GANFuzz: A GAN-based industrial network protocol fuzzing framework

Zhicheng Hu[1], Jianqi Shi[1,2,*], YanHong Huang[1,3], Jiawen Xiong[1], Xiangxing Bu[1]
[1]National Trusted Embedded Software Engineering Technology Research Center,
East China Normal University
[2]Hardware/software Co-Design Technology and Application Engineering Research Center,
East China Normal University
[3]Shanghai Key Laboratory of Trustworthy Computing,
East China Normal University
Shanghai, China
{zchu,jqshi,yhhuang,jwxiong,xxbu}@sei.ecnu.edu.cn

ABSTRACT

In this paper, we attempt to improve industrial safety from the perspective of communication security. We leverage the protocol fuzzing technology to reveal errors and vulnerabilities inside implementations of industrial network protocols(INPs). Traditionally, to effectively conduct protocol fuzzing, the test data has to be generated under the guidance of protocol grammar, which is built either by interpreting the protocol specifications or reverse engineering from network traces. In this study, we propose an automated test case generation method, in which the protocol grammar is learned by deep learning. Generative adversarial network(GAN) is employed to train a generative model over real-world protocol messages to enable us to learn the protocol grammar. Then we can use the trained generative model to produce fake but plausible messages, which are promising test cases. Based on this approach, we present an automatical and intelligent fuzzing framework(GANFuzz) for testing implementations of INPs. Compared to prior work, GANFuzz offers a new way for this problem. Moreover, GANFuzz does not rely on protocol specification, so that it can be applied to both public and proprietary protocols, which outperforms many previous frameworks. We use GANFuzz to test several simulators of the Modbus-TCP protocol and find some errors and vulnerabilities.

CCS CONCEPTS

• **Security and privacy → Network security**; • **Computing methodologies** → *Neural networks*;

KEYWORDS

industrial safety, industrial network protocols, implementations, fuzzing, generative adversarial network, generative model, protocol grammar

*Corresponding author.

CF '18, May 8–10, 2018, Ischia, Italy

ACM ISBN 978-1-4503-5761-6/18/05…$15.00
https://doi.org/10.1145/3203217.3203241

ACM Reference Format:
Zhicheng Hu[1], Jianqi Shi[1,2,*], YanHong Huang[1,3], Jiawen Xiong[1], Xiangxing Bu[1]. 2018. GANFuzz: A GAN-based industrial network protocol fuzzing framework. In *CF '18: CF '18: Computing Frontiers Conference, May 8–10, 2018, Ischia, Italy.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3203217.3203241

1 INTRODUCTION

Industrial control systems(ICS) are necessary facilities for the proper running of critical infrastructures, such as petroleum and petrochemical industries, electrical power system, nuclear power plants, etc. The early ICS are closed systems that independent of the external network, so they are unlikely to be attacked. With the development of industry and manufacturing, ICS get developed and become inter-networked and inter-connected, which exposes ICS to external attacks, such as worms, trojans, posing significant threats to its normal operation. In fact, famous attacks towards the industrial network, such as Stuxnet, BlackEnergy have caused heavy damages around the world, which demonstrate the severity of current industrial security situation.

As a matter of fact, considerable cyber attacks on ICS result from the security flaws of industrial network protocols(INPs). On the one hand, to provide real-time, anti-interfere and controllable communications, the protocol format of most INPs is designed to be short-frame and unencrypted, which makes the conversations can be easily intercepted and exploited, posing enormous threats to the ICS. On the other hand, developers may fail to properly handle the protocol details, such as boundary values, the interrelation between fields, etc. when implementing INPs in real scenarios according to the specifications, since interpreting specs are tedious and error-prone tasks, especially under the condition of the limited time and workforce. The errors introduced during protocol implementation is known as implementation errors. If these errors are found and exploited by hackers, it will cause immeasurable damage. Thus, one effective solution to improve industrial safety is to find out and remedy those implementation errors.

