Reproducing Practical GAN-based Synthetic IP Header Trace Generation using NetShare

Akanksha Cheeti, Annus Zulfiqar, Ashwin Nambiar, Syed Hasan Amin, Murayyiam Parvez, Syed Muhammed Abubaker

Purdue University

ABSTRACT

We aim to reproduce the NetShare, a novel GAN-based synthetic header trace generation approach from SIGCOMM'22. This paper proposes Generative Adversarial Neural Networks (GANs) for synthetic packet trace generation for applications such as telemetry and anomaly detection since real-world network traces are a scarce resource. Using GANs for data generation is not a new concept in Machine Learning. However, networking applications are very sensitive to difficult-to-model properties of real network traces (such as inter-arrival time, flow sizes, RTTs) that are hard to capture in synthetically generated traces from simulations or hard-coded models. The novelty in their approach involves identifying and resolving some key fidelity and scalability challenges (along with their tradeoffs) for generating synthetic traces. Our Key evaluations indicate that NetShare generated traces have very high fidelity (similarity of trace properties and distribution) and perform equally well on key downstream applications including Telemetry and Anomaly Detection.

1 INTRODUCTION

Packet and flow level header traces are critical to many network management tasks, for instance they are used to develop new types of anomaly detection algorithms but access to such traces remains challenging due to business and privacy concerns. An alternative is to generate synthetic traces. There are many old approaches to generating synthetic data using simulation-driven methods, model-driven techniques, and machine-learning models. But these approaches have shortcomings as both model, and simulation-driven strategies require domain knowledge and human effort to determine critical workload features and do not generalize well across applications. In contrast, ML models generalize easily but don't capture domain-specific properties.

Here in this approach, they explore the feasibility of ML-based synthetic packet header and flow header trace generation using GANs. In practice, there are several challenges, such as fidelity, scalability-fidelity tradeoffs, and privacy-fidelity tradeoffs, which need to be handled by the existing GAN-based approaches.

In this paper, the authors propose NetShare [9], which can tackle these critical challenges by carefully understanding the limitations of GAN-based methods. They followed the following key ideas in building NetShare to tackle the above issues: (1) Learning synthetic models for a merged flow-level trace across epochs instead of treating header traces from each epoch as an independent tabular dataset. This reformulation captures the intra-and inter-epoch correlations of traces. (2) Data parallelism learning was introduced in this approach to improve the scalability. (3) To deal with privacy concerns for sharing the traces, differentially-private model training was used.

We reproduced NetShare on our testbed and analyzed their evaluations.(1) We Additionally performed evaluations to understand why we need ML to generate synthetic data when it can be done using sampling-based trace generation, it's suitable to generate a copy of data but this approach wouldn't preserve privacy. (2) Fidelity evaluation using metrics like JSD/EMD on PCAP (CAIDA), NetFlow(UGR16) related data provided by the authors and finally on our dataset(PCAP or Netflow) and compared the results.(more in the evaluation section). (3) Evaluation of scores on original vs synthetic on some ML algorithms. (4) How netshare generated traces perform on downstream applications when compared to original traces. More details are provided in the evaluation section.

2 DESIGN

2.1 Preliminaries

2.1.1 Generative Adversarial Network. Generative Adversarial Networks (GANs) estimate generative models through a process corresponding to a minimax two-player game. The setup involves simultaneously training two models with adversarial objectives: a generator G that captures a data distribution to generate real-like data and a discriminator D that distinguishes between real and generated data.

The generator $G(\mathbf{z}; \theta_g)$ can be modelled as a differentiable function that maps input noise variable $\mathbf{z} \in \mathcal{Z}$ to data space $\mathcal{X}$, i.e., $G : \mathcal{Z} \mapsto \mathcal{X}$. On the other hand, the discriminator $D(\mathbf{x}; \theta_d)$ is a function that maps input from data space to the probability that the input data is real, i.e., $D : \mathcal{X} \mapsto [0, 1]$. D is trained to maximize the probability of assigning correct

