Improved Packet-Level Synthetic Network Traffic Generation

Improved Packet-Level Synthetic Network Traffic Generation

1st Jacob Soper
Queensland University of Technology
Brisbane, Australia
jacob.soper@hdr.qut.edu.au

2nd Yue Xu
Queensland University of Technology
Brisbane, Australia
yue.xu@qut.edu.au

3rd Ernest Foo
Griffith University
Gold Coast, Australia
e.foo@griffith.edu.au

4th Zahra Jadidi
Griffith University
Gold Coast, Australia
z.jadidi@griffith.edu.au

5th Kien Nguyen Thanh
Queensland University of Technology
Brisbane, Australia
k.nguyenthanh@qut.edu.au

***Abstract*—While using generative models to create synthetic network traffic is faster and cheaper than traditional testbeds, synthetic traffic suffers from problems with realism and structural completeness. State of the art traffic generation frameworks usually omit payloads because of the difficulties in representing their high-dimensional data, which makes the synthetic traffic unrealistic and limits its usefulness. This work proposes a two-stage process that takes advantage of the high repetition of some protocols, particularly those used by Industrial Control Systems, to selectively simplify payloads, greatly reducing the number of classes and reducing model loss and consequently the ability of the model to handle sequences of payloads. Model training loss was reduced by 47.796%, and payload class selection was improved up to 69% over state of the art approaches, allowing for more realistic synthetic network traffic with reduced memory and computation overheads.**

***Index Terms*—Industrial control systems, network traffic generation, time-series generation**

I. Introduction

There is a demand for diverse, relevant, and up to date network traffic datasets for research purposes and the training of intrusion detection systems [9], and there are several methods for generating or augmenting these datasets. The first is the use of testbeds involving real hardware [7], the second is the use of simulators that feature software replacements for hardware [16], and the third is the use of machine learning generative models to synthetically generate new traffic by learning the behavior of real devices transmitting traffic over the network. [1] In the past few years, synthetic traffic generation has become increasingly sophisticated but still cannot fully replace testbeds because the synthetic data is not realistic enough. This challenge and opportunity is especially relevant for systems such as Industrial Control Systems where the real equipment is very expensive.

Payloads present a significant problem for network traffic generation as they are high dimensional data that is in many cases at least partially encrypted, thus indistinguishable from noise. State of the art approaches such as state-space generation models [3], that tokenize the whole packet struggle with the problem of payloads taking up a bulk of each packet sample, increasing model size and complexity. The authors of NetShare, a GAN-based traffic generation model, state that "NetShare does not currently generate payloads, which are much higher-dimensional than the headers generated in this work. We expect that realistic payload generation would be challenging, and require different techniques." [20]

This work is motivated by a desire to increase the realism and usefulness of synthetically generated or augmented datasets by improving the ability of models to select an appropriate payload class or token.

There are several approaches to dealing with payloads.

1) Record length of the payload, and then map back to first payload from the original set with this length when reconstructing the final output. Storing payload length or even the length of the entire packet as an attribute is seen in frameworks such as NeCSTGen [14] and NetDiffusion [9]. This has problems, however. The ICMP (ping) protocol has request and response payloads of identical length. Length is good for breaking the traffic down into broad classes, but the subtler subclasses are lost.
2) Hash the payloads and use unique hashes as classes. PCAPGAN [5] hashes each layer of the packet, including payloads, and stores them in a dictionary of classes. But this has problems in practice due to the extremely high number of classes that result. For example, given 12,736 ICMP packets there will be 6838 payload hashes, as seen in Table III.
3) Tokenize the payload and train a model such as a transformer or state-space model to attempt to learn to reproduce them by trying to predict which byte of the payload comes next. This approach has its own set of issues because high dimensionality combined with these architectures being prone to hallucination results in semantically incorrect payloads. [9]

To illustrate the problem, the true number of classes for ICMP is typically two: one response, one request. Incrementing counters and checksums in the ICMP protocol create

numerous false variations. As explored in Section IV, when the NeCSTGen traffic generation model was trained on hashes, the model was only able to select the correct high level class of payload (request or reply) 49% of the time for ICMP and 31% of the time for DNP3.