Fuzz testing(fuzzing)[19, 20] is used widely for revealing errors in implementations of protocols, which is a testing technique that unveils security loopholes and program errors by feeding system under test(SUT) with massive amounts of random and unexpected test inputs. Most research on fuzzing today generate test data from the SUT's input grammar. In the area of protocol fuzzing, testers

have to figure out the protocol grammar to guide the generation of test inputs. But building protocol grammar is not easy. For open protocols, the protocol grammar is established by manually reading and analyzing the protocol specification, time-consuming and labor-extensive. As for proprietary protocols, such as Harris-5000, whose specs are private or only public in a limited scope, reverse engineering techniques are used to extract the protocol grammar from network traces, which mainly rely on manual efforts at first. In order to save labor and time, a better way is to automate the reverse engineering process, but it demands access to the source or binary code and still needs some human interaction.

Motivated by these issues, we propose a novel test case generation method for testing implementations of INPs, in which the protocol grammar is uncovered from real-world protocol messages automatically by using deep learning techniques. More specifically, we leverage the generative adversarial network(GAN) [12] to train a generative model over the dataset of protocol messages to reveal the protocol grammar. By taking advantage of the learned generative model, we can produce fake but plausible messages, which are ideal test cases for testing implementations of INPs. Based on this approach, we present the GANFuzz, an intelligent and automatical industrial network protocol fuzzing framework. In contrast to previous work, GANFuzz present an attempt at using GAN for this problem, providing a new solution. Moreover, GANFuzz releases the dependence on protocol specification, so that it can not only be applied to public protocols but also proprietary protocols, which outperform numbers of preceding frameworks. The main contributions of our work are threefold:

1. We propose a new test case generation approach and build a fuzzing framework(GANFuzz) on top of it.
2. To increase the code coverage and testing depth, we present three clustering strategies for classifying the protocol messages from different dimensions, using which the learned generative models can generate more diverse and well-formed test case.
3. In our experiment, we test several Modbus-TCP simulators by using the GANFuzz prototype and successfully reveal some new flaws and already-known issues.

The remainder of this paper is organized as follows. In Section 2, we present some preliminary knowledge about the deep learning models used in this study. Section 3 introduces our proposed test case generation method. In Section 4, we present the design and implementation of our fuzzing framework–the GanFuzz. We evaluate our approach by testing several simulators for Modbus-TCP protocol in Section 5. In Section 6, we discuss the related work. Finally, a brief conclusion and future work are given in Section 7.

2 PRELIMINARY

In this section, we first give an overview of the generative adversarial network, then introduce the recurrent neural network and the convolutional neural network briefly.

2.1 Generative adversarial network

Generative Adversarial Network[17] is a deep neural network architecture proposed by Goodfellow et al., which have been applied wildly in generating realistic data, such as images, videos, texts[4, 16, 30] and shown very striking results. GAN is composed of a generative model(generator) and a discriminative model(discriminator), in which the generator's goal is to fool the discriminator to believe that the data created by it is from real data distribution, while the discriminator aims to distinguish the synthetic data from real ones. To achieve this, the generator is fed with random noises sampled from a prior distribution, such as Gaussian distribution to estimate a generator function G that approximate the real data distribution. The discriminator D is trained to maximize the probability that the generated data and the real data are correctly classified. Conversely, the generator is trained to maximize the probability that the generated data is misclassified.

Through this adversarial training way, we can estimate a generative model that covers the distribution function of real-world data. In this work, we use GAN to train generative models to deprive the protocol grammar of real protocol messages.

2.2 Recurrent neural network

Recurrent neural network(RNN) is a kind of neural network used wildly in various of NLP tasks. RNN is able to learn the long term dependence of input sequence by capturing the context information so far and stores it in a "memory" unit–the hidden state. It transfers the input sequence $X_{1:n} = (x_1, x_2, ..., x_t, ..., x_n)$ into a series of hidden states $h_{1:n} = (h_1, h_2, ..., h_t, ..., h_n)$, where for time step t, the hidden state h_t is dependent on the previous hidden state h_{t-1} and the input x_t and it produce an output y_t. Using RNN, we can learn the conditional distribution $p(x_t|(x_1, x_2, ..., x_{t-1}))$ of the input sequences.

We use RNN as the generator in GAN in this study.