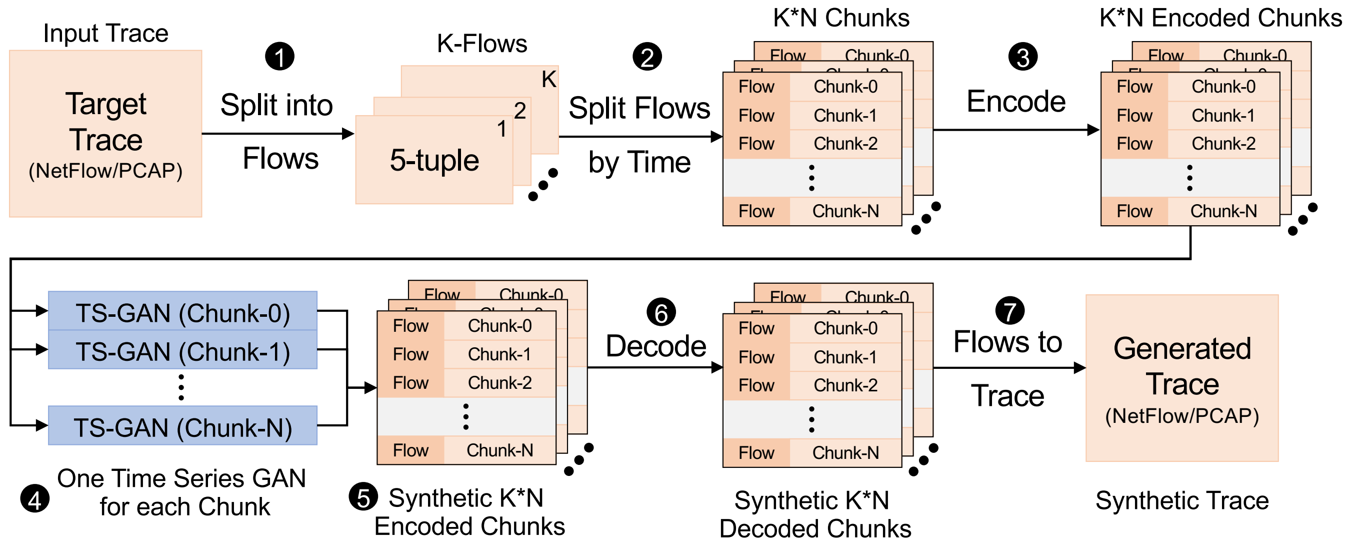


Figure 1: The NetShare [9] trace generation pipeline. The input packet/flow trace is split into a set of flows based on their 5-tuples (K in number), and each of these flows is further split into N chunks based on time duration of the flow. NetShare [9] divided each flow into 10 chunks ($N = 10$). These packet/flow fields of these chunks are encoded using continuous and categorical encoding functions, and these ecoded chunks are fed into NetShare's [9] DoppleGANger [6] time-series GANs. There are N number of GANS in total, each of which learns to generate a specific chunk of any flow. Once trained, these models are used to generate synthetic (encoded) chunks, which are decoded and assembled into a full trace by sorted all the packets/flows by their timestamps.

labels to both real and generated data. G is simultaneously trained to minimize $\log(1 - D(G(\mathbf{z}))$, i.e., the probability of the discriminator correctly identifying generated samples as not real.

The zero-sum game can be framed as the following optimization problem, which in practice is implemented using an iterative, numerical approach:

$$\min_G \max_D \mathbb{E}_{\mathbf{x}\sim p_{data}(\mathbf{x})} \log\left[D(\mathbf{x})\right] + \mathbb{E}_{\mathbf{z}\sim p_z(\mathbf{z})}\left[\log(1 - D(G(\mathbf{z}))\right]$$

2.1.2 GANs for Time Series Modeling. For effective time series generation, recent advancements in GANs for time series data have been proposed. For instance, TimeGAN [10] combines the flexibility of the unsupervised GAN framework with control over conditional temporal dynamics afforded by supervised autoregressive models. An embedding, and associated recovery network, is introduced to provide reversible mapping between features and latent representations while reducing the high dimensionality of the adversarial learning space, capitalizing on the fact that temporal dynamics are often driven by factors aligned with lower dimensional variation.