The goal of this work is to find a compromise between not capturing enough classes (measuring length) and capturing too many classes (measuring unique hashes). Shannon Entropy has previously been used in the network traffic context to detect whether data is encrypted or not [4], and this idea was adapted to this problem. Instead of measuring the entropy of the payload itself, the change from one payload to the next is measured. Consider Table I for a visual illustration of packet data. Each row represents a payload, and each column represents values at that column index changing from one payload to the next. If there is a column that belongs to a random string such as a checksum, this will have very high entropy, while a single repeating character has low entropy. By measuring the Shannon Entropy of a string and dividing by the log of 15, the value is normalized for hex data in the range 0-F. If there are six payloads and the values down each column are "080808", "bbaa99", and "bc24ef", these will have normalized entropy values of 0.255, 0.405, and 0.661 respectively. If there are 15 payloads and a particular column's values are '1234567890ABCDEF', then this has an entropy value of 1.023832099239262, which represents constant change over the full domain. As the allowed entropy threshold is increased, more unique classes will exist. Figure I shows a hexadecimal matrix where each row is a payload. The highlighted sections of the figure also show that protocols such as DNP3 and ICMP contain sequence counters that constantly increment from one payload to the next.

```
05640bc40c0001002ec5cfce013c0206eb3b
05640bc40c0001002ec5d0cf013c0206ce24
05640bc40c0001002ec5d1c0013c0206722d
05640bc40c0001002ec5d2c1013c0206249d

08004cab000100b061626364656667686a6
08004caa000100b161626364656667686a6
08004ca9000100b261626364656667686a6
08004ca8000100b361626364656667686a6
```

Fig. 1. DNP3 (Top) & ICMP (Bottom) payloads, showing repetition and the presence of high entropy sections.

Sequence counters vary in number, position, and presence from protocol to protocol, but if generically identified, the sequence counters could be zeroed out regardless of entropy, and stored in a lookup table to aid in sequence reconstruction. As seen in Figure 2 the reference traffic generation model is augmented at the data pre/post-processing stages. The training data is simplified and then reconstructed.

The key contributions of this work are:

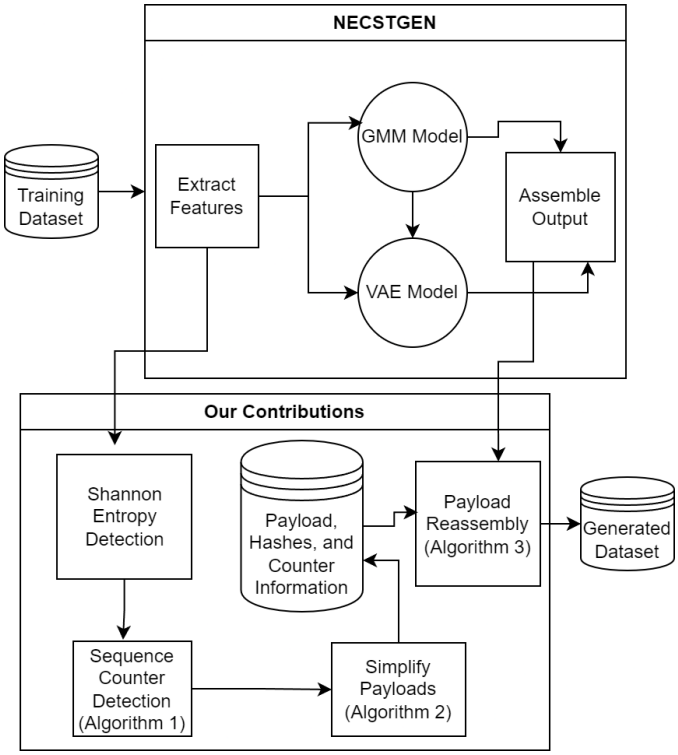

Fig. 2. The contribution augments both the data used for training the model and the data generated by the trained model.

- Algorithm for using Shannon Entropy to selectively simplify payload data.
- Algorithm for identifying sequence counter values in payloads.
- Algorithm for converting generated payload classes back into appropriate payload from training data.

The rest of the paper is structured as follows. In Section II, more detail is provided on generative model synthetic network traffic generation and its challenges. In Section III the paper's proposals for reducing payload dimensionality are discussed. In Section IV the contributions of the paper are evaluated by testing the impact on training loss and synthetic traffic fidelity compared to ground truth by state of the art traffic generation networks such as NeCSTGen. Section V explores how this work might be extended in future, or ways in which it might inspire parallel research.