2.3 Convolutional neural network

Convolutional neural network(CNN) is a class of neural network that has shown great success in the field of computer vision. A CNN is usually composed of many layers, including the input layer, convolution layer, pooling layer, fully-connected layer, the output layer, etc. CNN specializes in extracting advanced features from input data. Recently, CNN attracts a lot of attention in NLP community and have achieved favorable results in various of text classification tasks [15, 31].

In this work, CNN serves as the discriminator in GAN.

3 METHODOLOGY

In this section, we begin with giving an overview of our proposed test case generation method, which is the fundamental method of our proposed GANFuzz framework. After that, we present the approach in detail.

3.1 Overview of approach

Our ultimate goal is to conduct fuzz testing towards implementations of INPs. One of the first problems to be solved is to find out efficient methods for generating test data. In this work, we use GAN to train generative models on the dataset of protocol messages. Under diverse training settings, the learned generative models grasp the knowledge of the protocol grammar in various degree. By making use of the generative models, we can produce spurious protocol messages that share varying similarity with the

authentic ones, which are favorable test cases for testing. The main steps of our approach are as follows:

1. *Clustering the messages.* Given a corpus of real-world protocol messages as the training data, we employ three clustering strategies to provide three means to classify the data. For each strategy, the features and functions we are concerned w.r.t the protocol messages are different. It allows us to carry out fuzzing from different dimensions. And it helps to improve the code coverage and testing depth.
2. *Learning the protocol grammar.* We model the protocol grammar learning problem as a process of estimating a generative model by using the generative adversarial network and the SeqGan algorithm[30]. Through deep learning training, the generative model reveals the protocol grammar from real protocol messages automatically.
3. *Generating test cases.* The learned generative model is capable of producing sequences that similar to realistic protocol messages. Making use of the generative model, we can generate test cases for fuzz testing.

In the following, we describe these three steps in detail.

3.2 Clustering the messages

Code coverage and testing depth are two essential concepts in fuzzing. High code coverage and testing depth lead to promising fuzzing results[18]. In general, the more diverse the test cases, the higher code coverage we are likely to reach, and the better we understand the protocol grammar, the better the testing depth we can realize. Based on these points, we present three clustering strategies, i.e., NoClustering, SameLengthClustering, and AdvancedClustering, for dividing the training data. NoClustering strategy target for increasing test case diversity, while the other two strive to master the protocol grammar better, aiming to achieve better code coverage and testing depth respectively. As a result, the fuzzing effect is improved. In the following, we introduce the strategies in detail.

- **NoClustering**: Using this strategy, all the messages in the training dataset are treated as one category, which implies that we try to capture the grammar of all message type in just one generative model. It is difficult, since protocol messages belong to different class may differ dramatically from each other in both length and structure. Hence, it leads to the roughest generative model. As a result, the messages generated from the learned model are ill-formed, so that the SUT may reject the vast majority of them. In spite of its weakness, this strategy is still worth performing, since the learned generative model tends to produce the most diverse test cases compared to the other two, which indicates a good code coverage.
- **SameLengthClustering**: Under this strategy, the messages are classified by employing the message length as the metric. To begin with, we gather the data with the same length to the same group. Then, groups whose data quantity smaller than a given threshold(e.g., 2000) are moved to the longer adjacent group, while groups contain more data than the threshold are left unchanged. This clustering strategy seems crude, but it is indeed efficient and succinct in many cases since, for lots of INPs, their length of the same message type is similar or identical. Compared to the NoClustering strategy, this strategy can achieve a better understanding of the protocol grammar, which leads to better testing depth.
- **AdvancedClustering**: Employing this strategy, we first build the character set of the messages and establish a mapping from the characters to unique integer numbers. Then, each message is converted into the vector representation, where each character is represented by the corresponding integer number. After that, we normalize the vectors by dividing by the maximum integer number. Finally, we categorize the vectors by applying the k-means clustering algorithm, with Euclidean distance as the similarity metric. Compared with the other two clustering strategy, this clustering strategy offers a more reasonable partition of the messages and can be applied in more general cases. Using this strategy, we are more likely to cluster the messages with the same function together, which lets us be able to deal with the grammar of each message type separately so that our generative models can understand the protocol grammar better. Thus, we tend to acquire the most well-formed test data. Thanks to these advantages, we are more likely to dig out severe bugs and achieve best testing depth. By the way, to speed up the clustering efficiency, we use CUDA parallel programming[23].