Relying solely on the discriminator's binary adversarial feedback, as captured by the unsupervised loss $\mathcal{L}_U$ described earlier, may not be be sufficient incentive for the generator to capture temporal dynamics in the data. Effectual embedding and recovery functions are therefore learnt using reconstruction loss $\mathcal{L}_R$ in TimeGAN. The generator is trained to first generate latent representation using noise coupled with a temporally earlier latent representation, and an additional supervised loss $\mathcal{L}_S$ is introduced to further discipline learning. Let $\theta_e, \theta_r, \theta_g, \theta_d$ denote parameters of the embedding, recovery, generator and discriminator networks, respectively, and γ and η be two hyperparameters. The optimization procedure is as follows:

$$\min_{\theta_e,\theta_r}(\gamma\mathcal{L}_S + \mathcal{L}_R)$$
$$\min_{\theta_g}(\eta\mathcal{L}_S + \max_{\theta_d}\mathcal{L}_U)$$

The losses are more formally described below. $\mathbf{x}$, $\hat{\mathbf{x}}$ denote actual and recovered features, y and $\hat{y}$ correspond to whether a sample is real or synthetic, $\mathbf{h}$ denotes latent representation, $\mathbf{z}$ denotes noise, and g is the generator function implemented using a recurrent neural network.

$$\mathcal{L}_R = \mathbb{E}_{\mathbf{x}_{1:T}\sim p}\sum_t \|\mathbf{x}_t - \hat{\mathbf{x}}_t\|_2$$
$$\mathcal{L}_U = \mathbb{E}_{\mathbf{x}_{1:T}\sim p}\sum_t \log y_t + \mathbb{E}_{\mathbf{x}_{1:T}\sim \hat{p}}\sum_t \log(1 - \hat{y}_t)$$
$$\mathcal{L}_S = \mathbb{E}_{\mathbf{x}_{1:T}\sim p}\sum_t \|\mathbf{h}_t - g(\mathbf{h}_{t-1}, \mathbf{z}_t)\|_2$$

2.1.3 DoppelGANger. A recent work—with largely the same team as NetShare—proposes a time series GAN particularly suited to networking data: DoppelGANger [6]. They find that the canonical approaches fail to adequately capture long-term temporal correlations between measurements (packet loss rate, bandwidth, delay) and metadata (e.g., ISP name or location). While they introduce quite a few innovations in the form of an auxiliary discriminator and a batched recurrent neural network generator to address the aforementioned issues, DoppelGANger is still largely a variant of the more standard TimeGAN.

2.2 NetShare

NetShare builds on top of DoppelGANger for improved fidelity, scalability and privacy while focusing on IP header trace generation. It sticks with DoppelGANger as the generative model (while referring to it as just "Time Series GAN"), and primarily introduces innovations in the data pipeline. The key changes and associated motivation are as follows:

(1) *Improved data formatting.* For better fidelity, header trace generation was reformulated as flow-based time series generation problem rather than using a per-epoch tabular approach. This should allow to better capture both intra-measurement and inter-measurement correlations.
(2) *Improved data preprocessing.* For better fidelity, the data features, especially ones with large number of unique values like IP address, are encoded using domain knowledge and ML to make the representation space more amenable to learning. For numerical semantics, like packets/byte per flow with large support value, log transformation is used to reduce the range. `IP2Vec`, an existing representation learning algorithm that transforms input such that similar data based on network context information is close together in the new representation space, is used to transform the categorical port numbers and protocols. The same is not used for IP addresses, and bitwise encoding is used instead, because `IP2Vec` is dependent on training data, which could lead to privacy violation.
(3) *Smart parallelization.* For better scalability, the training pipeline is understandably parallelized. However, naively dividing the entire trace to chunks poses the risk of losing correlations across chunks. Moreover, splitting by a fixed number of packets per chunk may impact differential privacy guarantees, as the presence of any single packet could change the final trained model in an unbounded way. NetShare decides to split by fixed time intervals instead, while adding flow tags to each flow header. Moreover, it utilizes fine tuning for parallel training across chunks while capturing cross-chunk correlations. Specifically, it uses first chunk to get an initial model, and use this as warm start for fine tuning subsequent chunks in parallel.
(4) *Transfer learning with differentially private optimizer.* For ensuring privacy, NetShare is trained using DP-SGD [1]. However, training the model from scratch using DP-SGD deteriorates fidelity, due to inherent privacy-fidelity tradeoffs. To counter this, NetShare first trains the model without DP-SGD on a public dataset, and trains this pretrained model further using DP-SGD on private dataset of interest (a process more popularly known as "transfer learning"). When the public and private dataset are somewhat similar, the model trained on private dataset can converge faster, preserving both fidelity and privacy in the process.

