Exploring Generative Adversarial Networks for Augmenting Network Intrusion Detection Tasks

MIHAI GABRIEL CONSTANTIN, DAN-CRISTIAN STANCIU, LIVIU-DANIEL ŞTEFAN, and MIHAI DOGARIU, University Politehnica of Bucharest, Romania
DAN MIHĂILESCU and GEORGE CIOBANU, Keysight Technologies, Romania
MATT BERGERON
WINSTON LIU and KONSTANTIN BELOV, Keysight Technologies, USA
OCTAVIAN RADU and BOGDAN IONESCU, University Politehnica of Bucharest, Romania

The advent of generative networks and their adoption in numerous domains and communities have led to a wave of innovation and breakthroughs in artificial intelligence and machine learning. Generative Adversarial Networks (GANs) have expanded the scope of what is possible with machine learning, allowing for new applications in areas such as computer vision, natural language processing, and creative AI. GANs, in particular, have been used for a wide range of tasks, including image and video generation, data augmentation, style transfer, and anomaly detection. They have also been used for medical imaging and drug discovery, where they can generate synthetic data to augment small datasets, reduce the need for expensive experiments, and lower the number of real patients that must be included in medical trials. Given these developments, we propose using the power of generative adversarial networks to create and augment flow-based network traffic datasets. We evaluate a series of GAN architectures, including Wasserstein, conditional, energy-based, gradient penalty, and LSTM GANs. We evaluate their performance on a set of flow-based network traffic data collected from 16 subjects who used their computers for home, work, and study purposes. The performance of these GAN architectures is described according to metrics that involve networking principles, data distribution among a collection of flows, and temporal data distribution. Given the tendency of network intrusion detection datasets to have a very imbalanced data distribution, i.e., a large number of samples in the "normal traffic" category and a comparatively low number of samples assigned to the "intrusion" categories, we test our GANs by augmenting the intrusion data and checking whether this helps intrusion detection neural networks in their task. We publish the resulting UPBFlow dataset and code on GitHub[1].

CCS Concepts: • **Computing methodologies** → **Machine learning**; • **Networks** → *Network simulations.*

Additional Key Words and Phrases: generative networks, network traffic, network flow

ACM Reference Format:
Mihai Gabriel Constantin, Dan-Cristian Stanciu, Liviu-Daniel Ştefan, Mihai Dogariu, Dan Mihăilescu, George Ciobanu, Matt Bergeron, Winston Liu, Konstantin Belov, Octavian Radu, and Bogdan Ionescu. 2023. Exploring Generative Adversarial Networks for Augmenting Network Intrusion Detection Tasks. *ACM Trans. Multimedia Comput. Commun. Appl.* 1, 1, Article 1 (March 2023), 19 pages. https://doi.org/XXXXXXX.XXXXXXX

[1]https://github.com/cmihaigabriel/UPBFlow_dataset_code

Authors' addresses: Mihai Gabriel Constantin, mihai.constantin84@upb.ro; Dan-Cristian Stanciu; Liviu-Daniel Ştefan; Mihai Dogariu, University Politehnica of Bucharest, Romania; Dan Mihăilescu; George Ciobanu, Keysight Technologies, Romania; Matt Bergeron; Winston Liu; Konstantin Belov, Keysight Technologies, USA; Octavian Radu; Bogdan Ionescu, University Politehnica of Bucharest, Romania.

1551-6857/2023/3-ART1 $15.00
https://doi.org/XXXXXXX.XXXXXXX

1 INTRODUCTION

Network intrusion detection is the process of monitoring computer networks to identify and respond to unauthorized access or malicious activity. It involves analyzing network traffic and system logs in real time to detect potential security breaches or attacks and taking appropriate action to prevent or mitigate the impact of such incidents. Intrusion detection systems (IDS) are the primary tools used for detecting malicious activities in computer networks and are represented by software components that are deployed on the network to monitor traffic and identify potential threats. They can be configured to detect known attack patterns or anomalies in network traffic that may indicate an intrusion. There are two main types of IDS: signature-based and anomaly-based. While signature-based IDS use a database of known attack patterns to identify and respond to specific threats, anomaly-based IDS, use machine learning and statistical analysis to detect unusual patterns or behavior that may indicate an intrusion. Once an intrusion is detected, IDS can trigger alerts, block traffic, or initiate other response actions to prevent or mitigate the impact of the attack. In some cases, intrusion detection may also involve forensic analysis of network logs and other data to investigate the cause and scope of the intrusion and identify potential vulnerabilities in the network.

Given the critical nature of IDS as network security components, it is paramount that the performance of these models be improved as much as possible. However, many datasets that contain network attack and intrusion data currently suffer from data imbalance problems [7, 27, 29], specifically, the class containing normal traffic has a large number of elements, while classes containing attacks have significantly fewer samples. This is perhaps a normal behavior, as one would expect that real-world data captured in one or more computer networks generally contains significantly more regular activity. However, the low number of samples assigned to anomalies in general and to certain types of less popular attacks, in particular, may be detrimental to training deep neural networks (DNNs) in detecting attacks and may adversely affect the adoption of DNNs in IDS scenarios.

This issue is, however, not limited to network data analysis [26]. Many domains, including medical, fraud and anomaly detection, and fault diagnostics, suffer from the same issue. Whether this may be an effect of data privacy in the case of medical data or of the inherent nature of the data, as is the case for anomaly and fault detection tasks, where the target classes are rare in a real-world scenario, this problem still has to be addressed. One of the main avenues of research in solving the imbalance problem is the use of Generative Adversarial Networks (GANs) to create additional artificial samples that augment the classes that are not adequately represented [9, 28, 32]. In the most general sense [10], GANs are a type of deep learning architecture used to generate new artificial data that follows a similar distribution to the original data, composed of a generator and a discriminator network. The generator network takes a random noise vector as input and produces samples intended to be similar to the actual data. The discriminator network takes as input a sample, and its goal is to distinguish between the generated artificial samples and the actual real-world data. The generator is trained to produce samples that can fool the discriminator, while the discriminator is trained to accurately differentiate between real and generated samples. Despite their effectiveness, GANs can be challenging to train and require careful tuning of hyperparameters and network architecture to achieve good results. However, with proper training and tuning, GANs can produce high-quality samples that are difficult to distinguish from real-world data.

The remainder of the article is structured as follows. Section 2 presents the current state of the art in this domain and positions our work with respect to current advances. Section 3 describes the set of GAN architectures proposed in this paper and analyzes the particularities of their implementation, while Section 4 describes the experimental setup used for training and testing the systems, as well as the UPBFlow dataset and post-processing methods applied to the gathered data. The results of these experiments and their analysis are presented in Section 5. Finally, Section 6 concludes the paper.

2 RELATED WORK

While the application of GANs to network data received relatively less attention in the literature compared to other domains, there still are some papers that augment their datasets using generative approaches. Wasserstein GANs (WGAN) have been applied to this problem in several papers [1, 16, 25, 35], and this type of GAN approach seems to be the most frequent in this domain. A gradient penalty-enhanced version of WGAN has been studied in [16], where the authors use generative networks for solving the data imbalance problem in malicious traffic detection tasks while also testing the results on a convolutional classifier. [25] analyzes several methods of encoding NetFlow data in order to enhance the performance of gradient penalty WGAN approaches. The authors propose numerical, binary and embedded transformations of the NetFlow data, therefore creating three methods of adapting flows to the input of a GAN network. [35] performs an analysis on how well WGAN-improved data behaves compared to additional real-world data when augmenting a classifier for network issues, with encouraging results with regard to the use of GANs in networking-related tasks.