We realize these clustering strategies in GANFuzz and evaluate the actual effect in the experiment section.

3.3 Learning the protocol grammar

Recently, lots of efforts have been made to utilize GAN for producing meaningful character strings from scratch, and have achieved striking results. The main idea behind this is to use the GAN to learn a generative model over a corpus of character strings to grasp the statistical characteristics or grammar of those data. Having such a generative model enable us to create new strings similar to the original ones. The idea is same as the idea of our approach. Since for INPs, their messages are formatted character strings, in essence, we can leverage the GAN to obtain the protocol grammar in the same way.

To this end, we begin with modeling our grammar learning problem with the generative adversarial network. After that, we design specific structure for the GAN model and settle down the training settings. Finally, we proceed with the model training step. After finish training, the learned generative model implicitly cover the protocol grammar.

3.3.1 Modeling protocol grammar learning with GAN. Given a corpus of protocol messages, we view each message as a sequence of characters $M_{1:T} = (m_1, m_2, ..., m_t, ..., m_T)$, $m_t \in V$, where V is the character set of messages. Follow the idea of GAN, each message can be thought as the result of a sample from an unknown distribution function Q_m. As a result, the problem of learning the protocol grammar is converted into the problem of estimating the distribution function Q_m.

To solve function Q_m, we train a generative model G_θ to produce new messages, while train a discriminative model D_ϕ as a classifier to distinguish the generated messages from realistic ones, where θ, ϕ are parameters of the generator and the discriminator. Q_m can

be approximated by training GAN properly until the generator can generate data share very high similarity with the real messages.

However, the protocol message is discrete data, which is non-differentiable, so that backpropagation cannot be applied directly during training. To work around this problem, we employ the SeqGan algorithm [30], which uses reinforcement learning [26] techniques to update the generator's parameters rather than backpropagating from the discriminator to the generator directly.

As a result, applying GAN to model the grammar learning problem of INPs can be depicted as Figure 1, where G represents the generator, and D refer to the discriminator. The x_i, y_i are characters in the collection V, and those strings, such as "$x_1x_2x_3...x_{42}$" and "$y_1y_2y_2...y_{43}$" indicate the real messages and generated messages respectively. The generator is modeled as a stochastic decision process, where the state is the generated message piece so far, and the action is next character to produce. The generator uses the rewards from the discriminator to update its parameters. The rewards are the probability values $p_y^1, p_y^2, ..., p_y^n$, which represent the chance that how likely the discriminator regards the fake messages as real ones. The rewards come from two sides. One is from the wholly generated messages, whose rewards are the outputs of the discriminator, while the other one is from intermediate tokens whose rewards are obtained from the discriminator by employing the Monte Carlo search with a rollout policy. Policy gradient method[27] is employed to update the parameters of the generator with the rewards.

Once the generator is updated, we are ready to improve the discriminator. We use fake messages produced from the generator along with the real messages as the training data for the discriminator. We train the discriminator to maximize the probability that both the real data and generated data are classified correctly.

To sum up, by employing GAN and the SeqGan algorithm, we can model the protocol grammar learning problem as above.

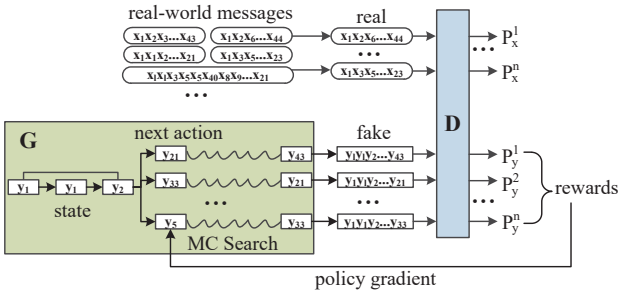

Figure 1: Applying GAN to learn the grammar of protocol messages

3.3.2 Determining the model. GAN only provides a general framework for training a generative model. We have to settle down its two components before getting into the practical model training. In our settings, we use a recurrent neural network(RNN) with LSTM [13] cells as the generative model and a convolution neural network(CNN) as the discriminative model. The structure detail of CNN and RNN should be determined in accord with the structural complexity of the protocol messages. Generally, the longer the protocol messages, the more hidden states the RNN model should contain, and more convolution layers may be added to the CNN model. In our experiment, the RNN is composed of an embedding layer, followed by an LSTM layer that contains 32 hidden states, and the CNN is comprised of an embedding layer, followed by convolutional, max-pooling and softmax layers.