3 IMPLEMENTATION

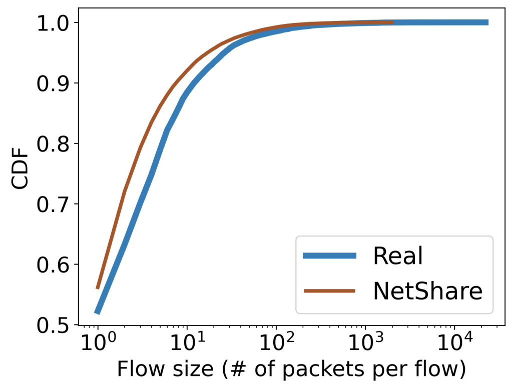
The Netshare implementation is publicly available on Github [4]. But, the implementation differs from the one that the Yin et al have designed. This is because the original implementation was designed to run on a cluster of resource intensive machines. The evaluation testbed for Netshare was 10 Cloudlab machines with each of them having two Intel Xeon Silver 4114 10-core CPUs at 2.20 GHz and 192GB DDR4 memory. [9]

We initially decided that we should re-implement the project to utilize GPU for the compute intensive training , but the existing implementation scaled better on multiple CPUs. The work required for adding support for GPU would have required significant development time and resources.

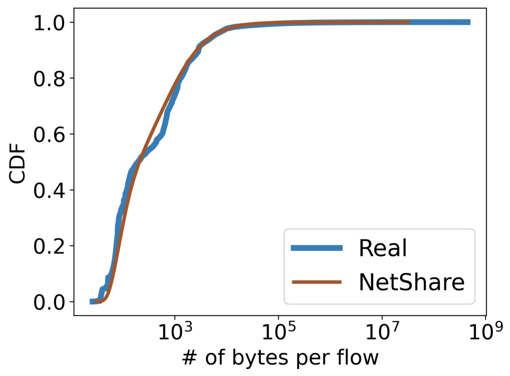
The implementation has multiple versions of the project available and we chose to use the work on the camera-ready as it would have enabled us to recreate the work presented in the original paper. The implementation by design was meant to be run on across various machines. We realized the creating or obtaining such resource would have required lot of monetary resources, hence we decided to invest in setting up a single VM with moderately powerful CPU and GPU for the compute intensive training tasks.

We then decided to make modifications to the existing implementation to enable it to run on a single machine. In the process, we also changed some of the hard coded paths/instructions in the files. This was done in order to enable running the implementation on different machines, with lower setup effort. We developed a Makefile similar to the one provided in the programming assignments to make it easier to run the experiments. And in order to further test the generated traces, we also modified the experiments to enable downstream application testing to determine the quality of the traces.

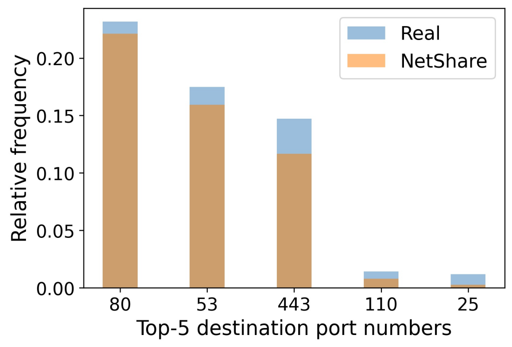
The initial development happened on a 8 core, 32 GB RAM with 75 GB disk VM on Google Cloud Platform (GCP) with access to a Tesla T4 GPU. When running the models with the same configurations as the paper, we noticed we ran into lack of disk memory and RAM (cause the models were stored in