Convolutional GANs represent another interesting approach, with several papers [5, 17] exploring the idea of using special input transformation methods that can represent sequences of flow data as greyscale images that can then be fed into a 2D-Convolutional GAN approach. For example [5] proposes an input transformation method where individual bytes that compose network packets are decomposed into digits and then into a hexadecimal base-16 representation. Values across the packets are then transposed onto a 2D matrix that can be interpreted as a greyscale image by any type of deep neural network. Conditional GANs are also explored in the literature [31], where the authors propose using labels of traffic or application types as the conditional input vector. A more extensive study is available in [1], where the authors explore vanilla, least squares, Energy-Based, Wasserstein, and gradient penalty Wasserstein GANs for creating artificial network flow data.

In this context, this work focuses on the generation of flow-based network traffic data and proposes a comparative study of several GAN architectures, including Wasserstein, conditional, Energy-Based, gradient penalty, and LSTM-based generative networks. We collect and process a NetFlow-based [6] dataset from a number of 16 subjects while exploring several methods of data processing and adapting NetFlow data in order to prepare it for GAN input. We measure the performance of the GAN models according to three metrics that measure networking domain-specific rules, data distribution, and temporal distribution. Finally, we apply the best-performing GAN model to an IDS setup and measure the performance of an intrusion detection network with and without artificial samples.

3 METHODS CONSIDERED IN THE COMPARATIVE STUDY

Recent years have seen a significant rise in the introduction of GAN-based methods for the generation of artificial data, especially in fields like image [19], text [22] and music [8] generation. However, even given the importance of network and computer security, lesser attention has been given to the generation of artificial network traffic data, as shown in Section 2. It is, therefore, imperative at this point to not only propose a large set of GAN-based approaches in order to find patterns regarding the GAN architectures that best emulate real-world network traffic data but also test their usefulness in a network security scenario. Given this, and starting from the basic principles of generative networks [10], a diagram of how GANs are able to create network data is presented in Figure 1. The role of the Generator Network is to create, starting from a noise vector of N elements $z = [z_1, z_2, ...z_N]$, artificial network traffic samples that the Discriminator Network will then have the role of distinguishing, in an adversarial process, from real-world network traffic samples. We propose several GAN approaches, based on the following architectures: (i) Wasserstein GANs [4], (ii) conditional GANs [21], (iii) Energy-based GANs [34], (iv) gradient penalty WGANs [12], (v) LSTM GANs [23].

As is the case for many network traffic research works [2, 3, 18], we will use flow-based network traffic as data samples for training and testing our proposed method. NetFlow [6] is a unidirectional traffic format where

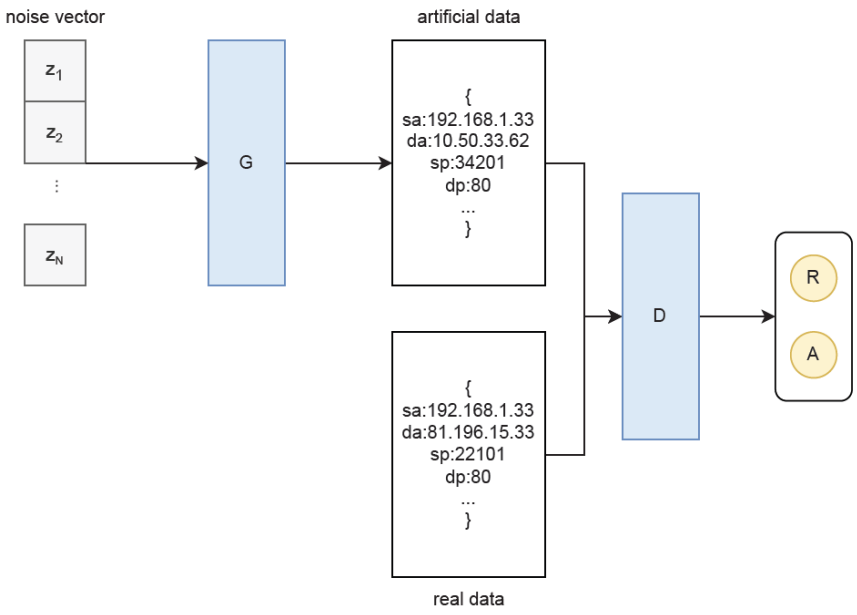

Fig. 1. General architecture functioning principles of GANs applied to network traffic data. Taking a noise vector z as input, the role of the generator network $\mathcal{G}$ is to create new samples of artificial network data, while the role of the discriminator network $\mathcal{D}$ is to differentiate between the artificially created data and real-world data, captured from network users in different scenarios.

all network packets are grouped around events with the same source and destination IP addresses, ports, and communication protocols. All packets in a predefined time duration with these five attributes in common are gathered and joined in a single flow, giving a condensed but still accurate view of network activity. The final list of attributes of a flow according to the NetFlow specification is diverse and contains but is not limited to details regarding the IP addresses and ports of the startpoint and endpoint of the communication, TCP and HTTP flags, timestamps for the beginning and end of the flow, communication protocol and direction, and data regarding the number of packets and the quantity of data transferred in each flow. More details regarding our approach to capturing and processing NetFlow data can be found in Section 4.1.

3.1 General framework

We start our experiments by building a general GAN framework that creates a baseline of architectures representing building blocks for the final set of experiments. Figure 2 presents a diagram of the proposed GAN framework. This architecture can be applied to both the Generator and Discriminator networks, and variations to it can be created by using the three architecture parameters: x, y, and z. Obviously, while for the Generator network, the input would have the size and shape of the noise vector, and the output would represent a NetFlow sample, for the Discriminator, the input would have the size and shape of a NetFlow sample, while a decision on the nature of the sample (artificially generated or real-world sample) would represent the output. This framework is composed of fully connected layers. The first internal variable of this framework, x, has the role of defining the size of the first layer, 2^x. This type of layer with a size of 2^x can be repeated z times, followed by a number of $|y|$ layers,

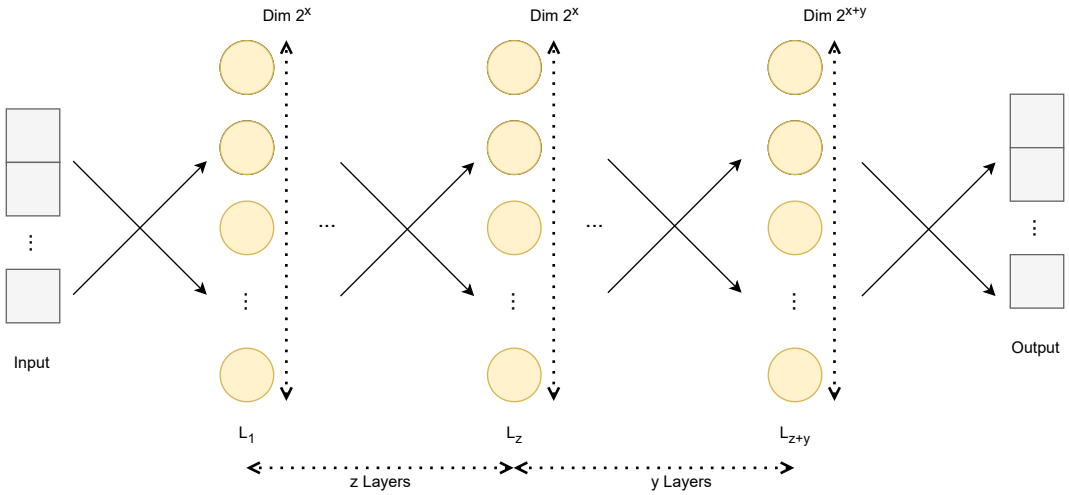

Fig. 2. General structure of our GAN framework that can be applied to both the Generator and Discriminator networks by varying the inputs and outputs and the three x, y, and z parameters accordingly. The implementations of WGAN, cGAN, EB-GAN, GP-GAN, and LSTM-GAN are built upon this general structure by adding the necessary modifications to it.

each of them changing the size of the respective fully connected layer with the formula 2^{x+i}, where $i \in [1, y]$. This allows us to achieve great flexibility in creating and testing variations of the network. For example, when creating a model with a decreasing number of neurons per layer, all we would have to do is set $y < 0$ and $z = 0$, while for a model with increasing number of neurons, we would have to set $y > 0$ and $z = 0$, and for a model with constant layer sizes we would set $y = 0$ and $z > 0$. Of course, many more such configurations can be available and can be created.