3.3.3 Training the model. Up to this point, we are ready to train the model. Since we demand the training data are length-equal, to keep the integrity of the message, we first calculate the maximum message length and then pad the messages shorter than the maximum length to maximum length with the special character that not included in the character set V. Later, we transfer the training data into numerical representation and launch the model training.

The training goes in three steps. Firstly, we pre-train the generative model for several epochs on the real message dataset. An epoch refers to iterating over all the protocol messages in the training dataset. Then we use the pre-trained generative model to produce the same amount of data as the training dataset. These generated data along with the real messages are fed to pre-train the discriminative model. After pre-training, we train the discriminative model and the generative model alternately, where every time we finish training the generative model for one epoch, we proceed with training the discriminative model for several epochs. We repeat this process until the generative model is able to create data that very similar to the real messages.

In general, the purpose of training GAN is to estimate a generative model that able to produce data that as close as possible to the original ones. But in this work, since the final purpose of learning is to execute fuzz testing, we not only want to get the best generative models, but also the semi-good ones. Having different models that understand the protocol grammar in varying degrees allow us to generate test data with different similarity degree, which introduces more randomness and increases the diversity of test cases, thus improving the fuzzing result. For this reason, we repeat the alternating process for different times(e.g., 5, 10, 15, 20 times).

Through this adversarial training process, we can obtain generative models that capable of generating sequences that similar to protocol messages. To put it in another way, the learned generator function G_θ is close to the distribution function Q_m, which means our learned models have a good grasp of the protocol grammar.

3.4 Generating test cases

By now, for each of the clustering strategies, we have obtained the generative models that cover the grammar in different degree. We are ready to use the learned generative model to generate new data which are the test cases in our approach. As we mentioned in Section 2.2, by using RNN, we can learn the conditional distribution of the input sequence. As a result, to generate new data, what we need to do is first to give a give a prefix character x_1 and then query the conditional distribution to sample the next token, and repeat this process until the maximum length is met. Once you have the generative models, you can generate as many new sequences as you want.

4 FUZZING FRAMEWORK ARCHITECTURE

Based on our proposed test case generation method, we now present an intelligent and extensible industrial network protocols fuzzing framework, namely GANFuzz. As illustrated in Figure 2, GANFuzz consists of MSG(message) Capturing and Analyzing Module(MCAM), MSG Preprocessing Module(MPM), Training and MSG Generating Module(TMGM), MSG Sending Module(MSM), Logging Module(LM), and Monitoring Module(MM). These modules collaborate with each other to complete the entire fuzzing process. When running GANFuzz, it first starts the MCAM to prepare the training data, and gather address information about the industrial devices in the current industrial network. After that, MPM turns the training data into the numerical representation and deliver to TMGM. Then TMGM launches the deep learning training. After training, TMGM query the learned generative models to generate test datum which is going to be sent to the target industrial devices by the MSM. In the meantime, MM supervises the occurrence of abnormal events, while LM records all events happened. We elaborate the workflow and realization details of each component as follows.

MSG Capturing And Analyzing Module(MCAM) is responsible for gathering and analyzing network communications. Firstly, a live capturing of network traffics is performed to collect the communication data. Secondly, MCAM extracts the device addresses from the data and store as pairs of IP addresses and Ports, which will be used by the MSM during the testing phase. Thirdly, MCAM rips out the TCP/IP headers of messages to extract the protocol-specific commands and store as the text file in a format of one message per line. In addition to packet capturing, another option is to read data from local files, including pcap files and text files. For pcap files, MCAM carries out the same analyzing steps as the capturing ones. For text files, GANFuzz does not intend to do anything on them but send to MPM directly. MCAM is implemented by using the pcapy[24] library.

