A two-pass approach for minimising error in synthetically generated network traffic data sets

A two-pass approach for minimising error in synthetically generated network traffic data sets

Jacob Soper
Kien Nguyen
Yue Xu
soperj@qut.edu.au
k.nguyenthanh@qut.edu.au
yue.xu@qut.edu.au
Queensland University of Technology
Brisbane, Queensland, Australia

Ernest Foo
Zahra Jadidi
e.foo@griffith.edu.au
z.jadidi@griffith.edu.au
Griffith University
Southport, Queensland, Australia

ABSTRACT

Network security research requires network traffic data sets of sufficient size, variety, and completeness in order to perform tasks such as training intrusion detection systems. While the standard is to use testbeds to create data sets or capture data sets from real systems, Generative Adversarial Networks have proven successful in generating new packet samples for protocols such as ICMP, DNS, HTTP, and SIP. However, existing approaches have problems with quality evaluation due to insufficient sampling, or they require non-generalised criteria to be created specifically for the data set being trained on. This paper proposes a new and generalised two-pass approach to evaluating the quality of samples produced by the generator to produce a filtered, higher-quality output data set. Compared against SIP-GAN, which is a Generative Adversarial Network model targeting Session Initiation Protocol samples, we reduced the ratio of malformed SIP samples from between 9.6% and 19.8% down to 1.2%.

CCS CONCEPTS

• **Computing methodologies** → **Machine learning**; • **Networks**;

KEYWORDS

datasets, neural networks, generative adversarial networks, network traffic

1 INTRODUCTION

In order to protect computer networks against malicious behaviour, machine learning models, usually known as Intrusion Detection Systems, are employed to identify abnormal or attack-classified behaviour. The challenge is that training these models requires large amounts of data representative of real, diverse network behaviour. While many data sets do exist[10], they are usually very general in their protocols, focusing on very generalised network behaviours, failing to include Industrial Control Systems or other specialised protocols. Data sets for non-generic devices are much rarer, and this poses a challenge for researchers.

Another problem is that sometimes data sets are class or protocol imbalanced, or do not have enough samples. There are three potential solutions. One is to run real hardware to capture a data set[4], which can be time-consuming and expensive. Another is to write a simulator that mimics the behaviour of real devices on a network to generate a fully synthetic data set[3]. The third method, which is the focus of this research is to take an existing data set that does not meet our requirements, and then use machine learning[11] to generate new samples, augmenting the data set to meet our needs.

While Generative Adversarial Networks have been successful in taking summarised network traffic and generating new samples with similar distributions in existing research[5][11], a complete data set needs to include not just the summaries of network activities, but the network activities themselves, usually represented by packets stored in PCAP files. Therefore, the challenge is to use machine learning to generate network traffic by learning to build actual network packets instead of summaries.

Approaches exist to measure the quality of samples such as using protocol analysers[2], which can be accurate, but require very specific checks to be written. Another approach is to measure values in the packet associated with a particular protocol[7], or some specific header values.[1] Yet another approach is to transmit packets to a target device and test for valid responses. There are some problems with each of these approaches we would like to address:

(1) Validation of generated samples is protocol or data set specific. Whether selecting specific regions in the packet to test[1][7] or writing scripts for the protocol analyser,[2] methods are either general or accurate, but fall short of achieving both.
(2) The training methodology relies on training the GAN for several hundred epochs, halting when detected error reaches a desired level, a process that can require several hundred epochs and several hours.[7]
(3) Valid response testing requires target devices or simulators, which is impractical.

What we would like to accomplish:

(1) We would like to determine the quality of generated packets in a more generalised way.
(2) We would like to improve both the accuracy and time efficiency of the training and generation process by using more precise information about the error in the set.

This paper makes the following contributions.

(1) A new general method for evaluating sample quality during training based on the static data in the data set.
(2) A filtering approach that combines multiple error checks to produce a final set with a minimal error rate.

The structure of the paper is as follows:
In Section 2, we examine the current state of synthetic sample generation and error evaluation. In Section 3 we explain the design of our model, and how it incorporates elements from two existing approaches, to lay the technical foundation for our contributions. In Section 4 we provide a high level overview of our contributions, followed by Section 5 where we explain our approach in detail for deriving multiple error rates then combining them to filter out malformed samples from the data set. Section 6 is broken down into three experiments where we demonstrate that our combined methodology yields better results than the current standard approaches. This is followed by Section 7 which is Discussion and Future Work.

2 RELATED WORKS

The first major piece of research on using Generative Adversarial Networks to generate network packets in a naive way was PAC-GAN[1]. This used a CNN GAN which adapted the MNIST digit generation approach to generate network packets.

This was followed by SIP-GAN[7], based on PAC-GAN, but with a less complex neural network, removing some fully connected layers, and using the simpler cross-entropy instead of Wasserstein loss function. It introduced an early stopping mechanism using a quality factor calculated from mean error distance between the generated data and specific known SIP protocol values. Early stopping was used to halt training when the Generator was performing optimally, instead of training it for a fixed number of epochs.

MirageNet[8] is a similar model to PAC-GAN, but it processes data as a one dimensional vector instead of a two dimensional matrix.

An adjacent piece of research was the PCAPGAN[2] model that cut up network packets and stored them as a dictionary of layers then rearranged those layers using machine learning to form new packets. Unfortunately, PCAPGAN did not publish their data sets nor code, and they are only mentioned for the sake of completeness.

The SIP-GAN model was a simplification of PAC-GAN that omitted the additional fully connected layers, while also using a cross entropy loss function and a quality-based early stopping mechanism. It is unclear why SIP-GAN chose that model, but we speculate that it was because the PAC-GAN model was unable to process a Session Initiation Protocol packet due to high memory usage.