3.2 Model types

3.2.1 Wasserstein GAN. The Wasserstein GAN or WGAN architecture [4] is a modification of the original GAN architecture [10], where the Earth-Mover or EM distance is used instead of the Jensen-Shannon (JS) divergence, with the goal of improving the stability and quality of the generated samples. In our implementation of WGAN architectures, given z the input noise vector, $\mathcal{G}$ a function representing the output of the generator network, $G = \mathcal{G}(z)$ the set of generated flow instances at training time and T the set of real-world test flows, and $\mathcal{D}$ the discriminator output for a set of samples, the Wasserstein loss computation for the generator network ($\mathcal{L}_g$) and for the discriminator network ($\mathcal{L}_d$) can be expressed as:

$$\mathcal{L}_g = -\frac{1}{n}\sum_{i=1}^{n}\mathcal{D}(G) \tag{1}$$

$$\mathcal{L}_d = -(\frac{1}{n}\sum_{i=1}^{n}\mathcal{D}(G) \cdot \frac{1}{m}\sum_{i=1}^{m}\mathcal{D}(T)) \tag{2}$$

3.2.2 Conditional GAN. Unlike other types of GANs, the conditional GAN (cGAN) architecture [21] takes an additional vector of restrictions c that can represent additional information, such as class labels, attribute values, or other auxiliary data. The goal of the cGAN is to learn a mapping from the conditional vector and a noise vector

to a realistic output that matches the specified conditions. This can be useful in various applications, such as image-to-image translation, text-to-image synthesis, and data augmentation. In our implementation, we feed labeled information regarding the type of application that creates each individual NetFlow (see Section 4.1.1 for more information on the type of applications).

3.2.3 Energy-Based GAN. For the Energy-Based GAN (EBGAN) approach [34], the discriminator is viewed as an energy function. Concretely, while the generator is trained to produce samples with minimal energy, the discriminator is trained to assign low energies near the real-world data manifold and high energies for artificially generated data. The energy of the data is expressed as a function of the data features, using an encoder-decoder architecture for creating and processing the features. Given a positive margin m, we implement the loss function according to [34]:

$$\mathcal{L}_g = \mathcal{D}(G) \tag{3}$$

$$\mathcal{L}_d = \mathcal{D}(T) + max(0, m - \mathcal{D}(G)) \tag{4}$$

3.2.4 Wasserstein GAN with gradient penalty. In addition to the classical WGAN approach, a gradient penalty term is added to the Wasserstein loss (WGAN-GP), as introduced in [12], with the role of enforcing the Lipschitz continuity of the discriminator. The gradient penalty term encourages the gradient of the discriminator with respect to its input to have a norm of one, which ensures that the discriminator's function is Lipschitz continuous with a Lipschitz constant of one. This approach has several benefits compared to the original Wasserstein GAN. First, it leads to a more stable training process, as the gradient penalty term helps to prevent mode collapse and other common issues that can arise during training. Second, it allows the use of deeper and more powerful discriminator networks, which can help to improve the quality of the generated images by forcing the generator to improve the artificial samples it creates.

3.2.5 LSTM-GAN. The power of LSTMs [14] has long been established in tasks that involve sequential data, such as natural language processing, speech recognition, video analysis, and time series forecasting. They excel at capturing long-term dependencies in sequential data, which is difficult for traditional recurrent neural networks. We theorize that LSTM-GANs [33] may be able to not only create meaningful data with regards to the individual NetFlow samples but also arrange these samples in an optimal way, understanding the sequences that appear during normal network use. We propose integrating LSTM layers into the WGAN-GP architecture.

3.3 Implementation details

We empirically tested and set the limits associated with the (x, y, z) parameters of the general GAN framework and obtained and used the following results throughout our experiments: $x \in [5, 9]$, $y \in [-3, 3]$, $z \in [0, 4]$. An Adam optimizer [15] was used for all the GAN variants, with a learning rate of 0.001 for the generator network and 0.005 for the discriminator, a weight decay of 0.01, and β_1 and β_2 equal to 0.9 and 0.999. The networks are implemented using the popular PyTorch library [2], while training is performed for 50 epochs, with a batch size of 64 samples. Weight initialization is performed using the Kaiming (or He) initialization scheme [13], given its benefits in preventing vanishing and exploding gradients in the training process of deep neural networks, and its good results reported when used with ReLU activation units. Furthermore, in order to preserve a set of stable initial conditions, we applied the *manual_seed*() PyTorch function when initializing the weights of the GAN networks using the Kaiming method.

[2]https://pytorch.org/

4 EXPERIMENTAL SETUP

In the following section, we will present the experimental setup utilized to validate our proposed set of approaches. We will start by describing the dataset we created for training and testing our generative models, dubbed UPBFlow, looking into details on data acquisition, post-processing, distribution, and splits. After this, we analyze the external datasets used in our additional studies on the usefulness of a generative approach for traffic flow generation. Finally, we will discuss the main metrics used in computing the performance of our systems.

4.1 The UPBFlow dataset

As previously mentioned, we are interested in modeling and generating network traffic based on NetFlow [6] data. We therefore propose a protocol for building a new dataset, using several tools to help capture the data and post-process it in order to increase its usefulness and readability. Starting from the data collection method, we decided on using two different applications for capturing network traffic, namely WinPcap[3] for Windows systems, including installing its proprietary virtual driver, and tcpdump[4] for Linux systems. These applications are able to monitor one of the network devices on our subjects' computers and capture network traffic data accordingly. Following this step, we instructed the subjects to acquire data from at least 5 different sessions on different days and provide the capture files to us while also specifying a profile for each different file. We defined these profiles as:

- work profile - signifying that the computer was used during that session for work or office-related tasks, in cases where working on their own computer or laptop and recording traffic flow data was permitted by their respective companies or employers;
- study profile - associated with tasks and actions related to studying, such as searching for tutorials, either text-based or video tutorials, using educational platforms for independent or university-related study;
- home profile - a general profile where subjects are out of their work or study environments, either browsing based on their hobbies, gaming, watching or creating streaming content, etc.

While some of the activities in these profiles may overlap, for example, listening to music can be attributed to a home type of usage, but subjects may also listen to music while they are working or studying, our instructions state that we are interested in the primary goal of a session, whether it is work, study or home-use. Furthermore, we believe that allowing activities to overlap creates a more realistic image of user actions on the network.

4.1.1 Post-processing. Our main target in developing post-processing methods is creating better and more human-readable data for training our GAN-based methods. Therefore, we propose a framework for adding additional information to the captured flows that would be able to transform source and destination IPs into application or service names, thus allowing humans to better understand the subject's activity at any given time. We firstly use the nfdump[5] toolset for reading and transforming the NetFlow files into a human-readable csv format. Furthermore, in order to correctly identify the applications or entities that certain IP addresses belong to, we used the autonomous system number (ASN). ASN is a system where certain registered entities have common routing prefixes. For example, at the time of writing this, the "GOOGLE US" entity has the following ASNs: "ASN1516", "ASN22859", "ASN36039", "ASN394507", "ASN394639" Thus, we then performed an ASN lookup for all our collected flow files, using the pyasn[6] Python package, generating the ASN code, as well as the name of the entity that holds the ASN in a human-readable format. For IP addresses without an assigned ASN (i.e., internal network addresses), we just replaced their ASN value with the string "0".

[3]https://www.winpcap.org
[4]https://www.tcpdump.org/
[5]https://github.com/phaag/nfdump
[6]https://pypi.org/project/pyasn/

Table 1. Overview of the final processed UPBFlow dataset. We present the main attributes or fields of the dataset, their format, their interpretation, and a short example.