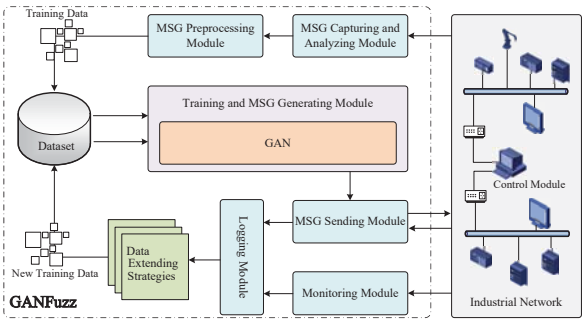

Figure 2: Main components of GANFuzz

MSG Preprocessing Module(MPM)'s task is to prepare the training data. In the first place, MPM enforces the AdvancedClustering strategy to identify the outlier data, which will be removed from the original dataset. After that, the other two clustering strategies are conducted on the processed data. For each of the three clustering strategies, GANFuzz saves the results in different directories. Lastly, MPM transforms these training data into a form of the numeric matrix by mapping each character of the character set of the messages to a unique integer, and for different clustering results, the transferring result will be kept in separate pre-defined global variables.

Training and MSG Generating Module(TMGM) is kernel module of GANFuzz, which responsible for deep learning training and test cases generating. We realize our proposed test case generation method in this module. The current prototype tool of GANFuzz provides default model settings. The generator consists of an embedding layer, followed by an LSTM layer that contains 32 hidden states. For discriminator, it comprises of an embedding layer, followed by a convolutional, max-pooling and softmax layer. TMGM trains the GAN model by using the backpropagation algorithm. Stochastic gradient descent with a batch size of 64 is utilized to update the parameter. To avoid over-fitting, we employ the dropout [25] and L2 regularization[21]. The default configuration can be used directly under the most situation. But for better learning effect, it is recommended to design specific structures of the models and use favorable training settings.

TMGM proceeds with the training process as follows. Firstly, TMGM loads all the training data in numerical representations in, then perform the training according to the training settings. Since training always takes a long time and exceptions may happen during the period, TMGM will automatically export the model parameters to checkpoint files once in a while to prevent previous work from being wasted. If there are any exceptions occurs during training, GANFuzz will recover from the nearest system checkpoint. Also, TMGM will dump the model parameters to checkpoint files according to the settings. After training, TMGM utilizes different models by reading parameters from different checkpoint files to generate test cases. All the produced test cases will be stored in text file and then read by the Message Sending Module. The TMGM is developed by using the TensorFlow[1], which is an open source software library for deep learning owned by Google.

MSG Sending Module(MSM) is responsible for sending messages to the industrial network and receiving its returns. One the one hand, each of test cases generated by TMGM is delivered to MSM. MSM wrapped those data with TCP/IP headers by using the address information obtained by MCAM. On the other hand, MSM receives the responses from industrial devices. After finish test case sending, all the communications will be conveyed to the Logging Module.

Monitoring Module(MM) monitors the industrial network for abnormalities during fuzzing. This is mainly achieved by monitoring and analyzing the interaction data. MM attempt to find out whether there are any manifestations of faults and errors. Specifically, MM looks for three kinds of exceptions in the current prototype tool, including timed-out connections, overtime return, and exception messages. If any sign of these abnormal behaviors is found, an alert will be given. To achieve better fuzzing results, manual inspection is also necessary, since some exceptions may have no direct indications in the system running traces and can only be noticed by observing its operation.

Logging Module(LM) records all the events happened during the fuzzing process. Once the first test case is sent out, LM get activated. Then LM writes down every test cases and its returns in sequence. Meanwhile, LM receives information from the MM. Any exceptions and anomalies noticed by MM will be written to the log

files. Also, clustering algorithms, such as k-means, k-medoids are used in LM to help to perform log analysis, assisting vulnerabilities finding. Also, LM picks out those messages that have triggered system errors or misbehavior for later use.