A particular packet's quality is determined by structural adherence to a networking protocol protocol, meaning that loss values are not enough, although WGAN loss does correspond more closely to quality than other loss functions. Programs such as Wireshark use Berkeley Packet Filters (BPF), which decode the packet layer by layer, and a researcher can determine if a packet is complaint or malformed by interpreting the results of the protocol analysis.

This doesn't generalise well because effectively using BPF filters require specific values to be selected, such as IP addresses or MAC addresses or the specific sub-class of packet. We could design a scripting system that determines what values a network sample should have based on characteristics of the source data set, but this would fail if it encountered a protocol that didn't have a BPF profile, and isn't generalised.

In PAC-GAN, a byte error value was determined by checking headers and seeing whether any bytes were incorrect. This determined whether error existed in the generated samples, but not the measurable error distance.

The SIP-GAN model checks a specific series of values in the generated sample (corresponding to the SIP request line), and subtracts these values from the known good values taken from an original data set sample. It then takes a mean of all the error to produce a total error distance over the set.

Thus, there are a few existing approaches to this problem, but their generalisation and accuracy are lacking.

3 PROPOSED MODEL

Our proposed model is called FPAC-GAN, or Filtered PAC-GAN. It is a combination of the 2019 PAC-GAN model[1] with the 2021 SIP-GAN model[7], incorporating the simpler generator and critic of SIP-GAN as well as the early stopping ideas. Our contribution is the addition of two new error measurement techniques and a filtering process combining both measurements.

The general structure of this model is broken down into two phases, shown in Figures 1 & 2. Figure 1 deals with the training process, where the source data set is taken, converted, processed using Algorithms 1 & 3, and the results are used to train our generator. The final output of the first phase is a trained generator and a dictionary of static positions, which is discussed in our contribution.

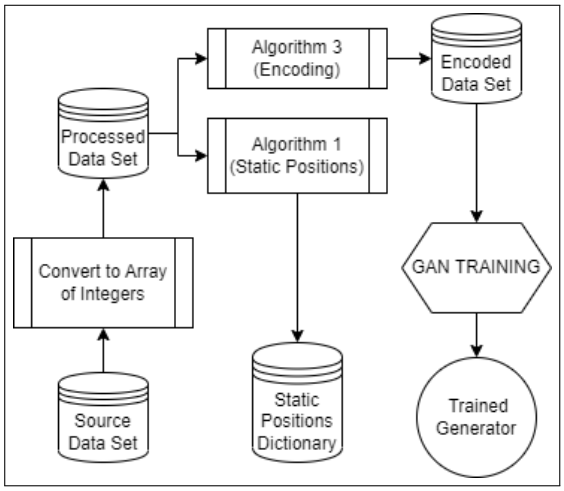

Figure 1: Phase 1

Phase 2, as seen in Figure 2, takes the generator and static position dictionary, then generates a batch of packets, and measures the error rates of the generated samples using Algorithms 2 & 5. It then uses this information to reduce the error ratio of the final set by using only samples that exist in both sets. Phase 2 is used both during training, where the error measurements are taken from a batch of samples produced each epoch, and used to determine whether the generator has improved from epoch to the next, and after training, when the generator is being used to produce output data sets.

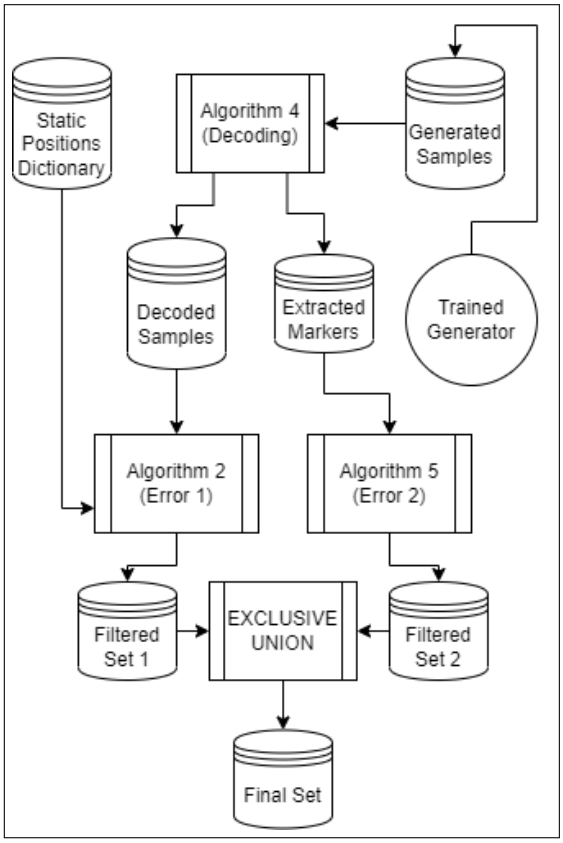

Figure 2: Phase 2

FPAC-GAN consists of a WGAN-WP Generator and Critic, with a Wasserstein loss function, training the Critic 5 times for every Generator training.

The generator consisted of an input noise vector of 64, into a fully connected layer of dimensions of 'matrix height * 7 * 256', so for a matrix of dimensions 28x20, this would be 35,840. This fully connected layer was then reshaped into a tensor of 'number of samples' by 256 by 'matrix height' by 7. This was then run through two deconvolution layers of kernel size 4, stride 2, and padding 1. The first deconvolution layer was from 256 to 64, and the second was from 64 to 32. Then a final convolutional layer was used to reduce the 32x32x'height'x28 tensor down to a 32x'height'x28 tensor for the final set of 32 network packet samples, with a final Tanh activation function.