II. Background

A. Overview

Researchers have been attempting to use machine learning to replace traditional computer network testbeds for some time, particularly those involving expensive physical equipment such as Industrial Control Systems. Approaches range from semi-physical testbeds where Sensors can be simulated using Matlab & Simulink, and paired with real Programmable Logic Controllers. [7] Instead of a real gas pipeline, a model can be created which simulates the physical properties of a real pipeline, and feeds simulated data to PLCs. The

fully synthetic approach replaces all physical components with software simulations, and the network topology itself is simulated. SCADASim [16] (2011) is an example where the programmable logic controllers, sensors, and network topology are all simulated.

TABLE I
TRAFFIC GENERATOR TYPES

Type	Details
Throughput	Generate specific volume of network traffic.
Application-level	Generate traffic for a specific application, often by using libraries from that application.
Trace replay	Take an existing stream of network traffic and play it back at the original speed or at a different speed. Usually the payloads are empty.
Model-based	Model trained on traffic represented as features.
Trace-driven	Model fed real traffic traces.
Script-driven	More complex behavior, but still manually scripted by the researchers.

A 2022 survey identified 92 distinct network traffic generation projects [1], divided into the categories listed in Table I. These approaches involve agents communicating over a network where the internal logic of these agents is driven by scripts or physical behaviors. Training a generative model to mimic the network traffic patterns of devices on the network is much faster and cheaper, but this approach comes with its own challenges. Table II explores the differences between state of the art traffic generation models. Do they understand network traffic as a sequence? Do they extract packet header features? How are payloads implemented? Does this project have source code available?

TABLE II
GENERATIVE MODEL COMPARISON

Name	Level	Seq	Header	Payload	Source
PAC-GAN [2]	Packet	No	Naive	Naive	Yes
PCAPGAN [5]	Packet	Yes	Hash	Hash	No
NetGPT [13]	Packet	Yes	Features	No	No
NeCSTGen [14]	Packet	Yes	Features	Features	Yes
NetDiffusion	Packet	Yes	Yes	No	Yes
SPATGAN [19]	Flow	N/A	Yes	No	No
PAC-GPT [10]	Packet	Yes	Yes	Partial	Yes

Naive GAN models such as PAC-GAN [2], SIP-GAN [12], and MirageNET [15] are inspired by GAN work on handwritten number classification/generation. The authors of this work made a contribution to reducing the error rate of payloads generated by these models [18], but while simple GAN models implicitly include payloads, they do not have any concept of traffic as a sequence, which limits their usefulness.

Frameworks such as NetGPT [13], PCAPGAN [5], NetDiffusion [9], and NeCSTGen [14] represents network traffic as a multivariate time-series problem using explicit features. Diffuse-based models such as the stable diffusion-based NetDiffusion [9] and the large language model-based PAC-GPT [10] suffer from the same payload dimensionality problems as their GAN counterparts. Additionally the risk of hallucinations made them less appealing for evaluation purposes. Models such as NetDiffusion [9] and the State Space model that is unnamed at the time of writing [3] require post-correction of traffic.

B. Reference Models NeCSTGen & PCAPGAN

NeCSTGen [14] was chosen as the reference model because the framework is capable of generating traffic sequences, but not prone to hallucinations like the transformer and state space architectures, aiding in evaluation. While NeCSTGen had not yet implemented packet reconstruction, prelim work had been outlined by the authors, thus the model's outputs were able to be easily adapted to construct packets using the Scapy framework in Python. Thus each of the model's output features were mapped to a newly constructed packet field.

NeCSTGen consists of a Gaussian Mixture Model (GMM) and a Variational Auto-Encoder (VAE) which transforms the sequence of packets into a sequence of classes which are compressed/decompressed by the VAE and converted into packet data.

NeCSTGen's authors had experimented with trying to model payloads by mapping individual hex characters to LSTM features, similar to tokenization in transformer models, but this approach was abandoned due to poor accuracy and high memory usage. Storing payload length proved limiting because it is common for different payload classes to have the same length. The decision was made to pair NeCSTGen's architecture with the hashing approach outlined by PCAPGAN [5] which had sliced packets into layers and hashed each layer creating a dictionary. This idea was combined with NeCSTGen so packet headers are represented as features while the payloads are hashed. This forms the state of the art baseline for incorporation payloads into time-series generative models.