A notable characteristic of GANFuzz is that it incorporates a retraining phase. The phase is based on the logic that similar test inputs may trigger similar or even more serious problems, which help acquire more information about the bugs, contributing to bug analysis. During this period, *Data Extending Strategies* are employed to get new training data, including single-point mutation and multi-point mutation. Single-point mutation means randomly change one character in the seed message, while multi-point mutation refers to alter several tokens at the same time. GANFuzz carries out the *Data Extending Strategies* on the problematic messages selected by LM to get new training data and start a new round of training and further fuzzing over these data.

5 EXPERIMENT

In this section, we evaluate our approach by testing several implementations of Modbus-TCP protocol, including MOD_RSSIM v8.01, Modbus Slave v3.10a, and Diaslave v2.12, and ModbusPal v1.6b, which are the most popular Modbus-TCP simulators.

5.1 Modbus-TCP protocol

Modbus is an application layer protocol developed by Modicon in 1979. Nowadays, Modbus protocol has become a de facto standard communication protocol for many industrial networks. Modbus-TCP is invented to meet the application requirements of industrial Ethernet, which utilize the TCP/IP protocol stack to equip the Modbus RTU protocol with a TCP interface so that it can run on Ethernet. As illustrated in Figure 3, a Modbus-TCP message is composed of the Modbus application protocol(MBAP) header and the protocol data unit(PDU). The MBAP header is 7 bytes long and consists of four data fields, including transaction identifier field, protocol identifier field, etc., and the PDU is comprised of function code field and a data field. The maximum length of the Modbus-TCP message is up to 260 bytes.

We present the protocol format just for facilitating understanding. In our experiments, we do not make use of the prior knowledge but deduce the protocol grammar through deep learning.

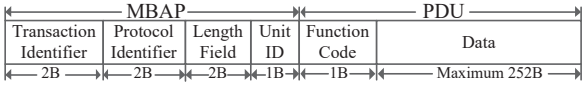

Figure 3: Message format of Modbus-TCP

5.2 Evaluation Metrics

To evaluate the result of the experiment qualitatively and quantitatively, we employ the following two metrics.

- **Test inputs reject rate(TIRR)**. TIRR means the percentage of test cases refused by the SUT. A test case is regarded as rejected if SUT treats it as illegal and returns an error message. A low TIRR means good test case quality, which shows our trained model grasp the protocol grammar well, contributing to a good fuzzing result. In converse, a high TIRR demonstrates bad test case quality, which implies that our deep-learning-based method behaves terribly, leading to unsatisfactory fuzzing result.
- **Ability of vulnerability detecting(AoVD)**. The most direct evaluation metric of fuzzing is the ability of vulnerability detecting since the ultimate purpose of fuzzing is detecting bugs [3]. In general, AoVD manifest as the number of exceptions or crashes happened in the SUT during testing. It is worth noting that exceptions mainly refer to abnormal system behaviors, but not including test case rejection. In this work, we assess AoVD more comprehensively. In addition to considering the number of exceptions, we also take the number of test cases used into account. As a result, we calculate the AoVD value as follows:

$$AoVD = \frac{100b}{n} \tag{1}$$

where b is the number of anomalies noticed, and n is the test inputs used. Its intuitive meaning is the number of exceptions found per 100 test cases. Given the same number of test cases, the more exception found indicates higher AoVD value, which means better testing result.

5.3 Training data and model training

The training data is generated by utilizing a famous python library– the Pymodbus. Pymodbus [22] is a full Modbus protocol implementation. By using Pymodbus, it is very convenient to make Modbus-TCP requests and responds by invoking the corresponding functions. We randomly call the functions in our program to generate 50000 requests strings and put in the directory of the MSG Capturing and Analyzing Module of GANFuzz.

We use the default setting of GANFuzz for model training. We first pre-train the generator for one epoch, then pre-train the discriminator for five epoch and do this for three times. Secondly, we proceed with the alternating training process, in which each time we train the generator for one epoch, we train the discriminator for five epochs. We evaluate the experiment results by repeating the alternating training process for 5, 10, 15, 20 times. The training is conducted on the NVIDIA GeForce GTX TITAN X GPU.