The critic consisted of two convolutional layers followed by a fully connected layer. The number of channels were 1 to 64 for the first layer, and 64 to 128 for the second layer, followed by a flattening for the fully connected layer. Leaky relu activator functions were used between each layer. Instance normalisation was used instead of Batch normalisation because batch normalisation is incompatible with WGAN.

An additional change by our design was to dynamically adjust the size of the matrix in order to have as little padding as possible, with a fixed width of 28, because of the multi-mapping scheme using 2x2 blocks. The code for this calculation is implemented in Algorithm 3.

4 OVERVIEW OF CONTRIBUTIONS

Our contribution is a generalised approach to identifying request packets which were not formed correctly by the generator, but which pass more rudimentary tests. This contribution consists of two error measurement algorithms plus a data augmentation algorithm for encoding and decoding the marker values into packets.

The first error measurement exploits the fact single protocol data sets have large amounts of common data that is static between individual samples, and this data can be expected to be static in new generated samples. These indices can be mapped programmatic to maximise coverage, rather of manually selecting indices. The limitation is that some protocols have low percentages of static data.

The second error measurement addresses this by placing marker values in non-static regions of the packet data, which could not be evaluated using the first approach. The accurate reconstruction of these values correlates to correct construction, as we demonstrate.

Marker values are effectively a non-destructive watermark. If the generator reproduces the watermark pattern, there is a correlation to correctly reproducing the surrounding regions.

Given this information, we can then take the first error measurement, from the static regions of the sample, and use this to filter out samples with static errors. We can then do the same with the watermark values that lie in non-static regions, producing a second error measurement. Combining these two measurements, we produce two lists of samples, and exclusively union them, resulting in a minimal number of malformed samples.

5 TECHNICAL DETAILS OF CONTRIBUTIONS

5.1 Measuring Error in Static Sections

Network packets with a shared protocol have common elements, and this causes single-protocol network data sets to contain a measurable percentage of repetitive data within a section of traffic dedicated to a single protocol.

We propose taking advantage of this to create a dictionary of key pairs where each key is a position in the matrix representing a packet and each value is the value which exists at that position in every sample packet, similar to the engram system of Packet2Vec[6], but only keeping positional engrams which appear in 100% of samples.

We take each cell position in the matrix, and step through each matrix in the array, to determine whether there is a difference. If the value '6' appears in every packet at index 3, we can say this value should be '6' in our output set.

Algorithm 1 is our algorithm for iterating over the data set and finding the maximum number of positions in that set with static values. The packets are scanned based on the shortest in the set, before any padding is done, because we do not want to include padding values. We are only interested in positions which are identical across packets, so when packets are different sizes, the shortest length is used. For this pseudo code, we have used a matrix where each row is a packet, although in a coded implementation it might be a list of lists or something similar.

Algorithm 1: Static Position Algorithm

Inputs : Data set:
- batch of samples as matrix of integers where each row is a packet: S
- Length of shortest sample: L

Outputs : Static Indices:
+ dictionary of static indices: D2

```
1  Initialise empty dictionary D;
2  Initialise empty dictionary D2;
3  count := count of rows in S for i := 0 to L do
4      for j := 0 to count do
           // Add key/pair to dictionary
5          D[i] := D[i] ⋃ S[j,i];
6      end
7      if |D[P]| == 1 then
           // Copy valid key to second dictionary
8          D2[P] := D[P];
9  end
10 Return D2;
```

In order to justify this technique, we need to examine sample data sets. For more information on the data sets used in this paper, see Section 6.

As seen in Figure 3, SIP-GAN used a manually selected section of the packet. (Highlighted white section.) Our algorithm uses every section it can, resulting in Figure 4. The hypothesis is that more valid sampling points means more errors detected, thus fewer malformed packets slipping through. Please note that the bottom half of the images have been truncated for space reasons.

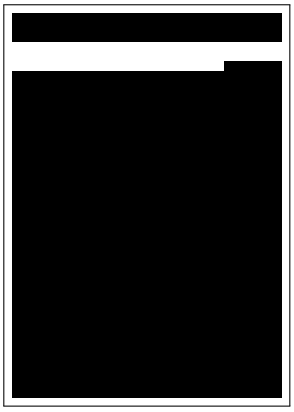

Figure 3: SIP-GAN's Manual Static Indices Map (78 Indices)

As seen in Table 1 our approach finds 11.549% more usable positions than the manual approach that only tests the SIP request line. We are measuring three values. The total length of the packet (Length), the number of cells indexes in the packet that were determined to be static (Usable), and Usable divided by Length (Percentage). The SIP sample has 1092 candidate indices. The manual technique used used 78 indices. Algorithm 1 automatically finds 279 usable indices.

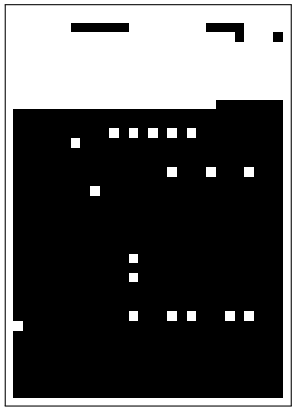

Figure 4: Algorithm 1's Static Indices Map (279 Indices)

Table 1: SIP-GAN Manual Selection vs Algorithm 1 on SIP data set

Type	Length	Usable	Percentage
SIP-GAN	1092	78	14%
Algorithm 1	1092	279	25.549%

As seen in Table 2, different protocols have different ratios of static data.

The advantage of this approach is that it will automatically have, at minimum, as many sample points as the manually selected version. It will also potentially identify various common points in IP addresses, MAC addresses, and other variables if those exist.