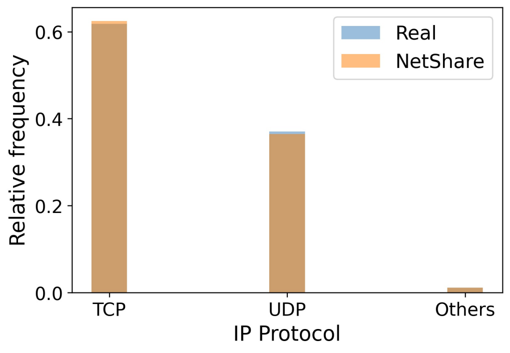
(a) PCAP flow size distribution



(b) NetFlow flow volume distribution



(c) NetFlow destination port distribution



(d) NetFlow protocol distribution

Figure 2: CDF comparisons of various packet/flow properties of raw vs. NetShare-generated [9] traces.

RAM for each epoch). We then switched to a much resource intensive machine with 32 core, 120 GB of RAM with 150 GB disk space to accommodate larger models generated by the program. We figured out the reason for the excess disk space being the frequency of checkpoints for the model being too high, and came up with an alternative checkpoint saving mechanism for the current implementation.

4 EVALUATIONS

We evaluate NetShare [9] based on the fidelity metrics of generated traces and additionally, we perform evaluations on downstream tasks. Similar to the NetShare paper, we PCAP and NetFlow datasets, including CAIDA [3], UGR16 [7] and a Malicious Botnet (P2P) dataset from FlowLens [2]. Flow traces in UGR16 [7] are based on traffic from NetFlow v9 collectors in a Spanish ISP network. Packet traces in CAIDA [3] contain anonymized traffic from a commercial backbone link.

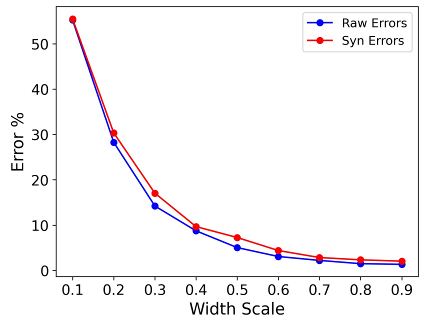
4.1 Fidelity

First, we evaluate the fidelity of NetShare-generated [9] traces on feature distribution metrics.

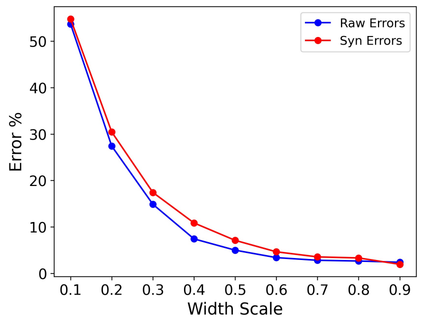
Fidelity measured through header correlations of packets/flows. For URG16 [7], we calculate the CDF of flow volume (bytes per flow) and for CAIDA [3], we calculate the CDF of flow size (no. of packets). Figure 2a and Figure 2b compare the CDFs of these properties between the raw and NetShare-generated [9] traces, which shows a high fidelity in the distribution of these properties.

Fidelity measured through most frequent service ports. We calculated the relative frequencies of the most recurring service ports for the synthetic and real traces. Figure 2c and Figure 2d shows their distributions for the port and three protocol sets (TCP, UDP, and all others), respectively, again indicating a high fidelity between both traces.

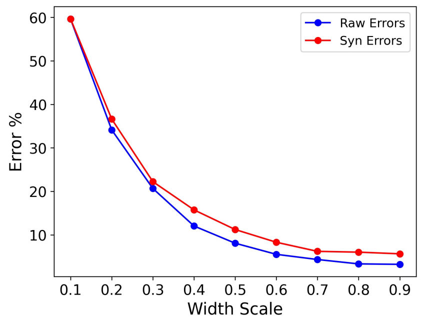
Fidelity measured through Jensen-Shannon divergence. Table 1 shows the JSD calculate for various protocol fields for the synthetic traces generated from NetShare [9] as measure of departure from original raw datasets. The small values indicate that generated traces have high-fidelity in these protocol fields.