C. Research Problem

Generation of flows of network packets is a time series generation problem where payload representation is challenging due to payloads being very high dimensional data and not easily represented. Table III shows the four datasets used for this paper. The protocol, number of packets, number of classes, and number of unique hashes are listed. "True Classes" are determined via manual analysis in Wireshark. For example, if a Modbus device returns two different temperature values, this is considered two distinct classes.

TABLE III
PAYLOAD PROTOCOL CLASSES

Protocol	Packets	True Classes	Unique Lengths	Unique Hashes
AVTP	8000	1	1	8000
DNP3	15088	4	4	2612
ICMP	12736	2	1	6838
Modbus	12000	5	4	5

State of the art approaches as listed in Table II have been unable to solve the dimensionality problem of network traffic sequences without omitting payloads or suffering from poor

accuracy. As seen in Table III the ICMP payload falsely appears to have 6,838 classes, when there are only two. The challenge is finding a way to incorporate payloads while reducing the negative impact on the model's accuracy.

III. Our Proposal

Payloads are stored in a matrix, then columns with normalized Shannon Entropy values over a threshold are filled with zeroes. Sequence counters are also stored and their columns zeroed. These zeroed out, simplified payloads are then hashed and used to train the model. Eliminating high entropy columns produces a subset of classes closer to the true number of classes in the training data, improving model performance over existing methodologies. The synthetic packets generated by the model then have an appropriate payload attached using the stored information from the training set.

A. Data Preparation

Sequence counters often increment independently and protocols are often asymmetric, so traffic is analysed one direction at a time. Headers are dissected into features and payloads are stored in a hexadecimal matrix. Shannon Entropy, minimum hex, and maximum hex values for each column are stored in three vectors for later use. In Algorithm 1 normalized Shannon Entropy is calculated from the payloads. As payloads can have varying length, they are grouped by length, the entropy is calculated for each group, then a mean is taken for each column across the sets. As mentioned in the introduction the Shannon Entropy is divided by $\log_2(15)$ to ensure that the upper boundary is 1 if all characters in the 0-F hex range are used randomly.

Algorithm 1: Calculate Normalized Shannon Entropy

```
Inputs: - List of payload strings in hexadecimal format: PLDS
Outputs: + Array of normalized Shannon Entropy values: final_entropy
Initialize payload_lengths as array of unique string lengths from PLDS;
Initialize final_entropy as empty list;
foreach length in payload_lengths do
    Initialize temp_entropy as array of length payload_lengths;
    foreach column in PLDS do
        Initialize temp_column as empty list;
        Append column values to temp_column, skipping rows where
          length is not equal to payload's length;
        Initialize frequencies as counts of each unique character in
          temp_column;
        Initialize entropy := 0;
        foreach f in frequencies do
            entropy := entropy + f × log2(f);
        end
        entropy := -entropy ÷ log2(15);
        Append entropy to temp_entropy;
    end
    Append temp_entropy to final_entropy;
end
Compute the mean of each column's values in final_entropy and assign the
  resulting array to final_entropy;
return final_entropy;
```

Algorithm 2 takes a matrix of payloads and vectors of maximum and Shannon entropy values for each column of the matrix. It iterates over payloads searching for incrementing indices that indicate sequence counters up to 4 characters in length. C, N, and Y represents "counting change", "no change", and "yes change". A column where C is higher than Y increments more than changes arbitrarily. The max value of each column is used as modulus. If a column with a max value of A changes from A to 0, this is detected as an increase of C, not Y. It returns a dictionary where keys are counter indices, and values are counter they belong to.

