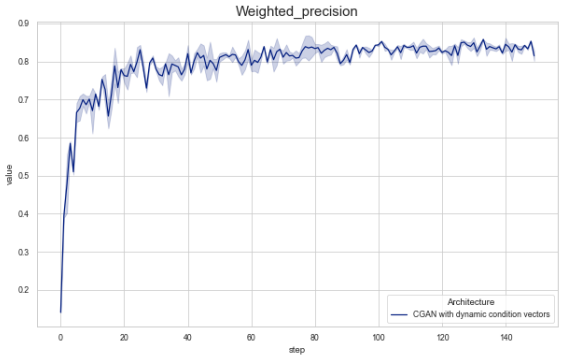

**(c)** Macro precision.




**(d)** Macro recall.




**(e)** Weighted F1.




**(f)** Weighted precision.


**(g)** Weighted recall.

**Figure 5.** External classifier classification scores of generated flows by CGAN with dynamic condition vectors.

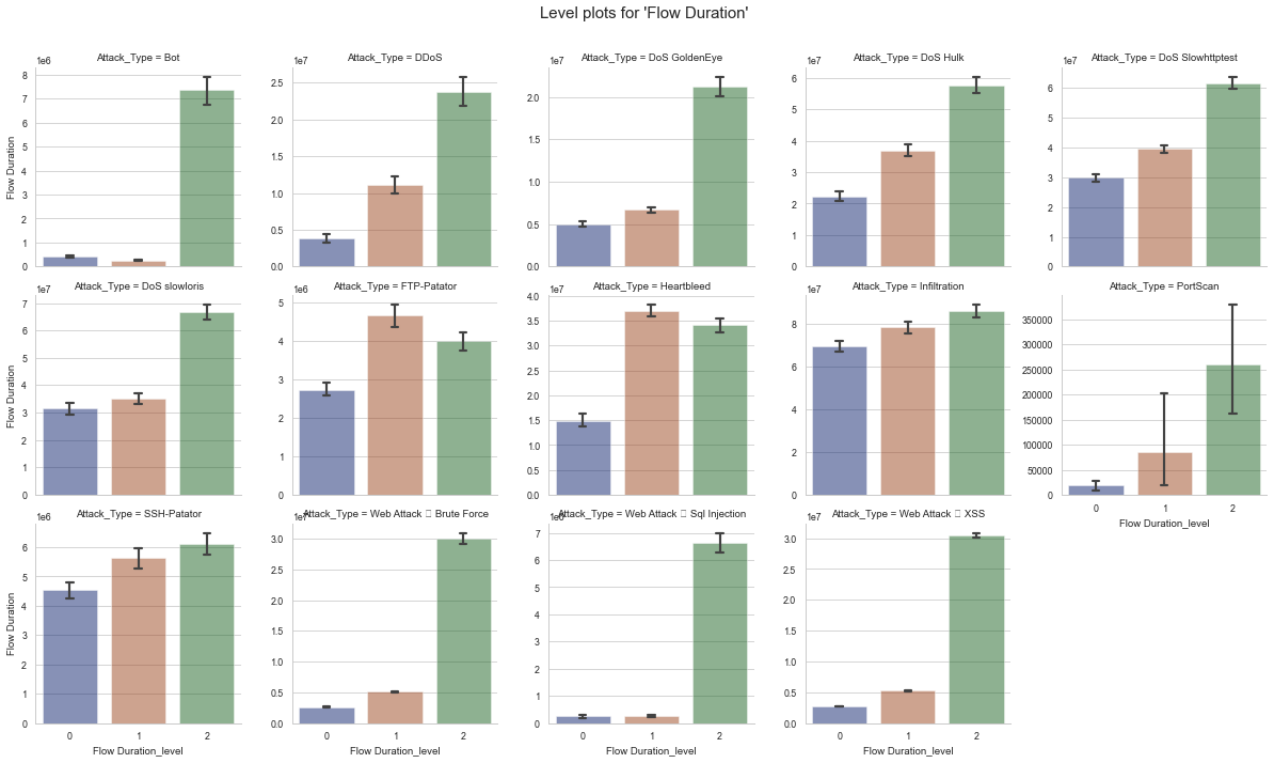



**Figure 6.** Generated attacks with low/mid/high level conditions for Flow Duration.