(a) Source Address Key



(b) Destination Address Key



(c) 5-Tuple Key

Figure 3: Comparison of Count-Min Sketch error rate performance on top-10% heavy-hitter detection using raw and NetShare-generated [9] CAIDA dataset.

Dataset	Type	SA	DA	SP	DP	PR
Botnet [2]	PCAP	0.25	0.13	0.27	0.30	0.00
CAIDA [3]	PCAP	0.40	0.04	0.22	0.43	0.00
UGR16 [7]	NetFlow	0.17	0.04	0.15	0.15	0.00

Table 1: Jensen-Shannon Divergence (JSD) evaluated for various packet fields on several packet traces.

4.2 Downstream Application Performance

NetShare-generated [9] traces perform similar to raw traces on Telemetry applications. We analyze the downstream task of heavy hitter count estimation with Count-Min Sketch (CMS) [5]. Count-Min Sketch (CMS) is an algorithm that allows us to approximately count the frequency of the events on the streaming data with sub-linear memory requirement. It uses pairwise independent hash functions to store and estimate the count of distinct keys with a certain probability. In our tests, we evaluate the performance of CMS using csiphash [8] hash function to detect top-10% heavy-hitters in the CAIDA [3] dataset. We customize our CMS implementation to modify the memory (in terms of bin count per row) that is available to the real and the synthetic data according to the size of the trace, and evaluate overall error rate for several width settings. We use the following keys for CMS: *Destination IP*, *Source IP*, and *five-tuples* and measure the heavy hitter error rate as following:

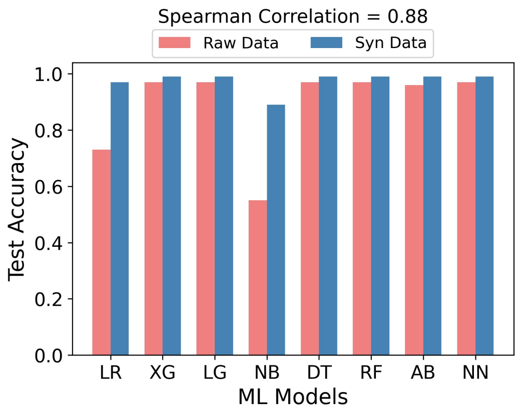
$$\text{Heavy Hit Error\%} = \frac{\text{GT Count} \cap \text{CMS Count}}{\text{GT Count} \cup \text{CMS Count}}$$

Figure 3 shows the heavy-hitter error plotted against different width-scales, where a width-scale indicates the percentage of memory allocated for CMS per row (0.1 means 10% of total bins in the ground truth). The trend among the real and the synthetic data remains consistent. We observe that the error rate decreases with the same pattern for both, raw and synthetic datasets, and their error rates at any sample are also very close. The difference in the error rate between the real and the synthetic data across different width-scales is negligible. This corroborates that the data generated by NetShare [9] can perform at par with the original data on Telemetry applications.

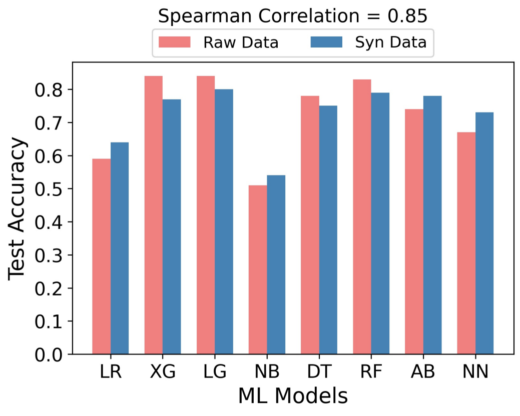
NetShare-generated [9] traces perform with high-accuracy on Anomaly Detection Tasks. The purpose of this experiment is to validate whether NetShare [9] generated traces can successfully substitute original traces for anomaly detection tasks. We use the FlowLens [2] dataset which is a collection of bengin P2P and malicious botnet traces. This dataset has been used before for botnet detection by collecting flow statistics from peer-to-peer networks. Packet size distribution (histogram) within a flow is used for classification since it is known that malicious P2P traces exhibit long connections with particular patterns of packet sizes.