Algorithm 2: Sequence Counter Detection

```
Inputs: - Matrix where each row is a payload in hexadecimal format: PLDS
        - Vector of max values for each column of PLDS: max_values
        - Vector of entropy for each column of PLDS: shannon
Outputs: + dictionary of sequence counters and associated indices: positions;
Initialize counter_length, discovered_counters := 0;
Initialize dictionary positions;
foreach column in PLDS iterating reversed do
    if shannon[column]= 0 then
        continue;
    Initialize Y, N, C := 0;
    foreach row in PLDS at current column do
        if PLDS[row, column]= PLDS[row - 1, column] then
            N += 1;
        else if PLDS[row, column]= PLDS[row - 1, column]modulus
          max_values[column] then
            C += 1;
        else
            Y += 1;
        end
    end
    if C >Y and C >N and counter_length = 0 then
        discovered_counters += 1;
        positions[column]:= discovered_counters;
        counter_length += 1;
    else if counter_length = 1 then
        if C >Y then
            positions[column]:= discovered_counters;
            counter_length += 1;
        else
            counter_length := 0;
    else if counter_length = 2 then
        if C >Y then
            if C divided by number of columns in PLDS>0.01 then
                discovered_counters += 1;
            positions[column]:= discovered_counters;
            counter_length += 1;
        else
            counter_length := 0;
    else if counter_length = 3 then
        positions[column]:= discovered;
        counter_length := 0;
end
return positions
```

Algorithm 3: Simplify Payloads

```
Inputs: - matrix where each row is a payload stored in hexadecimal format:
          PLDS
        - Vector of shannon entropy data: shannon
        - Vector of identified sequence counter indices: indices - Shannon
          threshold: threshold
Outputs: + dictionary of results: results
Initialize dictionary results;
foreach payload in PLDS do
    original := payload;
    for i:= 0 to length of payload do
        if i in indices then
            payload := [0];
        else if shannon[i]>threshold then
            payload := [0];
    end
    Add {payload : payload, original} to results
end
return results
```

Algorithm 3 uses Shannon entropy and sequence counter information to simplify the payloads before they're rehashed.

If a payload index position is part of a sequence counter or over the Shannon Entropy threshold, that index's value is replaced with zero. The payloads are then rehashed and converted to classes, then the model is then trained on this simplified class set.

B. Post-Generation Phase

The post-training phase attempts to build a valid packet using the output of the model. The model generates values representing classes for source address, destination, Time To Live, Time Difference, and the payload class. The challenge is then mapping the payload class back to a plausible payload from the training set. The simplified hash has to be mapped back to an original hash, and if appropriate sequence counters are used to aid in the selection. Algorithm 4 details the reassembly.

Algorithm 4: Payload Reassembly

```
Inputs: - vector of generated packets with fields SRC and PAYLOAD: Packets
        - lookup Table with columns CLASS, OGPAYLOAD,
        SIMP_PAYLOAD, SIMP_PAYLOAD_HASH: Table
        - initial source counter: s_counter
        - initial destination counter: d_counter
Outputs: + updated packet data : packets;
Initialize current counter: c_counter as 0;
Initialize src_id := packet.SRC from first packet in packets;
foreach packet in packets do
    if s_counter = null and d_counter = null then
        packet.PAYLOAD := First Table.OGPAYLOAD where
          packet.SIMP_PAYLOAD_HASH =
          Table.SIMP_PAYLOAD_HASH
    else
        if packet.SRC = src_id then
            c_counter = s_counter;
            s_counter += 1;
        else
            c_current = d_counter;
            d_counter += 1;
        Initialize candidate := Table.OGPAYLOAD where Table.COUNTER
          = c_counter and Table.SIMP_PAYLOAD_HASH =
          packet.SIMP_PAYLOAD_HASH;
        if candidate = null then
            for i := c_counter to max(Table.COUNTER) do
                Retry using Table.COUNTER = i;
                If match found, break;
            end
            candidate := First Table.OGPAYLOAD where
              packet.SIMP_PAYLOAD_HASH =
              Table.SIMP_PAYLOAD_HASH
        packet.PAYLOAD := candidate;
end
return packets
```

If no source/destination counters are present, the first matching payload is used. Otherwise the closest matching sequence counter in the lookup table is used. Higher counter values from the sequence are tried and if no match is found then the first matching payload is tried as a fallback. The goal is to choose the most appropriate substitution if an exact match isn't found. Figure 3 visually illustrates the process.

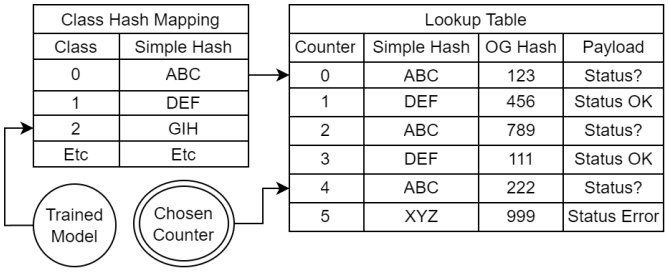

Fig. 3. Payload Class Reconstruction.