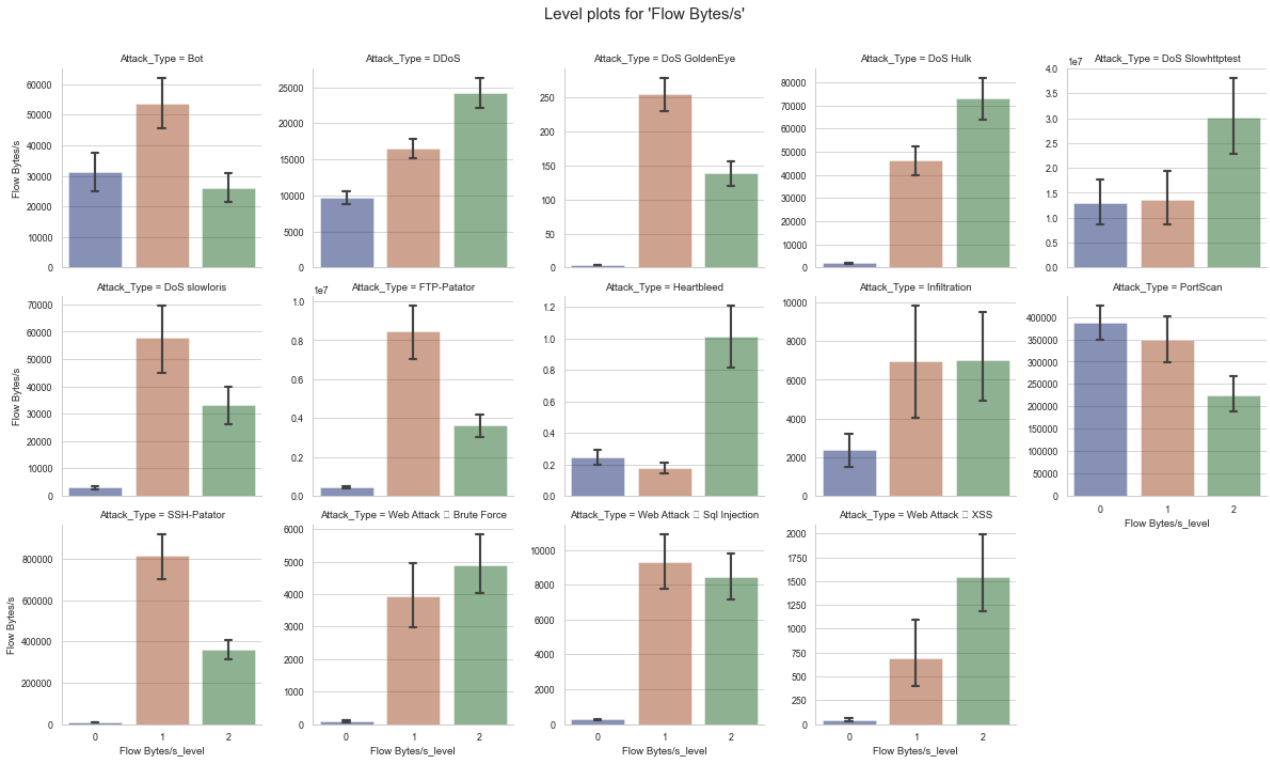



**Figure 7.** Generated attacks with low/mid/high level conditions for Flow Bytes/s.

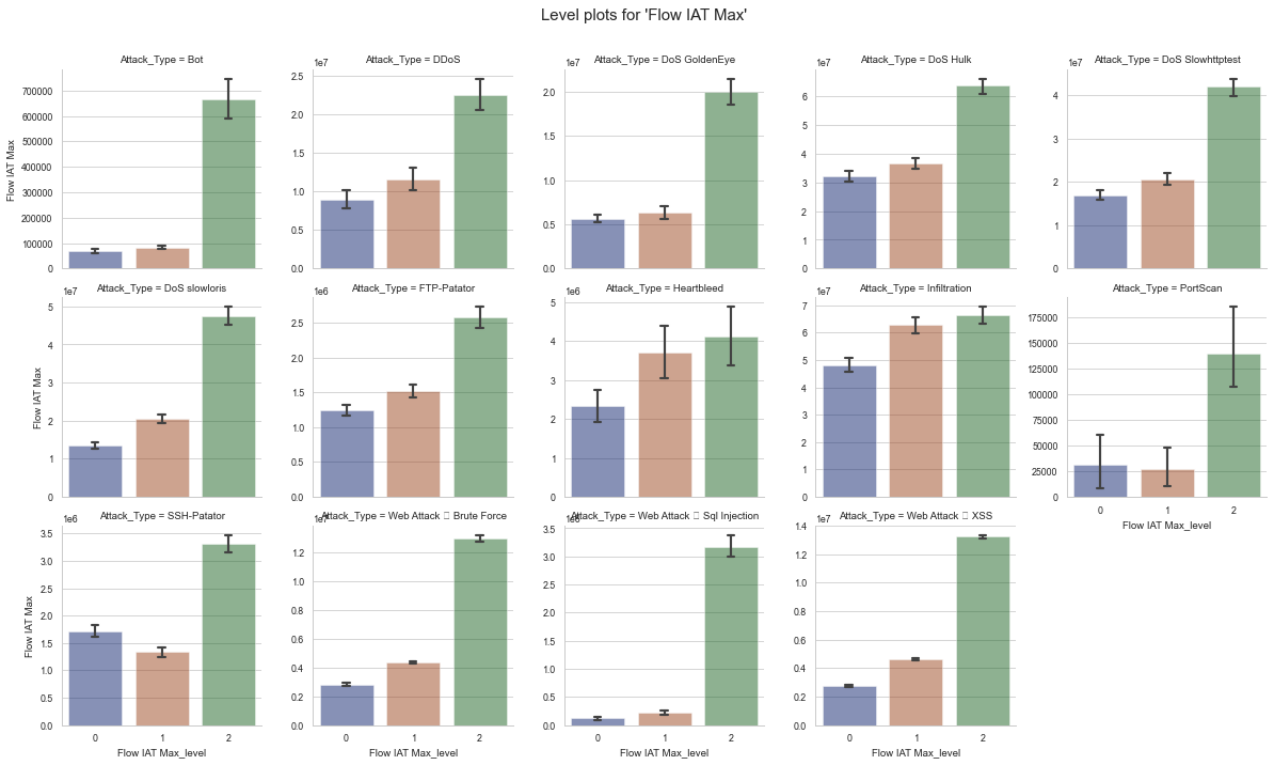