We randomly pick a single packet header trace from bengin P2P collection of FlowLens [2] and one malicious trace from its botnet collection. We train NetShare [9] to synthetically generate a malicious botnet trace (using the FlowLens [2] trace) as its substitute. Then, we train several machine learning algorithms (including Logistic Regression, XGB Classifier, LGBM Classifier, Gaussian Naive Bayes, Decision Tree, Random Forest, AdaBoost, MLP Classifier models) by first taking the two original traces from FlowLens [2]. Then, we again train the same set of ML algorithms by using the bengin trace from FlowLens [2], but the malicious trace generated using NetShare [9]. For both these setups, we keep the benign trace the same (the raw FlowLens [2] trace) but we vary the malicious trace between original and synthetic.

Figure 4a) shows the accuracies of both synthetic and raw traces on these algorithms and their relative standings against each other. We can see that all models that perform with high accuracy on the original trace, also perform with high accuracy on the synthetic trace. This indicates that for botnet detection application on a packet header trace, we



(a) Botnet detection for P2P flows



(b) Anomaly detection on UGR16 flows

Figure 4: Comparison of various ML algorithms on botnet and anomaly detection tasks using UGR16 (NetFlow) and FlowLens [2] (PCAP) datasets using raw and NetShare-generated [9] traces. Performance rank-order is maintained between raw and synthetic traces.

can substitute an original trace by its NetShare [9] generated variant.

Next, we setup a second experiment using the UGR16 [7] NetFlow dataset and use its *TYPE* field for classification between blacklist and background type packets using individual packet header fields. We train NetShare [9] to synthetically generate a UGR16 [7] NetFlow trace as its substitute. Next, we take both, the bengin and malicious samples first, from the raw dataset and then from the generated dataset. We again train the same set of ML algorithms given above, first using the raw trace, and then using the synthetic one. Figure 4b) shows the accuracies of both synthetic and raw traces on these algorithms and their relative standings against each other. We can see that all models that perform with high accuracy on the original trace, also perform with high accuracy on the synthetic trace. This indicates that for anomaly detection application on a flow trace, we can substitute an original trace by its NetShare [9] generated variant.

NetShare [9] preserves the relative rank-order of Anomaly Detection algorithm performance In figure Figure 4a) and Figure 4b), notice that not only do the ML models perform with high accuracy on both raw and synthetic traces, their relative rank-order is also preserved with high-correlation between using the original and the NetShare [9] generated malicious traces against the benign ones. This is measured using the Spearman correlation coefficient, which is 0.88 out of 1.00 and 0.85 out of 1.00 for botnet detection and anomaly detection applications, respectively. This indicates that for such anomaly detection applications, we can substitute an original trace by its NetShare-generated [9] variant without changing the relative algorithmic performance.

5 LIMITATIONS

- NetShare [9] traces fail to capture the fine-grained inter-arrival times of raw traces. This indicates that for elaborate testing of new hardware/software systems which rely on such fine-grained properties of traces, NetShare [9] is still not a viable alternative.
- The other major issue is the scalability of the project for generating these traces. The original testbed setup by the authors uses a cluster of 10 machines (200 CPUs in total with 200GB of memory) and hundreds of CPU training hours to generate traces, which is not viable with smaller test setups.

6 CHALLENGES

- We required a beefy test setup to run NetShare [9] owing to its compute and memory requirements. We used $300 worth of Google Cloud credits to setup a machine with 40 CPUs and 230GBs of memory to train NetShare [9] models over several hundreds of CPU hours.
- We extended NetShare by adding the FlowLens [2] botnet dataset and generating the malicious traces using our testbed. Unfortunately, generating CAIDA [3] and UGR16 [7] on our setup was extremely difficult due to the compute requirements mentioned in (§5). For this reason, we contacted the original authors to receive those synthetic traces generated from NetShare [9] trained on a compute cluster of 200 CPUs, which we used for performing our evaluations for fidelity and downstream applications.

7 ACKNOWLEDGEMENTS

We would like to thank the author (Yucheng Yin) for providing synthetic traces for CAIDA [3] and UGR [7] generated from their compute cluster of 200 CPUs.