The model's payload class maps to the hash of a simplified payload. A sequence counter loop, for instance from 0 to 255, represents a re-usable pattern. The more source traffic used for training, the lower the chance of a combination of counter and payload not existing in the lookup table. If there is no combo that matches a given counter value, the nearest matching entry in the lookup table is substituted.

IV. Evaluation

The work is evaluated by measuring the reduction of training loss and closeness of distribution between the training set and the generated traffic, as well as the closeness of regenerated sequence counters to originals.

The problem being evaluated is the inability of the model to include payloads without suffering training loss so high it may as well be randomly guessing which payload comes next. In these experiments, the chosen Shannon Entropy Threshold was 0.25 based on iterative experimentation, but it can be adjusted to any setting between 0 and 1, and could be automatically adjusted to select a maximum number of payloads instead of a specific threshold.

A. Data Sets

Three datasets were sourced, and one was created for this paper, displayed in Table IV. By testing multiple protocols, the goal was to show that this work can generalize.

TABLE IV
Datasets Used

Protocol	Packets	Counters	Source
AVTP	8000	4	Derived from set [8].
DNP3	15088	2	Derived from set [17].
ICMP	12736	1	Created by pinging Cloudflare DNS
Modbus	12000	None	Derived from set [6].

B. Sequence Counter Detection

Algorithm 2 needs to identify sequence counters in the payloads, and not have false positives. Table V shows the positional indices of the sequence counters found through manual protocol inspection, the ones found by the general algorithm, and the percentage that were correct.

TABLE V
DETECTION RESULTS

Protocol	Manual	Algo	Correctly Detected
AVTP	4-5, 54, 80, 464,	4-5, 54	50%
DNP3	20-21, 23	20-21, 23	100%
ICMP	12-15	12-15	100%
Modbus	None	None	100%

C. VAE Loss Reduction

By using Shannon entropy information from Algorithm 1 to process the payloads using Algorithm 3, the Variational Auto-Encoder Loss is reduced, improving the payload class selection. Table VI shows an average Variational Auto-Encoder training loss decrease of 47.796%. The Modbus Protocol has no sequence counters, so the minor difference in training loss is a result of non-deterministic training.

TABLE VI
TRAINING LOSS AFTER 100 EPOCHS

Protocol	Before	After	Reduction
ICMP	21.781	3.207	85.28%
Modbus	42.518	43.750	N/A
DNP3	74.312	62.38	16.06%
AVTP	31.338	18.159	42.05%

D. Payload Class Detection

By using Algorithm 3 the goal is to ensure the simplified payload set resembles the true class set. Table VII compares the number of classes in the original dataset to the number of classes identified by naive hashing, and then the number of classes identified by hashing the simplified payloads.

TABLE VII
NUMBER OF PAYLOAD CLASSES DETECTED

Protocol	Classes	Standard Hash	Improved Hash
ICMP	2	6838	2
Modbus	5	5	5
DNP3	4	2612	4
AVTP	1	115846	14

Three of the protocols had the correct number of payload classes identified. AVTP is more ambiguous because it has protocol flags columns that fall below the Shannon entropy threshold. However, a reduction from 11,584 to 14 classes still improves training loss.

E. Traffic Class Ordering

With network traffic, a protocol has an expected pattern of back and forth communication that can be validated, such as a request packet followed by a response packet. Requests will come from the client and replies from the server. When a model is trained on hashes, this relationship is implicit. By reducing the number of hashes, thus the number of classes, the model is better able to select an appropriate payload. Table VIII measures the percentage of packets that were correctly ordered based on source/destination and payload class. To evaluate the training data and generated samples were compared. A score of 0 would mean that every sample was wrong. A score of 50 would be akin to random guessing of a binary class. We see a significant improvement for the DNP3 and ICMP protocols. This demonstrates that the work does yield an improvement for this style of traffic generation with payloads included. Modbus lacks sequence counters and checksums, so sees no change, and AVTP is excluded because the traffic isn't bidirectional.