Field	Format	Meaning	Example
ts	integer	Starting time of the flow - day	Monday (=1)
hs	datetime	Starting time of the flow - seconds	10800 (=03:00:00)
te	integer	Ending time of the flow - day	Tuesday (=2)
he	datetime	Ending time of the flow - seconds	43321 (=12:02:01)
td	float	Time elapsed between start and end - seconds	11.574
sa	ip	Address of the source	192.168.1.54
da	ip	Address of the destination	192.168.1.86
sp	integer	Port of the source	57434
dp	integer	Port of the destination	53
pr	integer	Protocol of communication	UDP
ipkt	integer	Number of incoming packets	3
ibyt	integer	Number of incoming bytes	1588
opkt	integer	Number of outgoing packets	1
obyt	integer	Number of outgoing bytes	59
flg	boolean array	TCP Flags	U,A,P,R,S,F
asn_src	string	ASN for the source	ASN9808
asn_dst	string	ASN for the destination	ASN15169
app	string	Application or Service using ASN	GOOGLE

Table 2. Overview of the captured data, split by profiles, for the UPBFlow dataset. We show the duration of the recordings, transmitted data size, the number of flows, sessions, and subjects that used each profile, and the total number.

Profile	Duration (hh:mm)	Data size (MB)	No. flows	No. sessions	Subjects
home	14:17	15,313	52,661	47	13
study	22:55	7,988	96,268	24	10
work	07:59	6,690	23,846	4	2
total	45:11	29,991	172,775	75	-

Another step in creating the dataset is changing the format of the datetime features. We consider the exact date of lower importance for our experiments and would like to place a higher priority on the day of the week that corresponds to each flow. Finally, the exact time of day is encoded as the number of seconds from midnight. Thus, our dataset has 18 attributes in its final form, as presented in Table 1. However, considering the redundancy of the time elapsed field and the ending times, we choose to abandon the te and he features, thus not using them when training the systems, and creating the ending time for generated samples as a sum of the generated starting time (ts, hs) and the elapsed time.

4.1.2 Data analysis and data splits. We collected data from 16 subjects, represented by university master students and PhD candidates from Romania, 9 females and 7 males. In total, this data amounted to 45 hours and 11 minutes of network traffic, compressed into 172,775 individual NetFlows. These details, along with the types of profiles represented by this data, are presented in Table 2. First of all, we can observe that a very low number of sessions are dedicated to the work profile. This was perhaps to be expected as a large number of our subjects either do not use their own computers for work, or did not get the permissions to record the data. Secondly, it is interesting to

note that, while a longer duration is dedicated to the study profile, this profile actually has a smaller size in MBs spread over a larger number of flows compared with the home profile. We attribute this to the type of traffic usually associated with home profiles. For example, while the home profiles may involve a large percentage of traffic towards video streaming platforms that transfer large quantities of data, a study profile may involve more textual and tutorial-oriented traffic as well as browsing a significant number of sites in searching for answers to specific technical questions. Finally, we would like to mention the fact that most of the subjects in this data collection experiment returned traffic for more than one profile, most often home and study.

Given this, we divide the available data into three splits: training, testing, and baseline data, using a random selection. We theorize that the best method of splitting the data is not by taking random flows from the entire collection, as this may send NetFlow samples that are part of the same user interactions with a certain website or application to different splits, thus breaking and losing the temporal correlations in the data, but by taking entire sessions and assigning them to one of the three splits, thus keeping the logical temporal sequence of flows intact. The *training* set is composed of 37 sessions, covering almost 90,000 flow samples, the *testing* set is composed of 32 session, covering over 73,000 flow samples, while the *baseline* set is composed of the final 6 sessions, covering almost 10,000 flow samples. We use the training set for training the networks and the testing data for comparing the results generated during the training phase. Given a set of generated samples during training G, the testing set real-world data T, and a comparative metric m, we can compute the performance of a network as the result of $m(G, T)$. Finally, the baseline split is used as a method of measuring what a "maximal" performance could be. We treat the baseline samples B as a generated set of samples and compute the metrics $m(B, T)$, thus obtaining a practical "maximum value" that can be attained by the network. It is highly unlikely that generative networks will obtain better performances than these baseline values, as this would mean that there is low variation in the generated data (i.e., most likely, we will obtain $m(G, T) < m(B, T)$) across all metrics that measure performance.

4.1.3 Metrics. We propose three metrics for measuring the performance of the proposed GAN approaches on the UPBFlow dataset. The first metric consists of *Domain Knowledge Checks* (DKC), inspired by the work of Ring et al. [25]. We implement a series of networking-specific rules that all generated flows must obey; otherwise, they will be considered invalid from a technical networking standpoint. These rules are as follows:

- TCP flags must only be present when the communication protocol is set to TCP;
- at least one IP address must be internal or local in a communication flow (i.e., 192.168.xxx.xxx);
- if the source or destination ports are 80 or 443, then the flow represents an HTTP or HTTPS communication, and therefore, the protocol must be TCP;
- similarly, when the port is 53, the protocol must be UDP;
- a destination port of 137 or 138 represents a NetBIOS message; therefore, the source IP must be internal, while the destination IP must have an internal broadcast address (192.168.xxx.255);
- the generated packets and bytes must adhere to the same limits as the original data in UPBFlow, therefore: $34 * nmb\_packets \leq nmb\_bytes \leq 65535 * nmb\_packets$.

The DKC metric for a set G composed of N artificially-created flows, with $G = \{g_0, g_1, ...g_N\}$ can be expressed using the following equations:

$$DKC(G) = \frac{\sum_{i=1}^{N} DKC(g_i)}{N} \tag{5}$$

$$DKC(g_i) = \frac{\sum_{i=1}^{6} DKG_j(g_i)}{6} \tag{6}$$

where DKC_j symbolizes the six DKC rules, and the value for $DKC_j(g_i)$ is 1 if a rule is passed and 0 if it is broken. Therefore, higher values for the final DKC of the generated set symbolize a better set.

Secondly, we propose a metric based on computing the *Euclidean Distance* (ED) between the distribution of the generated data and the distribution of the original UPBFlow dataset. More to the point, we measure the ED between the distributions of the following data attributes: source port, destination port, time elapsed, protocol, number of bytes and of packets, application, and finally, the communication type defined as incoming or outgoing. We collect and define 10 components for the Euclidean Distance metric with the following special processing considerations:

- *ctype* defines the type of communication - incoming external (defines incoming traffic from an external IP address), outgoing external (defines outgoing traffic from an external IP address), incoming internal (defines incoming traffic within the local network), and outgoing internal (defines outgoing traffic within the local network).
- *ipaddr* We compute Euclidean Distances between IP sets by transforming the addresses to integer values. Theoretically, by using this method, we can ensure that differences in the first byte of the address are much more significant than differences in the last byte of the address, simulating the way real-world networks behave.
- *portsrc* and *portdst* - defines the port numbers for the source and destination of the flow. Considering the inherent randomness in network communications, we consider all the ephemeral ports, as defined in [11] and on IANA's website [7], as the same port number when computing the ED.
- *timediff* defines the Euclidean Distance for time differences.
- *proto* defines the Euclidean Distance for the protocol, having the following valid values: TCP, UPD, and ICMP.
- *flgs* defines the Euclidean Distance for the TCP flags, having six possible boolean sub-components.
- *bytes* defines the Euclidean Distance for the number of bytes assigned to the flow. The direction of the bytes is ignored, as this is actually defined with the *ctype* component.
- *packs* defines the Euclidean Distance for the number of packets assigned to the flow. The direction is again ignored, similar to the *bytes* component.
- *app* defines the Euclidean Distance for the application involved in the flow.