**Figure 8.** Generated attacks with low/mid/high level conditions for Flow IAT Max.

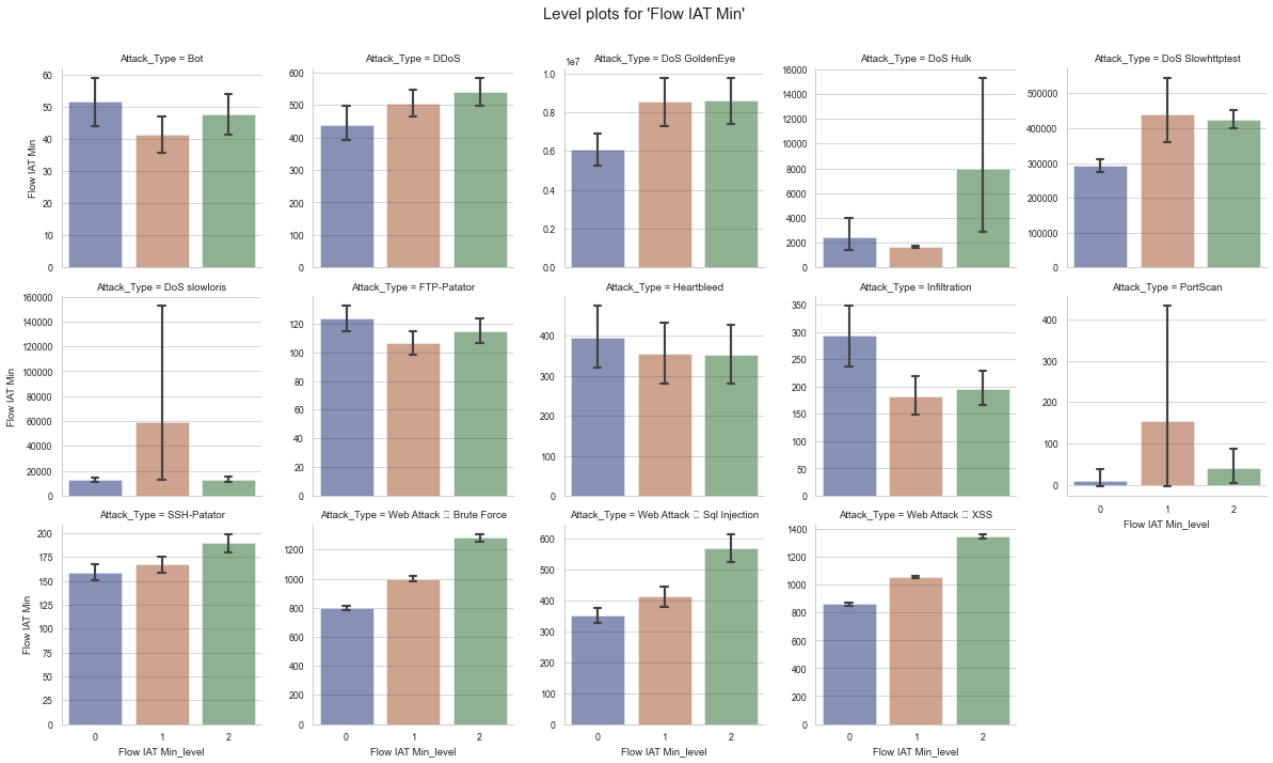



**Figure 9.** Generated attacks with low/mid/high level conditions for Flow IAT Min.

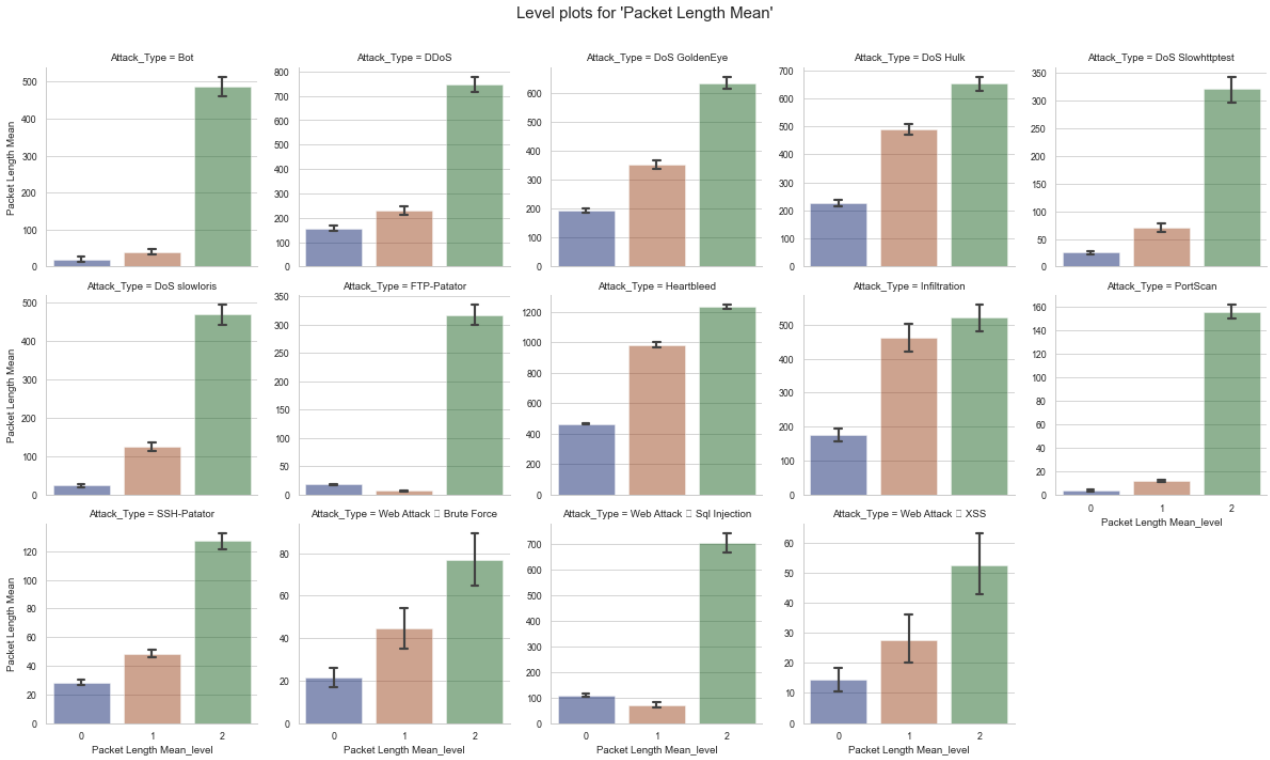