TABLE VIII
BIDIRECTIONAL CLASS SCORING

Protocol	Standard Hash	Improved Hash	Improvement
DNP3	31%	100%	69%
ICMP	49%	100%	51%
Modbus	100%	100%	0%

The model is no longer being overwhelmed by a huge volume of unique payload hashes and is now generating a more appropriate sequence.

F. Sequence Evaluation

The sequence counters are added back into the data based on the original sequence information, as seen in Algorithm 4. But this needs to be validated in some way. The Kolmogorov Smirnov test [11] determines whether the sequence counter values belong to the same distribution. A value of 0.0 means no match. 1.0 means a perfect match. Using the standard hashing methods, DNP3, ICMP, and AVTP achieved 0.0. By using stored sequence counter values to select the most appropriate payload from the sequence of possible payloads, they achieved 0.19, 1.0, and 1.0 respectively. The reason DNP3 was unable to attain the same level of accuracy is that at rare intervals the DNP3 protocol sends a "confirm" packet that disrupts the usually linear sequence, and is difficult for the model to capture. However, a K&S value of 0.19 still indicates correlation. A more sophisticated sequence counter tracking system might be able to fit the distribution better.

G. Sequence Splicing

An additional form of validation is to look at the visible sequence pattern of augmented data with or without sequence counter awareness. Because sequence counters loop, it presents an opportunity for more realistic splicing of synthetic traffic onto real traffic.

If a protocol sequence counter cycles from 0 to 255 and then back to 0, the spliced traffic might jump backwards from 100 to some arbitrary value, which is unrealistic. By taking the sequence counter of the most recent sample in the dataset and using it with Algorithm 4, the sequence can be continued more seamlessly, making the synthetic nature of the traffic less obvious. Figure 4 shows how this helps the synthetic sequence better match the real traffic it has been spliced onto.

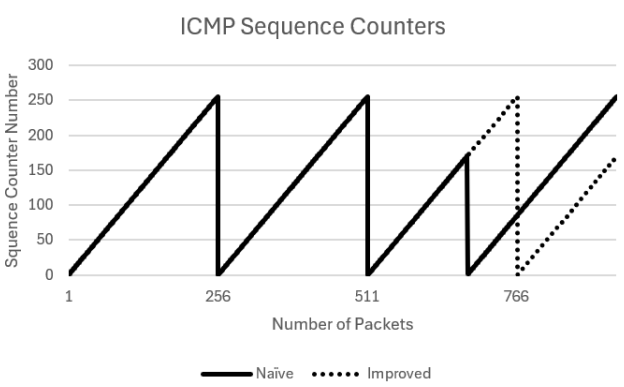

Fig. 4. Spliced from start of traffic vs nearest sequence position.

V. Discussion & Future Work

This paper has introduced novel methodologies that improve the realism of synthetic payload re-assembly for highly repetitive protocols by simplifying the payload classes and using information about the original data for reconstruction. By using four different protocols (ICMP, Modbus, DNP3, and AVTP), it was demonstrated that the techniques proposed in this paper can provide benefits to different protocols and can generalise to some degree.

Some protocols use precision time-stamps, and these are more challenging to reproduce, but it may be possible to generically detect them as well. It may also be possible to use multiple models on a per-protocol basis. Currently the Shannon entropy removal eliminates data such as JPEG image data in the AVTP protocol, but this could be replaced with data from another model. Also, instead of detecting sequence counters using an algorithm it might be possible to teach models to identify sequence patterns.

The hope is that this work can be extended and refined, and inspire new solutions to the problem of generating synthetic network traffic that is structurally complete, containing both headers and payloads. While this work was primarily built for NeCSTGen, it remains relevant for any other time-series based synthetic network traffic generation framework. With transformer and state-space models, the trade-off shifts because while these models can remember large amounts of data, memory usage and performance are great concerns, particularly for cost reasons. Being able to train on simplified representations, requiring less memory and computation to learn, that are then mapped back to the original data presents opportunities for all the major traffic generation architectures.

VI. Acknowledgment

The authors would like to thank Fabien Meslet-Millet for his assistance in understanding the reference code of the NeCSTGen project.

References

[1] Oluwamayowa Ade Adeleke, Nicholas Bastin, and Deniz Gurkan. Network traffic generation: A survey and methodology. *ACM Comput. Surv.*, 55(2), jan 2022.