While flows contain date and time information, we do not assign the processing of temporal correlations to the ED metric, but use a third, separate temporal metric. In a similar manner to DKC, given ED_j the ED computation for each of the 10 analyzed attributes, and T the original dataset used as testing set:

$$ED(G,T) = \frac{\sum_{i=1}^{10} ED_j(G,T)}{10} \tag{7}$$

Finally, we create a third *Temporal Distance* (TD) metric that compares the temporal distribution of the generated data with the temporal distribution of the original data via Euclidean Distance. We selected two attributes for computation, namely the applications and number of bytes for the communication flows. In order to allow for a small variation of these two distributions, we divided each day into 15-minute intervals representing the resolution of our distribution computation, giving a total of 96 intervals per day and a total of 672 intervals per week. The TD metric is then computed in a similar manner to ED:

$$TD(G) = \frac{\sum_{i=1}^{2} TD_j(G,T)}{2} \tag{8}$$

[7] https://www.iana.org/assignments/service-names-port-numbers/service-names-port-numbers.xhtml

4.2 Input Transformation

As presented in Table 1, the attributes that compose the individual flows in our dataset have fundamentally different formats, some of them being datetime-like variables, others being strings, integers, floating point numbers, and boolean values. Therefore, we must transform this data before using it as input for the proposed networks. Following the work of [25], we use three input processing methods, namely: numerical transformation, binary transformation, and embedding transformation.

The *numerical transformation* takes the entire set of attributes and transforms them into floating point values based on a normalization scheme. For example, while IP address values can be interpreted as strings or as an array of strings, the four values contained in an IP address can be normalized based on their maximum value of 255. For example, an IP address of 192.168.1.100 can be divided into four parts, and have each of them divided by 255. Therefore each byte component of the IP would be transformed into: $192/255 = 0.7529$, $168/255 = 0.6588$, $1/255 = 0.0039$, and $100/255 = 0.3921$. Similar methods can be applied to other integer values in the attribute set. For example, the port number can be divided by the maximum possible port number 65535, while the number of bytes and packets can be divided by the maximum number found in the dataset. The boolean attributes can be represented as an array of zeros and ones

The *binary transformation* scheme transforms all the attributes in the dataset into binary-coded values. For example, when taking IP addresses like 192.168.1.100, each byte would be transformed into its binary representation (i.e., 192 would be transformed into 11000000). Other attributes, like the number of bytes and packets, can be represented as 32-bit integers, while port numbers can be transformed into 16-bit integers.

Finally, the *embedding transformation* uses the IP2Vec [24] approach, a network traffic-based extension of the popular Word2Vec [20] method, in order to transform individual attributes into embeddings, using a single hidden layer for embedding the attributes, according to the method presented in [25].

4.3 Network intrusion detection

Finally, we propose a set of experiments designed to test the utility of the generated NetFlows as additional training data for a network intrusion detection setup. We, therefore, use the popular CIC-IDS2017 [27] dataset for performing these experiments. The data associated with this dataset contains recordings from 25 users using either Windows or Linux machines and consists of a large number of types of attacks, including but not limited to brute force, heartbleed, botnets, DoS, DDoS, web attacks, and port scans. These attacks are recorded over a period of 5 days, with only the first one being attack-free. We propose using the best-performing GAN architecture on the UPBFlow dataset and creating a set of additional artificially generated attacks in order to augment the training dataset for these types of attacks. Overall, the CIC-IDS2017 dataset has over 1.7 million samples assigned to the benign class, while the attack classes are generally underrepresented. Performances on this dataset are computed via accuracy, precision, and recall for each of the attack classes. We choose the following classes of attacks for processing and enhancing via our chosen GAN model: (i) DoS attacks, with over 280,000 samples in the training set, (ii) PortScans, with over 110,000 samples, (iii) FTP-Patator with almost 6,000 samples, (iv) SSH-Patator with almost 4,500 samples, and (v) Bot, featuring approximately 1,400 samples.

5 RESULTS

Given the set of studied architectures (Section 3.2) and the proposed data transformation schemes (Section 4.2), we analyze the results with regards to the quality of the generated NetFlow samples data, using the UPBFlow dataset, and the improvement that our GANs bring to the detection of network attacks, using the CIC-IDS2017 [27] dataset.

Table 3. Overview of the final results on the UPBFlow dataset. The 5 architectures (WGAN, cGAN, EB-GAN, WGAN-GP, LSTM-GAN) are compared against the maximal baseline created by the baseline split. The best-performing GAN architecture for each of the three metrics is presented in bold. For each architecture, we present the best-performing (x, y, z) sets for the Generator ($\mathcal{G}$) and Discriminator ($\mathcal{D}$). For the three metrics, the ↑ sign indicates that higher values are better, while the ↓ sign indicates that lower values are better. Also presented are the median performances and relative standard deviation (RSD) across 15 repeating runs for the best-performing (x, y, z) sets for the Generator and Discriminator.

	Best performances					Median performances (RSD)		
	DKC ↑	ED ↓	TD ↓	$\mathcal{G}$ (x, y, z)	$\mathcal{D}$ (x, y, z)	DKC ↑	ED ↓	TD ↓
WGAN	0.8937	0.4954	0.4910	(6, 2, 0)	(9, -2, 0)	0.8531 (6.47)	0.5212 (10.23)	0.5485 (5.89)
cGAN	0.8941	0.455	0.5268	(6, 3, 0)	(7, -3, 1)	0.8619 (4.38)	0.5107 (7.13)	0.5441 (6.71)
EB-GAN	0.8944	0.4483	0.4681	(7, 2, 2)	(6, 2, 1)	0.8618 (9.21)	0.4710 (10.14)	0.5162 (6.92)
WGAN-GP	0.8873	0.5263	0.4615	(6, 2, 1)	(7, -1, 0)	0.8358 (7.37)	0.5613 (6.82)	0.5109 (11.58)
LSTM-GAN	**0.9114**	**0.3917**	**0.3891**	**(6, 3, 0)**	**(7, -1, 1)**	0.8640 (8.88)	0.4409 (9.86)	0.4311 (12.7)
maximal baseline	0.9349	0.2381	0.3218	–	–	0.9349	0.2381	0.3218

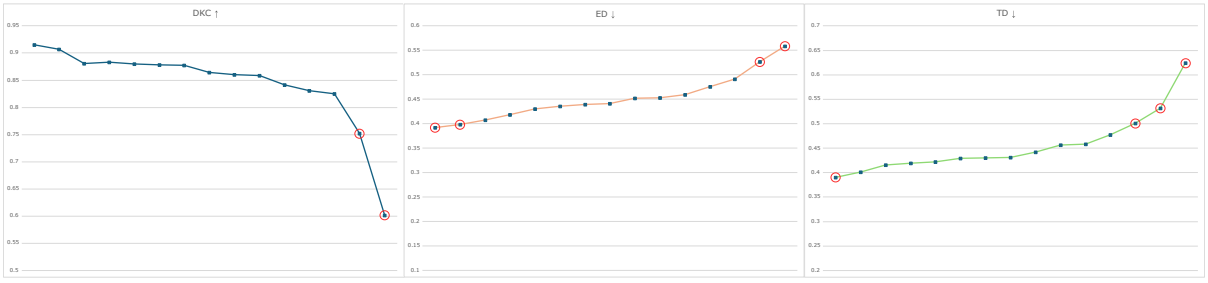

Fig. 3. Variations in results for the best-performing network (LSTM-GAN), for the optimal architecture settings for the Generator and Discriminator networks ($\mathcal{G}(6, 2, 0)$, $\mathcal{D}(6, -1, 1)$). Outlier results, both positive and negative, are circled.

5.1 NetFlow data generation

Given the unstable nature of GAN training, we performed training 15 times for each individual experiment, i.e., for each combination of GAN type and (x, y, z) parameter set. As mentioned in Section 3.3, weight initialization is performed using a Kaiming method, which should further stabilize the results of GAN training. We start by analyzing the performance of each type of GAN on the UPBFlow dataset, as presented in Table 3. The table shows that the best-performing architecture is the LSTM-GAN approach, which combines LSTM layers with a WGAN loss approach improved with gradient penalty. The results for this method are as follows: a 0.9114 DKC value, 0,3917 ED value, and a TD value of 0.3891, representing a 2.57%, 39.21%, and 17.29% degradation over the maximal baseline. It is interesting to notice that, while the maximal baseline has a good overall performance, there still are considerable differences between the baseline and testing set, signifying a high variety in the acquired data. The performance of LSTM-GANs in our case surpasses the performance of the other proposed networks, usually by a significant margin: 1.89% for DKC compared with the second-best approach, 15.39% for ED, and 17.8% improvement for TD.