**Figure 10.** Generated attacks with low/mid/high level conditions for Packet Length Mean.

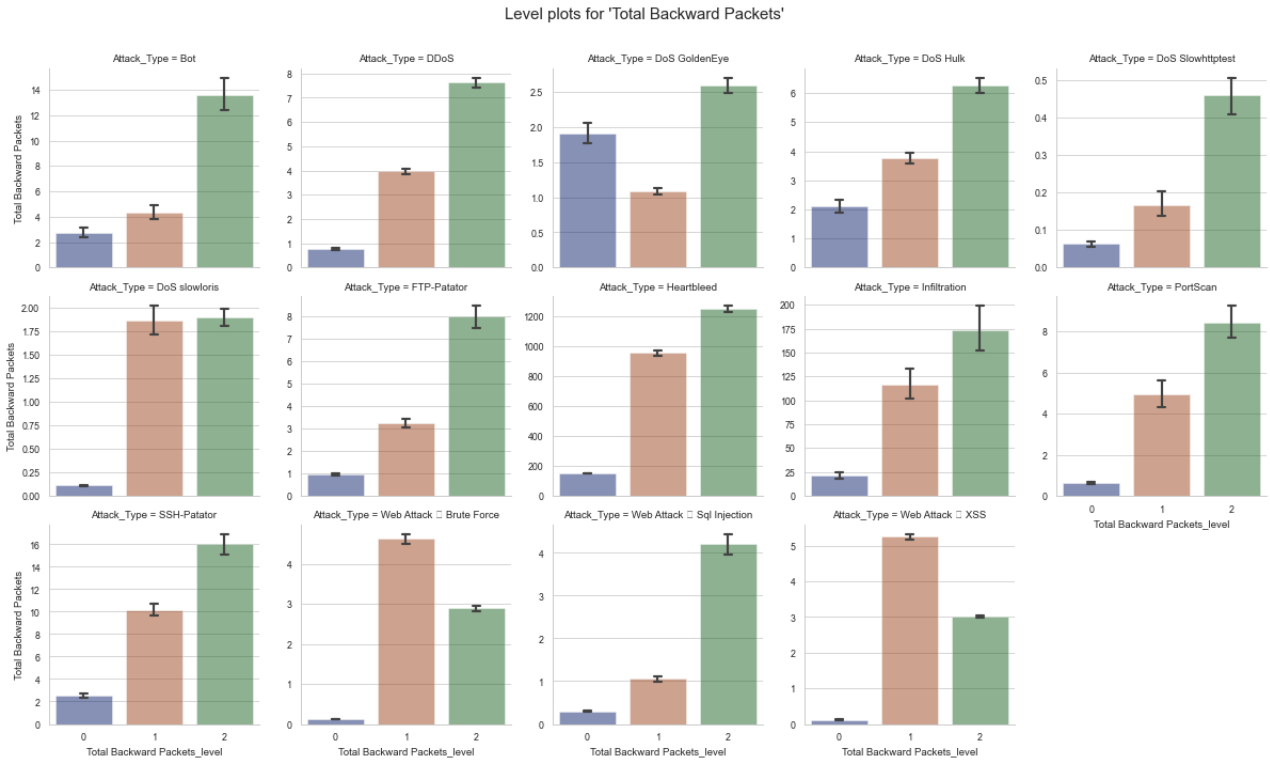



**Figure 11.** Generated attacks with low/mid/high