Table 2: Frequency of Static Positions identified using Algorithm 1

Type	Total	Usable	Percentage
SIP	1092	279	25.549%
Modbus	132	96	72.72%
ICMP	148	128	86.48%

Now that we have a dictionary of static values, we can use Algorithm 2 to determine the quality of generated samples. It produces two ratings. One is the mean error distance over the set of samples, and the other is the ratio of samples in which no errors were detected.

We have two measurement numbers, listing the ratio of packets that passed inspection and the mean error distance over the total set of samples.

Algorithm 2: Error Measurement Static Positions

Inputs: Data:
- S: batch of packets as matrix of numbers where each row is a packet
- D: dictionary of static positions and corresponding value

Outputs: Error Measurements:
+ Error1: Percentage of packets without detected error
+ Error2: Mean error distance over entire batch

```
r := number of rows in S;
l := length of shortest row/packet in S;
// calculate error 1 by subtracting sample count
   whenever error is found
Error1 := r;
for i = 0 to r do
    for (k, v) ∈ D do
        if |S[i,k] - v| != 0 then
            Error1 := Error1 - 1;
            break;
    end
end
Error1 := Error1 / r;
// calculate error 2
Error2 :=0;
for (k, v) ∈ D do
    T := 0;
    for j := 0 to r do
        T:= T + |(S[j, k] - v)|;
    end
    Error2 := Error2 + T / r;
end
Error2 := Error2 / |D|;
return Error1, Error2;
```

5.2 Data Augmentation for Dynamic Sections

As seen in Figure 5, the Generator can produce packets with correct headers but the payload is garbled. You can see that the INVITE string is present, the sip:service section, but after the upper section, the rest dissolves into chaos.

If the generator creates known good static sections correctly, surrounding data is likely structurally correct. However, as demonstrated by Figure 5 shows, sometimes the non-changing sections are correct, but the dynamic sections are not. The problem is that only a small portion of the packet lies in these static sections that can measure, as seen in Figure 4 and Table 2. Almost 75% of the packet lies in non-measurable regions.

How to solve this problem without resorting to manual protocol analysis? We determined that a viable approach was to place marker values at intervals throughout the dynamic sections of the data, mimicking the static data sections used in the previous approach.

Figure 5: Two SIP packets, one with a garbled payload (left).

This is not the same as Algorithm 2's measurement because for that algorithm, a perfect score means those sections of the packet are correctly constructed. However, correctly reproducing these new marker values does not equate to correctness because these values are arbitrary additions. However, it can be thought of as a confidence value. Accurately reproducing them means the packet is more likely to be correctly constructed.

There were a few ways in which this addition could be achieved. The first would be to add additional cells throughout the data at intervals. However, this would cause an increase to the size of the matrix, which would slow down the GAN.

The second approach would be to modify the encoding scheme from PAC-GAN and SIP-GAN where each pixel/value in the matrix representation of the packet is converted into a 2x2 pixel block of identical values, forming a matrix twice as wide and twice as high, which is useful for error correction because a mean can be taken from each block.

Instead of taking 2x2 blocks for each output value, we can take 3 indices from each 2x2 block then use the fourth index for error detection by placing watermark values in those reserved positions.

The trade-off is a potential loss in accuracy because we're using 3 indices for each block mean calculation instead of 4, and also this change introduces some new complexity into the data, which the generator has to try to learn to reproduce.

What should the marker values be? Our initial thought was to use zeroed out values. But the generator was quickly able to learn to reproduce these values, allowing numerous faulty packets through despite a high score, as seen in Figure 3.

We can exploit the multi-mapping scheme by replacing the fourth index of a block with an escalating value from 0 to 15, over and over, and this forms a secondary error measurement, or a confidence rating. The distinction is that Algorithm 2 has a direct relationship to accuracy. If the static positions are correct the packet is more

Table 3: Zero vs 0-15 Marker Results

Stage	Samples	Minor Errors	Major Errors	Valid %
Zeros	859	104	55	88%
0-15	332	1	0	99.997%

accurately constructed. However, if the confident rating from Algorithm 5 is higher, we are simply more confident that the surrounding regions are correctly constructed.

The results of Algorithm 3 are seen in Figure 6. The leftmost block is a 3x3 matrix (chosen size is merely a visual illustration), representing a set of values in a packet. With a standard multi-mapping algorithm, as used in PAC-GAN and SIP-GAN, this becomes a 6x6 matrix consisting of 2x2 blocks. For 'Multi-Mapped Stride 1', the fourth index of every block is replaced with a number from 0 to 15, modulus 15. With a stride of 4, that becomes every fourth block, calculated using the block's modulus stride length, starting from 0. Values with coloured cell backgrounds represent original values from a packet, while the white background cells represent our marker values.

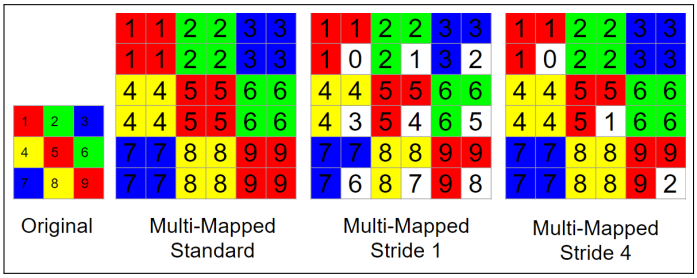

Figure 6: Visual illustration of Algorithm 3's multi-mapping

The trade-off is that introducing this additional complexity into the structure does impact the GAN's overall accuracy, as discussed in the experimental results of Section 5. (See Figure 4).