Table 3 also presents the median values and relative standard deviations (RSD) recorded for each type of GAN across all 15 training instances, showing the GAN (x, y, z) parameter sets which achieve top performance. We use the relative standard deviation to describe the variation in these results, as we wish to express it as a percentage of the results recorded across the different runs. Figure 3 presents the results of the training process for the best-performing architecture, namely the LSTM-GAN with $\mathcal{G}(6, 3, 0)$ and $\mathcal{D}(7, -1, 1)$, across all 15 repetitions

of the training process. We observe some interesting trends in this analysis. Firstly, it would seem that the top-performing LSTM-GAN architecture generally shows high deviations regarding the spread of the results over the 15 repeats of the training and testing processes. Concretely, it shows an RSD of 12.7% for the TD metric (highest among all types of GANs), 9.86% for the ED metric (only third highest), and 8.88% (second highest). We propose the hypothesis that, especially in the case of the TD metric, some training variants of the LSTM-GAN networks were able to correctly learn the temporal distribution of the training data, something that would be achieved for other GAN variants, while other variants of LSTM-GAN were unable to learn this distribution due to unfavorable network initialization. Also, as shown in Figure 3, some of the runs show very low performances, indicating a decay in the training process. Concretely, in the example shown in the figure, the following intervals fit in one RSD: $I_{DKC} = [0.9185, 0.7687]$, $I_{ED} = [0.4073, 0.4965]$, $I_{TD} = [0.3974, 0.5130]$, and outlier results outside this distribution are presented with a red circle. We obtained 2 negative outlier variations according to the DKC metric, 2 positive and 2 negative outliers on ED, and 1 positive and 3 negative outliers for the TD metric.

Generally, the three proposed input transformation schemes greatly influenced the final results. We conducted a series of experiments with the three schemes, targeting the WGAN architecture in order to select the best-performing scheme. While testing a variety of parameters, including different values for the (x, y, z) set on both the discriminator and generator, we obtained the following top (DKG, ED, TD) values: WGAN-numerical $(0.9011, 0.5417, 0.4871)$, WGAN-binary $(0.8916, 0.5184, 0.496)$, WGAN-embedding $(0.875, 0.5881, 0.5014)$. These results seem to favor the numerical and binary approaches. Therefore, we decided to perform an additional battery of tests where we attempt to combine these two approaches. We thus empirically tested alternating each feature representation between numerical and binary, obtaining a better set of results: $(0.8937, 0.4954, 0.491)$. The following features are represented as numerical, according to the abbreviations listed in Table 1: ipkt, ibyt, opkt, obyt, and td, while the other features are computed as binary.

In a similar manner, we deduced a set of network hyperparameters that we used throughout our experiments. We tested a set of network hyperparameters that include learning rates for the generator and discriminator, optimizers, weight decay, β_1 and β_2 parameters, number of epochs, and batch size. These variations were part of an initial set of experiments, again targeting the WGAN architecture, that resulted in the values for these parameters as presented in Section 3.3. While we realize that the hyperparameters we deduced in this set of experiments may not be the optimal ones for different types of GAN networks, the WGAN approach is at the core of many of the other types of GANs implemented in this paper. Thus, we propose that, at least partially, these values may represent an acceptable set of initial hyperparameters that we can use in the rest of our experiments.

The top performing LSTM-GAN architecture is composed of a leading LSTM layer, which is followed by the blocks of fully connected (FC) layers that compose the General Framework, as shown in Figure 2, with each fc layer being followed by a LeakyReLU operation [13]. For the generator network, the component fc layers have the following number of neurons: $\{64, 128, 256, 512\}$, while the discriminator network has the following dimensions for the fc layers: $\{128, 128, 64\}$.

5.2 Statistical significance analysis

In the following analysis, we wish to measure whether the results presented in the previous section can be considered statistically significant. Concretely, we want to see if there is an observable statistical difference between data collected from different variations of GANs. To do this, we propose using the independent-samples t-test, which can test whether there is a statistically significant difference between two independent groups of observations. This test measures the mean values in two unrelated groups, u_1 and u_2, composed of independent populations, $u_1 = [u_{1,1}, u_{1,2}, ..., u_{1,x}]$ and $u_2 = [u_{2,1}, u_{2,2}, ..., u_{2,x}]$, with the null hypothesis being formulated as: $H_0 : u_1 = u_2$. Given the following attributes of the generated flows:

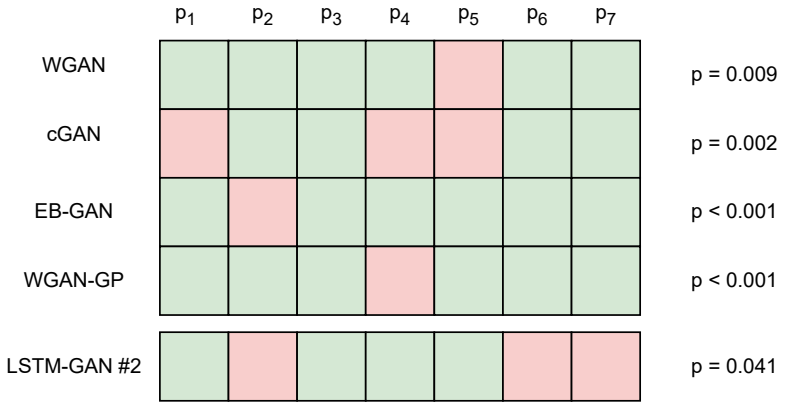

Fig. 4. Analysis of the differences between the samples produced by the top performing GAN architectures of each type. We compare the architecture with the best overall result according to Table 3 (LSTM-GAN), with the best performing WGAN, cGAN, EB-GAN, and WGAN-GP networks, as well as the second-best performing LSTM-GAN network. We present the results of the t-test on each of the 7 components, as well as the overall p-value computed as an average of the 7 components. Components that test as statistically different ($p < 0.05$) are marked in green, and components that are not statistically different in red.

- a_1 application, encoded as integer numbers, arranged according to the frequency of their appearance in the UPBFlow dataset: the most common application would be coded as the first element in the population of a_1, the second most common as the second element, and so on. As the application is closely tied to the IP addresses and ASN numbers, we believe this to be a good representation of the communication endpoints generated by the GANs.
- a_2 and a_3 incoming and outgoing ports.
- a_4 and a_5 incoming and outgoing bytes.
- a_6 time between start and end, encoded as milliseconds.
- a_7 starting time of the flow, encoded according to the method presented in Section 4.1.3, via temporal distance coding.

We present results of statistical significance for all these components, as well as the final p-value, computed as their average:

$$p_F = \frac{\sum_1^7 p(a_i)}{7} \tag{9}$$

Figure 4 presents the results for t-tests performed between the architecture with the best results, i.e., LSTM-GAN with $\mathcal{G}(6, 3, 0)$ and $\mathcal{D}(7, -1, 1)$, and the top performing WGAN, cGAN, EB-GAN, and WGAN-GP combinations. We can observe that the majority of the 7 components can be considered statistically different ($p < 0.05$) and are marked as green. The highest p_F value in these experiments is achieved for the WGAN network, with $p_F = 0.009$, < 0.05. We also present a comparison between the best and second best (denoted as LSTM-GAN #2) LSTM-GAN architectures, with a value of $p_F = 0.041$, < 0.05. While this value indicates statistically significant differences exists in the two sets of generated network data, it is rather close to the 0.05 limit. We attribute this phenomenon to the fact that LSTM-GANs are by far the best at encoding temporal information, and therefore, its variations

Table 4. Analysis of the DKC metric on the UPBFlow dataset for the 5 architectures (WGAN, cGAN, EB-GAN, WGAN-GP, LSTM-GAN) Each of the six DKC components are presented, along with the final DKC score.