REFERENCES

[1] Martin Abadi, Andy Chu, Ian Goodfellow, H Brendan McMahan, Ilya Mironov, Kunal Talwar, and Li Zhang. 2016. Deep learning with differential privacy. In *Proceedings of the 2016 ACM SIGSAC conference on computer and communications security*. 308–318.

[2] Diogo Barradas, Nuno Santos, Luıs Rodrigues, Salvatore Signorello, Fernando MV Ramos, and André Madeira. 2021. FlowLens: Enabling Efficient Flow Classification for ML-based Network Security Applications. In *NDSS*.

[3] CAIDA. last accessed: 11/30/2022. The CAIDA UCSD anonymized internet traces. https://www.caida.org/catalog/datasets/passive_dataset/.

[4] Akanksha Cheeti, Annus Zulfiqar, Ashwin Nambiar, Hasan Amin, Murayyiam Parvez, and Syed Abubaker. last accessed: 12/09/2022. Reproducing Practical GAN-based Synthetic IP Header Trace Generation using NetShare. https://annuszulfiqar2021.github.io/NetShare/.

[5] Graham Cormode and S. Muthukrishnan. 2005. An improved data stream summary: the count-min sketch and its applications. *Journal of Algorithms* (2005).

[6] Zinan Lin, Alankar Jain, Chen Wang, Giulia Fanti, and Vyas Sekar. 2020. Using GANs for Sharing Networked Time Series Data: Challenges, Initial Promise, and Open Questions. In *Proceedings of the ACM Internet Measurement Conference*.

[7] Gabriel Maciá-Fernández, José Camacho, Roberto Magán-Carrión, Pedro García-Teodoro, and Roberto Therón. 2017. UGR'16: A new dataset for the evaluation of cyclostationarity-based network IDSs. (2017).

[8] Zachary Voase. last accessed: 12/09/2022. csiphash 0.0.5. https://pypi.org/project/csiphash/.

[9] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. 2022. Practical GAN-Based Synthetic IP Header Trace Generation Using NetShare. In *ACM SIGCOMM*.

[10] Jinsung Yoon, Daniel Jarrett, and Mihaela Van der Schaar. 2019. Time-series generative adversarial networks. *Advances in neural information processing systems* 32 (2019).

8 CONTRIBUTIONS

(1) Akanksha Cheeti
- Initially setup NetShare [9] locally on her machine
- Assisted with analyzing the NetShare pipeline design
- Assisted with fidelity evaluation setup for the traces
- Contributed to the paper's results breakdown
- Assisted with introduction and design sections in the report

(2) Annus Zulfiqar
- Assisted with Project Proposal
- Patched NetShare [9] to work with a single node (without a cluster)
- Extended NetShare [9] with Botnet detection application using FlowLens [2] dataset
- Generated synthetic traces using UGR16 [7] and FlowLens [2] datasets and evaluated them for anomaly and botnet detection applications
- Assisted with evaluation, limitations and challenges sections in the project report
- Project presentation slides and video recording

(3) Ashwin Nambiar
- Setting VMs and other infrastructure on GCP for the project
- Assisted with figures for presentation and report
- Assisted with implementation of the NetShare [9] pipeline on our testbed
- Assisted with writing the implementation, limitation and challenges section in report
- Created the blog post for the project

(4) Syed Hasan Amin
- Examined NetShare [9] design details with particular focus on deep learning architecture used from earlier works
- Assisted in writing design section of report
- Assisted in preparing project presentation
- Implemented predictive modeling pipeline for running anomaly detection task and evaluation of both real and synthetic traces
- Implemented data simulation scheme as exploratory study on examine fidelity benefits of NetShare over simple sampling methods

(5) Murayyiam Parvez
- Assisted with Project Proposal
- Evaluated fidelity between real and synthetic PCAP and NetFlow traces
- Evaluated Count-Min Sketch [5] for CAIDA [3] traces
- Created plots for fidelity evaluation
- Assisted with writing the evaluation section

(6) Syed Muhammed Abubaker
- Assisted in initial project development
- Analyzing the NetShare [9] design approach
- Assisted with evaluation metrics
- Assisted with project implementation