How is this scheme achieved? Algorithms 3 and 4 are modified versions of the multi-mapping encoding/decoding algorithms from SIP-GAN[1]. The difference is that if a position in the matrix we are encoding or decoding is not in the list of static position dictionary, we encode/decode differently, treating the fourth pixel of that block as a special value. If the value is in the list, we use the standard SIP-GAN multi-mapping. There is no point in adding these markers to blocks that are already measured using Algorithm 2.

In order to understand Algorithms 3 & 4, we need to define a bijective function, denoted as (f) in Algorithm 3 and its inverse f^{-1} in Algorithm 4. This function, which is taken from SIP-GAN[1], converts numbers from their original range into an expanded range and back again. Our error measurement calculations work off the decoded values between 0-15, not the range 0-255 which is remapped into floats -1 to 1 for the GAN, because it is expected that the GAN will have fluctuation in values, thus why we do this conversion.

(1) To encode, the integer from 0 to 15 (corresponding to a single hex character) is converted to a range between 0 and 255 with formula 'number * 16 + 16 // 2'.
(2) To decode, the integer from 0 to 255 is converted to a range between 0 and 15 with the formula 'floor(((number - 8) / 16) + 0.5)'.

Another change in our encoding/decoding algorithm is that SIP-GAN featured an adjustable multi-mapping stride parameter, which we have hard-coded to 2, like PAC-GAN. Our stride parameter is instead used to control the stride of marker index placement.

The output matrix in Algorithm 3 is defined as $I_{(m \times n)}$. The value of m is always 28 for our purposes, and the value of n is derived from the length of the sample to produce a matrix that will fit the multi-mapped sample size. The word 'ceil' in the pseudo code means to take the ceiling of these numbers because we need our resulting packet to fit.

Algorithm 3: FPAC-GAN Encoding

Inputs:
- S: packet to encode represented as array of integers
- p: list of static positions
- l: stride value

Outputs:
+I: matrix corresponding to S

```
m := 28;
n := 4 * ceil((|S| / 28) / 4);
I := matrix of size [m X n];
index := 0 counter := 0
for i := 0 to n and step 2 do
    for j := 0 to m and step 2 do
        if enhanced = True and index % l = 0 and index ∉ p
         then
            I[i : i; j : j] := f(S[index]);
            I[i : i; j : j + 1] := f(S[index]);
            I[i : i + 1; j : j] := f(S[index]);
            // Use counter value for fourth index
               of block
            I[i : i + 1; j : j + 1] := f(counter % 15);
            // counter is updated separately from
               index due to variable stride
            counter := counter + 1;
        else
            I[i : (i + 2); j : (j + 2)] := f(S[index]);
        end
        index := index + 1
    end
end
return I;
```

By only encoding the markers into positions that do not exist in the static value list, we introduce as little extra data into the data set as possible because added complexity through markers makes

training the generator more challenging, as shown in Table 3 of Section 6.2.

For Algorithm 4, we decode similarly to SIP-GAN but if in enhanced mode, only take the mean from 3 indexes in each 2x2 block that is not in the list of static positions, storing the fourth index into an array.

Algorithm 4: FPAC-GAN Decoding

Inputs : Data:
- I: matrix corresponding to an encoded packet
- p: list of reserved positions
- l: length of stride
- enhanced: enhanced mode

Outputs : Data:
+ S: a matrix corresponding to decoded packet
+ M: a vector corresponding to marker values from packet

```
m := count of columns in I;
n := count of rows in I;
S := vector of size [m/2 * n/2];
index := 0;
index2 := 0;
for i = 0 to n and step 2 do
    for j := 0 to m and step 2 do
        if enhanced and index % l = 0 and index ∉ p then
            S[index] = f^-1((I[i,j] + I[i, J+1] + I[i+1, j]) / 3);
            M[index2] = f^-1(I[i+1, J+1]);
            index2 := index2 + 1;
        else
            S[index] = f^-1((I[i,j] + I[i, J+1] + I[i+1, j] + I[I+1,
              j+1]) / 4);
        end
        index = index + 1;
    end
end
return S, M;
```

With these changes to the encoding/decoding process, we can now implement a second error measurement.

5.3 Measuring Error in Dynamic Sections

Algorithm 5 is similar to Algorithm 2, however it uses marker values from Algorithm 4, derived from the generated packets. When the markers are decoded they form a sequence that should be a series of numbers from 0 to 15, making comparison simple.

After training for 100 epochs, the new error measurement produced 158/991 samples had significant errors. This was not an ideal result. The Generator was reproducing the zero values so quickly that the correlation to quality was lost. Our solution was to mimic the complexity that exists in typical static data in network packets, which is unlikely to be all-zeroes. This would slow the rate at which the GAN learned these values to strengthen the correlation between reproduction and sample accuracy. So we used an escalating number from 0 to 15, which fits into a single hex character. The numbers incremented from 0 to 15, then looped over to 0 again.

Algorithm 5: Error Measurement using Extended Markers

Inputs : Data:
- M: 2D matrix of numbers where each row contains the markers from a single packet

Outputs : Error Measurements:
+ Percentage of packets with no detected error: Error1
+ Mean error distance over set of packets: Error2

```
r := number of rows in M;
l := number of columns in M;
// Calculate error value 1 & 3
K := r for i := 0 to r do
    for j := 0 to l do
        T := |(M[i,j] - i % 15)|;
        if T not equal to 0 then
            K := K - 1;
    end
end
Error1 := K / r;
// Calculate error value 2
Error2 :=0 for i := 0 to r do
    T := 0;
    for j := 0 to l do
        T := T + |M[i, j] - R[j]|;
    end
    Error2:= Error2 + T / r
end
// for this algorithm we use the number of
   columns/count of marker values
Error2 := Error2 / l;
Return Error1, Error2;
```

With this new approach, 19/396 samples were faulty. We had successfully removed a higher ratio of faulty samples from the original set of 1,000 at the cost of losing more samples overall to the aggressive error checking.