	WGAN	cGAN	EB-GAN	WGAN-GP	LSTM-GAN
DKC1	0.7423	0.871	0.9001	0.8831	0.8923
DKC2	0.9594	0.9017	0.91	0.8736	0.9016
DKC3	0.8875	0.8913	0.8731	0.8915	0.9042
DKC4	0.8733	0.849	0.8633	0.859	0.8907
DKC5	0.9298	0.9134	0.915	0.9125	0.9353
DKC6	0.9696	0.9211	0.9036	0.9038	0.9441
DKC	0.8937	0.8913	0.8941	0.8873	0.9114

Table 5. Analysis of the ED metric on the UPBFlow dataset for the 5 architectures (WGAN, cGAN, EB-GAN, WGAN-GP, LSTM-GAN) Each of the eleven ED components are presented, along with the final ED score.

	WGAN	cGAN	EB-GAN	WGAN-GP	LSTM-GAN
ctype	0.2521	0.3379	0.1955	0.3352	0.1592
ipaddr	0.3072	0.3778	0.3940	0.2147	0.2466
portsrc	0.6174	0.5187	0.5533	0.5865	0.2419
portdst	0.5180	0.4295	0.4553	0.5885	0.2316
timediff	0.4016	0.6007	0.2716	0.4988	0.4617
proto	0.4676	0.5120	0.5419	0.4606	0.5409
flgs	0.5250	0.3463	0.6585	0.5611	0.4709
bytes	0.7182	0.6314	0.6780	0.7247	0.5730
packs	0.7252	0.4257	0.6862	0.7356	0.6441
app	0.4213	0.3396	0.1483	0.5568	0.3465
ED	0.4954	0.4520	0.4583	0.5263	0.3917

that have a good performance may also produce similar results with regard to the $p(a_7)$ component, which in this particular case scored $p(a_7) = 0.12$.

5.3 Generated NetFlow analysis

An analysis of the top-performing systems from each architecture with regard to the components of the DKC metric is presented in Table 4. Some interesting patterns emerge from this analysis, shedding light on which features are easier or harder to learn for the networks. Firstly, two DKC rules, namely DKC3 ("if the source or destination ports are 80 or 443, then the flow represents an HTTP or HTTPS communication and therefore the protocol must be TC") and DKC4 ("similarly, when the port is 53, the protocol must be UDP") tend to constantly be among the worst performers among the studied architectures. This may signify an inability of the networks to correctly learn the correlation between port numbers and type of protocol. Secondly, DKC5 ("a destination port of 137 or 138 represents a NetBIOS message; therefore the source IP must be internal, while the destination IP must have an internal broadcast address") and DKC6 ("the generated packets and bytes must adhere to the same limits as the original data in UPBFlow, therefore: $34 * nmb\_packets \leq nmb\_bytes \leq 65535 * nmb\_packets$") seem to constantly be among the top performers with regards to DKC scores, indicating good performance for the networks in encoding the correlation between packets and bytes, as well as port numbers and IP addresses.

Table 6. Overview of the results on the IDS task. We present the results for the five classes of attacks (DoS, PortScan, FTP-Patator, SSH-Patator, and Bot), as well as the original number of samples assigned to these classes in the CIC-IDS2017 dataset. We present the results of the IRNN4 architecture, trained with the original dataset as well as the GAN-augmented dataset, and compute the improvement between these two runs. Results that are improved or degraded by more than 1% are presented in bold.

Class	# orig. samples	Dataset version	Acc	P	R
DoS	284,575	original	0.9852	0.9955	**0.8891**
		GAN-augmented	0.9899	0.9954	**0.9261**
		% improvement	0.47%	-0.01%	**4.16%**
PortScan	118,993	original	0.9919	**0.9378**	0.911
		GAN-augmented	0.9933	**0.9629**	0.9104
		% improvement	0.14%	**2.67%**	-0.06%
FTP-Patator	5,970	original	0.9997	0.9321	0.9388
		GAN-augmented	0.9997	0.9346	0.9403
		% improvement	0%	0.26%	0.15%
SSH-Patator	4,419	original	0.9998	**0.9359**	0.9885
		GAN-augmented	0.9998	**0.9779**	0.9878
		% improvement	0%	**4.48%**	-0.07%
Bot	1,471	original	0.9998	0.9395	**0.805**
		GAN-augmented	0.9999	0.9428	**0.8548**
		% improvement	0.01%	0.35%	**6.18%**

Concerning the ED metric, the results of the five proposed architectures, along the ten dimensions of the ED metric, are presented in Table 5. Some general observations regarding the values attained for the components of the ED metric can be deduced from the table. For instance, the values generated for the source port are generally less accurate compared with those generated for the destination port. We attribute this to the possibility that random port numbers are more often assigned to the source ports, when initiating a communication towards services that use http or https destination ports. We also observe that results for the bytes and packet components have a higher distance to the ground truth values compared with the other components. On the other hand, we observe generally good values for the ctype, *ipaddr*, and *app* components. This indicates the learning process is able to thoroughly understand the relationships between the applications and network computers in the dataset.

5.4 Intrusion detection results

Finally, we take present the results for the IDS task using the CIC-IDS2017 dataset. Starting from the comparative analysis published by [30], we implement, train, and test the IRNN4 architecture, as it showed the most promising results according to the authors of the comparative study. The results are shown in Table 6, where we compare the performances of the IRNN4 systems trained with and without GAN-augmented data. We started from the pre-trained LSTM-GAN approach that achieved the best performance in the UPBFlow experiments (see Section 5.1). We adhered to the same training regiment as specified in Section 3.3 when re-training the network with CIC-IDS2017 data. After re-training, we generated three times the number of original samples, thus obtaining a dataset of attack classes four times as big as the original one.

It is interesting to note that accuracy (Acc) results were high enough to begin with. This was perhaps to be expected, given the overwhelming number of True Negative (TN) samples in the dataset (NetFlows that represent normal traffic and are classified by the detector network as such). Therefore, no great enhancements could be

achieved with the GAN augmentation for the Acc metric, with the highest improvement recorded for DoS attacks (0.47%). This, however, changes when looking at the Precision (P) and Recall (R) metrics, as these two metrics do not take into account the TN samples. We highlight four particular instances where improvements over the original training dataset surpass a 1% value, namely: (i) R metric for DoS attacks, with an improvement of 4.16%, (ii) P metric for PortScan attacks, where the improvement is 2.67%, (iii) P metric for SSH-Patator, with an improvement of 4.48%, and (iv) R metric for Bot, with an improvement of 6.18%. It is also encouraging to note that no significant decrease in performance has been noted across the three metrics, with the biggest decrease in performance being a -0.07% change for the recall metric for SSH-Patator.

6 CONCLUSIONS

In this article we presented a study of several types of GAN architectures targeting the generation of NetFlow-like network data. We analyzed five different types of networks, namely Wasserstein, conditional, Energy-Based, gradient penalty, and LSTM GANs, while training and testing them on the UPBFlow dataset, composed of over 170,000 individual real-world NetFlow samples, gathered with the help of 16 participants. We utilized three metrics in comparing the performances of these networks, based on domain-specific rules (DKC), euclidean distance of the distribution of NetFlow samples features (ED), and temporal distance of applications and entities (TD). Results on the UPBFlow dataset show that LSTM architectures display the best performance on this type of data, showing a degradation of 2.57% with respect to DKC, 39.21% for ED, and 17.29% for TD when compared against a maximal baseline. The maximal baseline is computed using a separate part of the UPBFlow dataset, which is compared against the standard testing set associated with this dataset. We performed a set of independent-samples t-test experiments aimed at measuring the statistical differences between flows generated with different GAN architectures and layer compositions. Results show a statistically significant difference between the flows generated with the best-performing LSTM-based architecture ($p = 0.009$ in the worst case), but also between the top two best-performing LSTM-based architectures ($p = 0.041$). We attribute this performance of the LSTM-based architectures to the fact that these types of networks can optimally encode temporal information, thus creating flows that adhere more to the real-world data.