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

[3] Andrew Chu, Xi Jiang, Shinan Liu, Arjun Bhagoji, Francesco Bronzino, Paul Schmitt, and Nick Feamster. Feasibility of state space models for network traffic generation, 06 2024.

[4] Simon R. Davies and Richard Macfarlane. Comparison of common mathematical techniques used in the calculation of file entropy. In *2022 International Conference on Electrical, Computer, Communications and Mechatronics Engineering (ICECCME)*, pages 1–6, Nov 2022.

[5] Baik Dowoo, Yujin Jung, and Changhee Choi. Pcapgan: Packet capture file generator by style-based generative adversarial networks. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*, pages 1149–1154, 2019.

[6] Ivo Frazão, Pedro Henriques Abreu, Tiago Cruz, Hélder Araújo, and Paulo Simões. Denial of service attacks: Detecting the frailties of machine learning algorithms in the classification process. In Eric Luiijf, Inga Žutautaitė, and Bernhard M. Hämmerli, editors, *Critical Information Infrastructures Security*, pages 230–235, Cham, 2019. Springer International Publishing.

[7] Haihui Gao, Yong Peng, Kebin Jia, Zhonghua Dai, and Ting Wang. The design of ics testbed based on emulation, physical, and simulation (eps-ics testbed). In *2013 Ninth International Conference on Intelligent Information Hiding and Multimedia Signal Processing*, pages 420–423, Oct 2013.

[8] Seonghoon Jeong, Boosun Jeon, Boheung Chung, and Huy Kang Kim. Automotive ethernet intrusion dataset, 2021.

[9] Xi Jiang, Shinan Liu, Aaron Gember-Jacobson, Arjun Nitin Bhagoji, Paul Schmitt, Francesco Bronzino, and Nick Feamster. Netdiffusion: Network data augmentation through protocol-constrained traffic generation. *Proc. ACM Meas. Anal. Comput. Syst.*, 8(1), February 2024.

[10] Danial Khosh Kholgh and Panos Kostakos. Pac-gpt: A novel approach to generating synthetic network traffic with gpt-3. *IEEE Access*, 11:114936–114951, 2023.

[11] Frank J. Massey. The kolmogorov-smirnov test for goodness of fit. *Journal of the American Statistical Association*, 46(253):68–78, 1951.

[12] Amar Meddahi, Hassen Drira, and Ahmed Meddahi. Sip-gan: Generative adversarial networks for sip traffic generation. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*, pages 1–6, 2021.

[13] Xuying Meng, Chungang Lin, Yequan Wang, and Yujun Zhang. Netgpt: Generative pretrained transformer for network traffic, 2023.

[14] Fabien Meslet-Millet, Sandrine Mouysset, and Emmanuel Chaput. Necstgen: An approach for realistic network traffic generation using deep learning. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 3108–3113, 2022.

[15] Santosh Kumar Nukavarapu, Mohammed Ayyat, and Tamer Nadeem. Miragenet - towards a gan-based framework for synthetic network traffic generation. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 3089–3095, Dec 2022.

[16] Carlos Queiroz, Abdun Mahmood, and Zahir Tari. Scadasim—a framework for building scada simulations. *IEEE Transactions on Smart Grid*, 2(4):589–597, Dec 2011.

[17] Panagiotis Radoglou-Grammatikis, Vasiliki Kelli, Thomas Lagkas, Vasileios Argyriou, and Panagiotis Sarigiannidis. Dnp3 intrusion detection dataset, 2022.

[18] Jacob Soper, Yue Xu, Kien Nguyen, Ernest Foo, and Zahra Jadidi. A two-pass approach for minimising error in synthetically generated network traffic data sets. In *Proceedings of the 2023 Australasian Computer Science Week*, ACSW '23, page 18–27, New York, NY, USA, 2023. Association for Computing Machinery.

[19] Chao–Lun Wu, Yu–Ying Chen, Po–Yu Chou, and Chih–Yu Wang. Synthetic traffic generation with wasserstein generative adversarial networks. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 1503–1508, Dec 2022.

[20] Yucheng Yin, Zinan Lin, Minhao Jin, Giulia Fanti, and Vyas Sekar. Practical gan-based synthetic ip header trace generation using netshare. In *Proceedings of the ACM SIGCOMM 2022 Conference*, SIGCOMM '22, page 458–472, New York, NY, USA, 2022. Association for Computing Machinery.