5.4 Filtering Data Set using Both Error Values

With our two existing approaches, we have two error check methods with complementary advantages and flaws. One detects errors in the common packet values, the other detects likely abnormality in dynamic packet values. We combine these error checks to produce a final set that is an exclusive union of the two results, which will be an empty set if no matches were made:

(1) The GAN generates a batch of samples.
(2) Using the first error check, we find the indexes of every sample that passes, creating Filtered Set 1.
(3) We repeat this for the second error check, creating Filtered Set 2.
(4) Then we take a union of the two sets of indexes, and pull these indexes from the batch of samples to create our output set.

(5) We now have a final output set that should have minimal errors.

As shown in Figure 1 and Figure 2 the two stages are interconnected. The known safe values are used to derive an optimal multi-mapping scheme, and two sub-sets are combined for a final data set.

6 EVALUATION OF CONTRIBUTIONS

Because this paper slightly overlapping contributions, we have chosen to present the experimental results in sequence, using Algorithm 2, then the combines results of Algorithm 2 and Algorithm 5.

6.1 Experimental Design

Different protocols have varying levels of repetition, variety, and complexity. A set of all zeroes has low complexity and can be learned by a GAN very quickly. A more structurally complex pattern in the data requires more training time to reproduce. Thus multiple data sets were used for evaluation, although SIP was the primary reference. All of the data sets consisted of request packets.

- The modbus dataset used in 'Denial of Service Attacks: Detecting the frailties of machine learning algorithms in the Classification Process'[4] was taken and Modbus request packets were extracted to form a new Modbus request dataset with 14,119 samples.
- A new ICMP request dataset was created by running a script containing the IP addresses described in PAC-GAN[1], as well as various additions, and contained 154 unique IP addresses and 7,737 unique ICMP request packets.
- The SIP dataset from the SIP-GAN[9] project was taken verbatim, consisting of 20,000 unique SIP protocol packets.

The SIP protocol[9] is particularly useful for our work because it consists of multiple layers, including a plain text payload that varies in position from one sample to another. Modbus and IMCP were also used for experimental results, but due to space constraints only Modbus and SIP were fully tabled with 100 and 500 epoch results.

In order to generate the output data sets used for evaluation, the following procedure was used:

(1) We train the GAN for 500 epochs, to match SIP-GAN. We take our measurements from the last GAN training session which showed improvement.
(2) We compare the number/ratio of valid samples in the output sets manually.
(3) We train the GAN for 100 epochs, to demonstrate a truncated training time scenario.
(4) We compare the number/ratio of valid samples in the output sets manually.

Learning rate was 0.0002, batch size was 32, noise dimension was 64. Optimiser beta values were 0.5 and 0.999. The stride rate of Algorithm 3 and Algorithm 4 was 4.

Integer values from 0 to 255 were scaled to a range between -1 and 1, using the formula '(number - 125.5) / 125.5' for encoding and '(number + 1.0) * 127.5' for decoding.

Modbus was tested using BPF command 'modbus && (ip.addr == 172.27.224.80 && ip.addr == 172.27.224.250 && eth.addr == 00:0c:29:e6:14:0d && eth.addr == 00:80:f4:09:51:3b)'.

ICMP was tested using the BPF command 'icmp.type==8 && ip.addr == 192.168.1.200'.

The measurements were taken from the generated samples at the end of each epoch, generating 1,000 samples and performing tests on them. So if 1,000 samples were generated and 500 were valid using a particular algorithm, that would be 50% accuracy. The results for each measurement were updated each time there was an improvement, which results in smoother graph lines than graphing the results of each epoch.

6.2 Experiment 1 Static Error Results

The first experiment contrasted Contribution 1 against the previous approach of non-specific BPF filtering and the protocol-specific analysis of SIP-GAN.

Figure 7 tested BPF filtering against Algorithm 2. A lower measurement was more desirable because it indicated errors being detected that the BPF filter had missed.

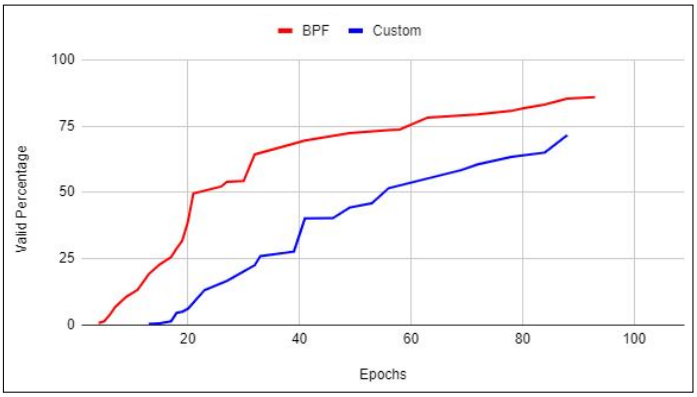

Figure 7: ICMP vs Algorithm 2

Figure 7 shows that using simple 'icmp' BPF filters under-estimates packets with detectable errors by 15% compared to Algorithm 2.

The GAN was run for 500 epochs on the SIP dataset consisting of 20,000 samples. Each epoch, the accuracy of samples was measured by first using the SIP-GAN evaluation, which tested the values between indexes 84 and 162, inclusive. Then we used FPAC-GAN's Algorithm 2.

The results in Figure 8 show that by selecting 279 sampling points instead of 78 sampling points, we detect more samples that contain errors. The SIP-GAN approach of manually selecting a smaller number of points under-estimates the error in the set. Thus our measurement is superior at detecting whether a sample contains errors in static sections, and we are able to calculate this in a generalised manner.