level conditions for Total Backward Packets.

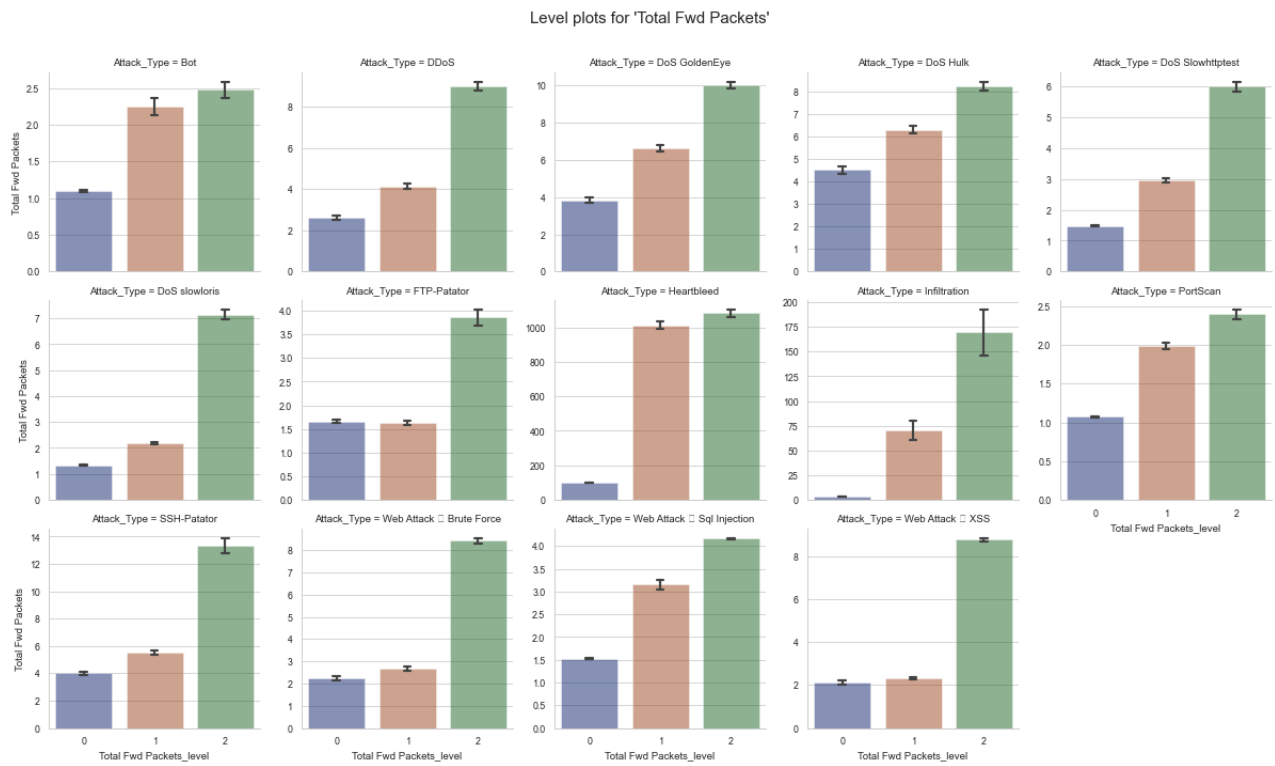



**Figure 12.** Generated attacks with low/mid/high level conditions for Total Fws Packets.

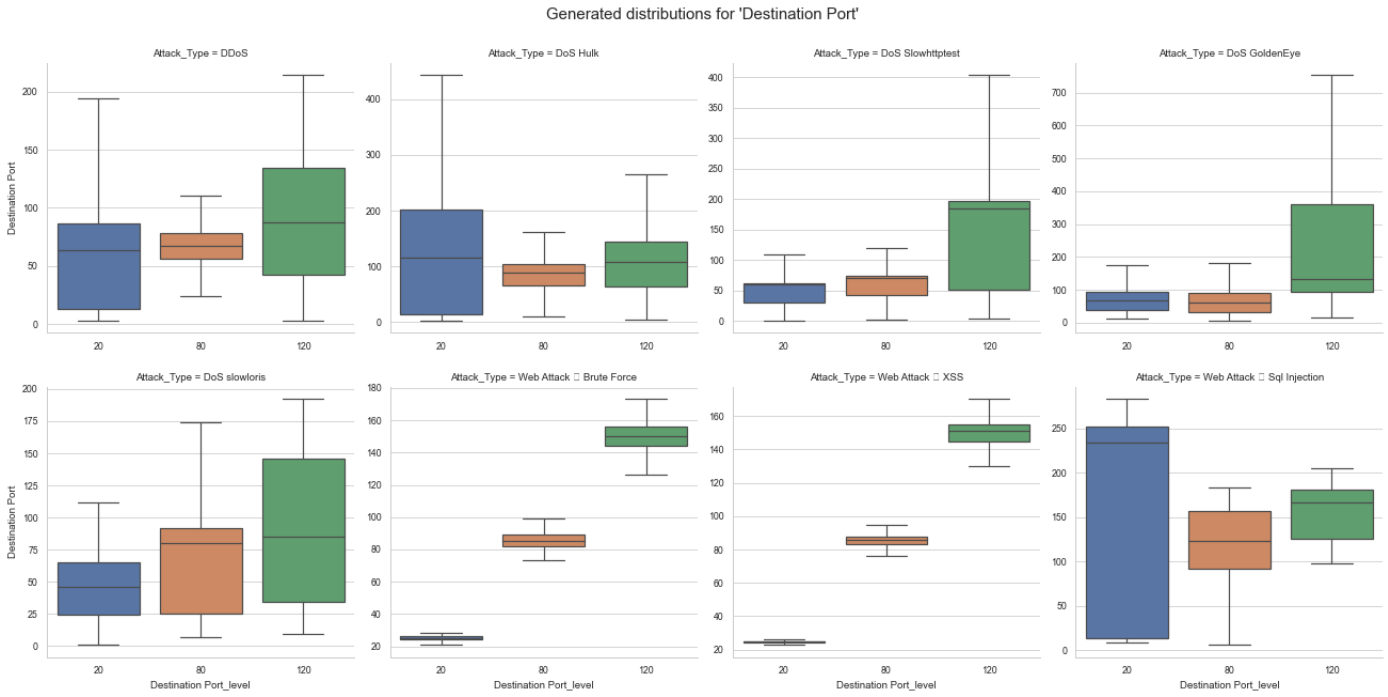