Furthermore, we tested the best-performing LSTM architecture on the popular network intrusion detection dataset CIC-IDS2017, looking to augment some of the attack data and testing the performance of a network intrusion detection model on this dataset with and without the GAN-enhanced data for DoS, PortScan, FTP-Patator, SSH-Patator, and Bot attacks. Results for this experiment are promising, showing significant improvements in the performance of the detection network for many of these attacks, with improvements going up to 6.18% for the recall metric and 4.48% for the precision metric.

ACKNOWLEDGMENTS

Financial support was provided under project AI4Media, a European Excellence Centre for Media, Society and Democracy, H2020 ICT-48-2020, grant #951911. Financial support for Mihai Dogariu was provided by a grant from the National Program for Research of the National Association of Technical Universities - GNAC ARUT 2023.

REFERENCES

[1] Jon J Aho, Alexander W Witt, Carter BF Casey, Nirav Trivedi, and Venkatesh Ramaswamy. 2018. Generating Realistic Data for Network Analytics. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 401–406.

[2] Blake Anderson and David McGrew. 2017. Machine learning for encrypted malware traffic classification: accounting for noisy labels and non-stationarity. In *Proceedings of the 23rd ACM SIGKDD International Conference on knowledge discovery and data mining*. 1723–1732.

[3] Zied Aouini and Adrian Pekar. 2022. NFStream: A flexible network data analysis framework. *Computer Networks* 204 (2022), 108719.

[4] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein generative adversarial networks. In *International conference on machine learning*. PMLR, 214–223.

[5] Adriel Cheng. 2019. PAC-GAN: Packet generation of network traffic using generative adversarial networks. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 0728–0734.

[6] Benoit Claise. 2004. *Cisco systems netflow services export version 9*. Technical Report.

[7] L Dhanabal and SP Shantharajah. 2015. A study on NSL-KDD dataset for intrusion detection system based on classification algorithms. *International journal of advanced research in computer and communication engineering* 4, 6 (2015), 446–452.

[8] Hao-Wen Dong, Wen-Yi Hsiao, Li-Chia Yang, and Yi-Hsuan Yang. 2018. Musegan: Multi-track sequential generative adversarial networks for symbolic music generation and accompaniment. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 32.

[9] Ugo Fiore, Alfredo De Santis, Francesca Perla, Paolo Zanetti, and Francesco Palmieri. 2019. Using generative adversarial networks for improving classification effectiveness in credit card fraud detection. *Information Sciences* 479 (2019), 448–455.

[10] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. 2020. Generative adversarial networks. *Commun. ACM* 63, 11 (2020), 139–144.

[11] Walter Goralski. 2017. *The illustrated network: how TCP/IP works in a modern network*. Morgan Kaufmann.

[12] Ishaan Gulrajani, Faruk Ahmed, Martin Arjovsky, Vincent Dumoulin, and Aaron C Courville. 2017. Improved training of wasserstein gans. *Advances in neural information processing systems* 30 (2017).

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*. 1026–1034.

[14] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.

[15] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[16] WooHo Lee, BongNam Noh, YeonSu Kim, and KiMoon Jeong. 2019. Generation of network traffic using wgan-gp and a dft filter for resolving data imbalance. In *Internet and Distributed Computing Systems: 12th International Conference, IDCS 2019, Naples, Italy, October 10–12, 2019, Proceedings 12*. Springer, 306–317.

[17] Woo Ho Lee, Chae Sang Lim, and Bong Nam Noh. 2020. Generation of Similar Traffic Using GAN for Resolving Data Imbalance. In *Advances in Computer Science and Ubiquitous Computing: CSA-CUTE 2018*. Springer, 1–7.

[18] Francisco Sales de Lima Filho, Frederico AF Silveira, Agostinho de Medeiros Brito Junior, Genoveva Vargas-Solar, and Luiz F Silveira. 2019. Smart detection: an online approach for DoS/DDoS attack detection using machine learning. *Security and Communication Networks* 2019 (2019), 1–15.

[19] Steven Liu, Tongzhou Wang, David Bau, Jun-Yan Zhu, and Antonio Torralba. 2020. Diverse image generation via self-conditioned gans. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*. 14286–14295.

[20] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg S Corrado, and Jeff Dean. 2013. Distributed representations of words and phrases and their compositionality. *Advances in neural information processing systems* 26 (2013).

[21] Mehdi Mirza and Simon Osindero. 2014. Conditional generative adversarial nets. *arXiv preprint arXiv:1411.1784* (2014).

[22] Weili Nie, Nina Narodytska, and Ankit Patel. 2019. Relgan: Relational generative adversarial networks for text generation. In *International conference on learning representations*.

[23] Xu Ouyang, Xi Zhang, Di Ma, and Gady Agam. 2018. Generating image sequence from description with LSTM conditional GAN. In *2018 24th International Conference on Pattern Recognition (ICPR)*. IEEE, 2456–2461.

[24] Markus Ring, Alexander Dallmann, Dieter Landes, and Andreas Hotho. 2017. Ip2vec: Learning similarities between ip addresses. In *2017 IEEE International Conference on Data Mining Workshops (ICDMW)*. IEEE, 657–666.

[25] Markus Ring, Daniel Schlör, Dieter Landes, and Andreas Hotho. 2019. Flow-based network traffic generation using generative adversarial networks. *Computers & Security* 82 (2019), 156–172.

[26] Neelam Rout, Debahuti Mishra, and Manas Kumar Mallick. 2018. Handling imbalanced data: a survey. In *International Proceedings on Advances in Soft Computing, Intelligent Systems and Applications: ASISA 2016*. Springer, 431–443.

[27] Iman Sharafaldin, Arash Habibi Lashkari, and Ali A Ghorbani. 2018. Toward generating a new intrusion detection dataset and intrusion traffic characterization. *ICISSp* 1 (2018), 108–116.

[28] Sungho Suh, Haebom Lee, Paul Lukowicz, and Yong Oh Lee. 2021. CEGAN: Classification Enhancement Generative Adversarial Networks for unraveling data imbalance problems. *Neural Networks* 133 (2021), 69–86.

[29] Ankit Thakkar and Ritika Lohiya. 2020. A review of the advancement in intrusion detection datasets. *Procedia Computer Science* 167 (2020), 636–645.

[30] R Vinayakumar, KP Soman, and Prabaharan Poornachandran. 2019. A comparative analysis of deep learning approaches for network intrusion detection systems (N-IDSs): deep learning for N-IDSs. *International Journal of Digital Crime and Forensics (IJDCF)* 11, 3 (2019), 65–89.

[31] Pan Wang, Shuhang Li, Feng Ye, Zixuan Wang, and Moxuan Zhang. 2020. PacketCGAN: Exploratory study of class imbalance for encrypted traffic classification using CGAN. In *ICC 2020-2020 IEEE International Conference on Communications (ICC)*. IEEE, 1–7.

[32] Yawen Xiao, Jun Wu, and Zongli Lin. 2021. Cancer diagnosis using generative adversarial networks based on deep learning from imbalanced data. *Computers in Biology and Medicine* 135 (2021), 104540.
[33] Yizhe Zhang, Zhe Gan, and Lawrence Carin. 2016. Generating text via adversarial training. In *NIPS workshop on Adversarial Training*, Vol. 21. 21–32.
[34] Junbo Zhao, Michael Mathieu, and Yann LeCun. 2016. Energy-based generative adversarial network. *arXiv preprint arXiv:1609.03126* (2016).
[35] Pasquale Zingo and Andrew Novocin. 2020. Can GAN-generated network traffic be used to train traffic anomaly classifiers?. In *2020 11th IEEE Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*. IEEE, 0540–0545.

Received 31 March 2023; revised xx March xxxx; accepted yy March yyyy