Stepping forward the results in section Section 6.4, we find that after being trained for 500 epochs, the SIP-GAN approach of only testing the SIP request values yielded an accuracy rate of 84.1%, while Algorithm 2 yielded 75.1%, meaning that the previous approach missed 9% of the set being corrupted.

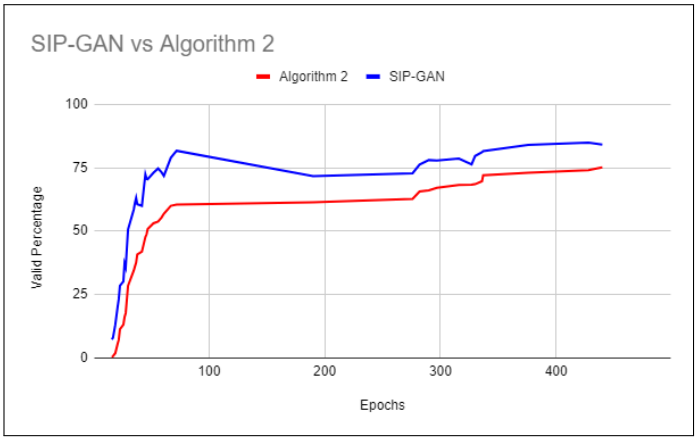

Figure 8: SIP-GAN vs Algorithm 2 ratio of validated samples.

6.3 Experiment 2 Data Augmentation

In our second experiment, we tackled the problem of trying to detect error in packets that had a low ratio of measurable static positions, as shown in Figure 5 in Section 5.2.

The results for this section are shown in Section 6.4 for the sake of keeping things together, but first we need to examine the error penalty of Algorithm 3 & 4. As seen in Table 4, there is a loss of accuracy over the set for SIP, but not for ICMP, where it improves instead. We theorise this is because the low ratio of markers.

Table 4: Peak Accuracy at 100 epochs

Protocol	Markers Added	Accuracy %
SIP	No	66.3%
SIP	Yes	55.8%
ICMP	No	64.3%
ICMP	Yes	72.6%

However, as discussed in section 6.4, we are able to counteract this accuracy loss to some degree.

6.4 Experiment 3 Combined Filtering Results

The samples that Algorithm 5 detects as invalid are detected using a different criterion to samples detected using Algorithm 2. Thus, detecting some errors that Algorithm 2 could not, but fails at detecting some errors its counterpart successfully caught. However, by combining the two output sets to create a final filtered set, we would lose a number of potentially viable packets, but might have a final subset with a lower error ratio. We set out to prove this in our third and final experiment.

The GAN was trained for 100 and 500 epochs on the SIP data set as the primary test, but the Modbus and ICMP data sets were also used.

For SIP protocol packets, the hand-crafted SPF command *'sdp && ip.addr == 10.0.0.1 && ip.addr == 10.0.2.15 && eth.addr == 08:00:27:07:0F:39 && eth.addr == 52:54:00:12:35:02'*, paired with manual visual inspection was used to analyse the packets for accuracy.

To understand what the tabled results mean mean, we need to define the terms used in Figure 5, Figure 6, and Figure 7:

- Original is the original data measured for errors using a BPF filter and manual inspection.
- OG + Filter 1 is the data processed using Algorithm 2.
- Enhanced is the data with markers added using Algorithm 4, measured the same way as Original.
- Enhanced + Filter 1 is the data encoded with Algorithm 4 processed/filtered using Algorithm 2
- Enhanced + Filter 2 is the data encoded with Algorithm 4 processed/filtered using Algorithm 5
- Combined is the results of Algorithms 2 and 5 combined using an exclusive union operation, as shown in Figure 2.

Table 5: 100 epoch SIP Results

Phase	Ratio	Accuracy	Error %
Original	663/1000	66.3%	33.7%
OG + Filter 1	623/686	92.1%	7.9%
Enhanced	558/1000	55.8%	44.2%
Enhanced + Filter 1	544/568	95.8%	4.2%
Enhanced + Filter 2	244/277	88.1%	11.9%
Combined	243/246	98.8%	1.2%

Table 6: 500 epoch SIP Results

Phase	Ratio	Accuracy	Error %
Original	745/1000	74.5%	25.5%
OG + Filter 1	733/751	97.6%	2.4%
Enhanced	692/1000	69.2%	30.8%
Enhanced + Filter 1	682/713	95.7%	4.3%
Enhanced + Filter 2	257/269	95.5%	4.5%
Combined	256/259	98.8%	1.2%

Table 7: 500 Epoch Modbus Results

Phase	Ratio	Accuracy	Error %
Original	627/1000	62.7%	37.3%
OG + Filter 1	597/592	99.2%	0.8%
Enhanced	810/1000	81%	19%
Enhanced + Filter 1	749/760	98.6%	1.4%
Enhanced + Filter 2	503/614	81.9%	18.1%
Combined	466/472	98.7%	1.3%

With 100 training epochs, SIP-GAN was off by 19.8%, claiming that 86.1% of packets were correct, while the true value was 66.3% determined through manual analysis after custom BPF filtering. For our work, Algorithm 2 yielded a result of 68.6%, off by 2.3%. When we added Algorithm 5 and combined the results we had a final set with only 1.2% of packets being malformed.

After 500 epochs, the claimed accuracy rating using SIP-GAN's error measurement was 84.1%. Our custom BPF filter found a true rate of 74.5%. This means a missed error rate of 9.6%.

Using simple BPF filters also yielded poor results. On the 500 epoch set, 'sip' claimed 82.7% of samples were correct, and 'sdp' claimed 79.2%. Since the true value was 74.5% this fell short of our results.