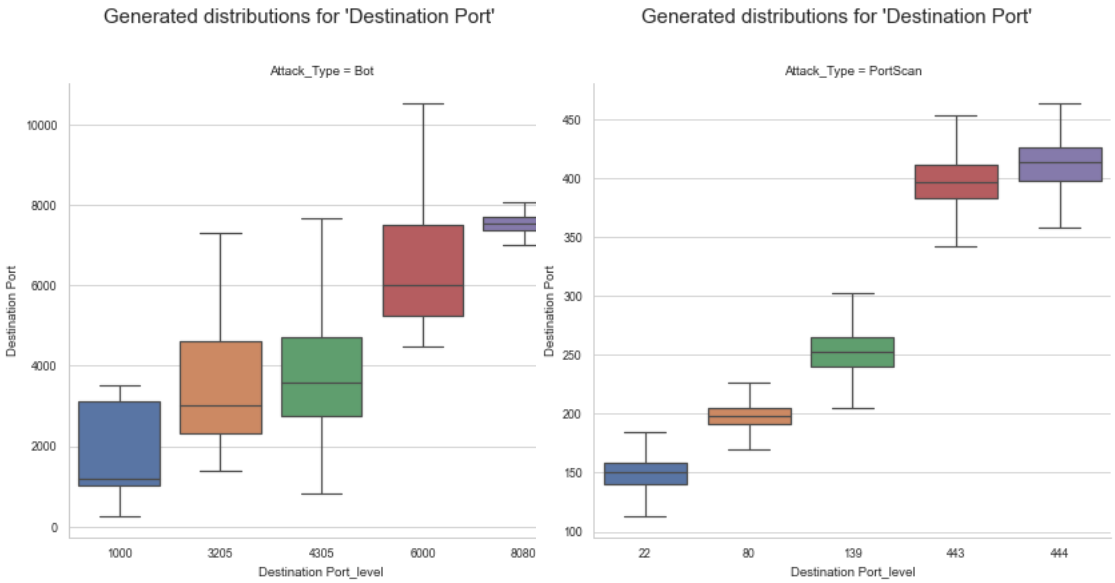



**Figure 13.** Distributions for 'Destination Port'. Default port 80.




**(a)** Bot **(b)** PortScan.

**Figure 14.** Distributions for 'Destination Port'