For the SIP protocol data set, whether we trained the generator for 100 epochs or 500 epochs, we attained a 98.8% validity rate, which is higher than the 92.1% (100 epochs) and 97.6% (500 epochs) of Algorithm 2 alone. Our methodology eliminates faulty samples at a higher rate than non-faulty samples, meaning that we can generate more samples and maintain this error/accuracy ratio.

The results in Table 7 reveal some shortcomings where the combined algorithm is slightly less accurate than Algorithm 2. This is because the Modbus protocol has a much lower percentage of dynamic sections compared to SIP, The ratio of valid samples in the reference set does improve with the addition of markers, however. This is also seen in the ICMP results of Table 4 in Section 6.3.

7 DISCUSSION & FUTURE WORK

Our contributions were able to increase the accuracy and effectively decrease the training time of GAN generated network samples because we could, in optimal conditions, obtain a batch of correctly constructed samples with less training time by correctly identifying the malformed samples and omitting them. There are caveats to be aware of, however. Algorithm 2 is always better than SIP-GAN's approach, but Algorithm 5 should only be used if the costs of adding additional markers to the data does not outweigh the accuracy gains in detecting malformed samples. It would probably be advisable to only use the extended technique if a certain percentage, perhaps over 50%, of the sample were not measurable using Algorithm 2.

With the SIP data set, training the generator for 100 epochs took two and half hours. Training it for 500 epochs took 12 hours, and the resulting filtered accuracy on the SIP data set was the same, as seen in Figures 5 & 6. However, more work must be done to determine the limitations of this shortcut.

There is a risk of non-obvious mode collapse. A plausible technique for detecting this would be to scan the the non-static sections and calculate a difference value. If the difference value decreased substantially below the source data set, this might indicate a problem.

We could combine the error and confidence values for a combined value, and used this to select our optimal stopping point. Our goal is to keep training until the Generator doesn't improve anymore. This, we could normalise the two values, combine them, and then use them as a single error rate value. This wasn't attempted due to time constraints.

The pattern we have used for storing markers in the data is potentially sub-optimal. The step values, currently 0-15, could be adjusted, and the stride size, currently 4, could potentially be increased to find a balance between base sample accuracy and our ability to detect errors. We're fighting against a 10% accuracy loss from Algorithm 3, and if the accuracy loss could be minimised without affecting our ability to detect malformed samples, it might yield better results.

By exploiting the characters of network packets, and augmenting data with markers, we were able to better determine sample quality than previous approaches, and generate synthetic network packet output data sets with a higher level of accuracy.

Machine learning and artificial intelligence are fast moving topics. The optimal approach for generating synthetic network data sets for research purposes may change quickly. However, the proposals in this paper will plausibly transfer to new models and approaches. For instance, MirageNet[8], the most recently published model for synthetic packet generation at the time of writing, could plausibly utilise a similar technique.

REFERENCES

[1] 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)*. 0728–0734. https://doi.org/10.1109/IEMCON.2019.8936224

[2] Baik Dowoo, Yujin Jung, and Changhee Choi. 2019. PcapGAN: Packet Capture File Generator by Style-Based Generative Adversarial Networks. In *2019 18th IEEE International Conference On Machine Learning And Applications (ICMLA)*. 1149–1154. https://doi.org/10.1109/ICMLA.2019.00191

[3] Gilberto Flores, Marcos Paredes-Farrera, Emmanuel Jammeh, Martin Fleury, and Martin Reed. 2003. OPNET Modeler and Ns-2: comparing the accuracy of network simulators for packet-level analysis using a network testbed. *WSEAS Transactions on Computers* 2 (01 2003).

[4] Ivo Frazão, Pedro Henriques Abreu, Tiago Cruz, Hélder Araújo, and Paulo Simões. 2019. Denial of Service Attacks: Detecting the Frailties of Machine Learning Algorithms in the Classification Process. In *Critical Information Infrastructures Security*, Eric Luiijf, Inga Žutautaitė, and Bernhard M. Hämmerli (Eds.). Springer International Publishing, Cham, 230–235.

[5] Prasanta Gogoi, Monowar H. Bhuyan, Dhruba Kumar Bhattacharyya, and Jugal Kumar Kalita. 2012. Packet and Flow Based Network Intrusion Dataset. In *IC3*.

[6] Eric L Goodman, Chase Zimmerman, and Corey Hudson. 2020. Packet2vec: Utilizing word2vec for feature extraction in packet data. *arXiv preprint arXiv:2004.14477* (2020).

[7] Amar Meddahi, Hassen Drira, and Ahmed Meddahi. 2021. SIP-GAN: Generative Adversarial Networks for SIP traffic generation. In *2021 International Symposium on Networks, Computers and Communications (ISNCC)*. 1–6. https://doi.org/10.1109/ISNCC52172.2021.9615632

[8] Santosh Kumar Nukavarapu, Mohammed Ayyat, and Tamer Nadeem. 2022. MirageNet - Towards a GAN-based Framework for Synthetic Network Traffic Generation. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*. 3089–3095. https://doi.org/10.1109/GLOBECOM48099.2022.10001494

[9] Tuomas Nurmela. 2003. Session Initiation Protocol. (11 2003).

[10] Markus Ring, Sarah Wunderlich, Deniz Scheuring, Dieter Landes, and Andreas Hotho. 2019. A survey of network-based intrusion detection data sets. *Computers & Security* 86 (2019), 147–167. https://doi.org/10.1016/j.cose.2019.06.005

[11] A. Sychugov and M. Grekov. 2021. Using wasserstein generative adversarial networks to create network traffic samples. *AIP Conference Proceedings* 2402, 050070. https://doi.org/10.1063/5.0072057