5.4 Experimental Results

In this subsection, we present the bugs and exceptions found during testing and evaluate the results by using the TIRR and AoVD metrics with a test case size of 50000. Meanwhile, we make a comparison of the three clustering strategies.

5.4.1 Bugs and Exceptions. The first bug was detected in the MOD_RSSIM simulator after it received some generated messages, it returns an abnormal message that "Station ID XX off-line, no response sent." Since the unit "XX" is still online, we think it may be an indication of implementation flaws. To prove our conjecture, we first sent the same command to Modbus Slave, but Modbus Slave reacts correctly. After that, we randomly modified the unit field of the problematic message but kept the other areas unchanged, then sent them to MOD_RSSIM. It turned out that MOD_RSSIM can handle these messages properly. Finally, to rule out the possibility of slave

"XX" being switched off during testing, we restart MOD_RSSIM and sent the same message to it and got the same return. Based on these analyses, we believe that there are some implementation flaws with the slave definition of the MOD_RSSIM.

A second exception was discovered after MOD_RSSIM received some test cases. The length of the sent messages do not match with their length field, but MOD_RSSIM treats them as legal ones and make response normally. This is a typical implementation error caused by the inconsistency against the protocol spec.

The third exception was found when testing ModbusPal whose graphical interface collapsed after receiving about 200 test cases. Drags or clicks on its graphical interface didn't make any difference. We had to turn it off through the task manager of the Window system. This phenomenon may result from a memory leak when dealing with the graphical user interface. A fourth exception was discovered during testing MOD_RSSIM. MOD_RSSIM raise an error of "file not found" occasionally, which occurs due to that the program does not correctly handle the file operation. The above two problems are not errors of protocol implementation, but it shows the potential of applying our method to reveal more general software bugs.

Apart from the issues mentioned above, we also found some other problems that we do not intend to list here due to page limitation.

5.4.2 TIRR and AoVD. The TIRR and AoVD value of executing the alternating training process from 5 to 20 epochs under the three different clustering strategies is illustrated in Figure 4.

It can be seen that the more times we iterate the alternating training phase, the lower the TIRR we can get, which confirms our claim that we can adjust the similarity degree of the generated messages by changing the times of training. Also, we can notice that the AdvancedClustering strategy outperform the other two strategies under every training setting. Moreover, after training for 20 times, we achieve the lowerest TIRR(43%) from the AdvancedClutering strategy and the highest TIRR(90%) from the NoClustering strategy. Last but not at least, we also noticed that the SameLengthClustering and AdvancedClutering strategy had yielded similar results.

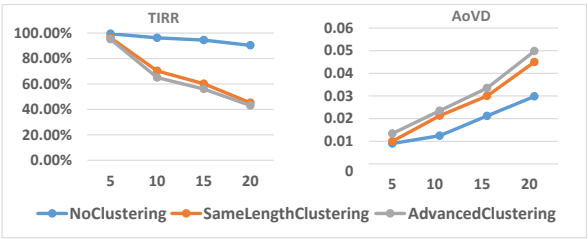

Figure 4: TIRR and AoVD values

As for the AoVD value, we can notice that as increase the times of alternating training, the AoVD value shows an upward trend. The best AoVD values come from the AdvancedClustering strategy, while the worst AoVD values are gained from the NoClustering strategy. And the SameLengthClustering and AdvancedClutering strategy show similar results. As is can be seen, we get an AoVD value of 0.05 from the AdvancedClustering, which means our approach can reveal five bugs per 10000 test cases. Even in the case of NoClustering strategy, we also achieve an AoVD value of 0.03.

Table 1: Training Time and Generating Time for 20 Epochs

	Training	Generating
NoClustering	15.7h	6.5min
SameLengthClustering	36.3h	4.6min
AdvancedClustering	17.6h	1.8min

To better compare our clustering strategies, we present the training time and the generating time for repeating the alternating training process for 20 epochs in Table 1. As we can see, the AdvancedClustering strategy outperforms the SameLengthClustering strategy significantly in both training time and generating time. For the NoClustering strategy, although it behaves worst in terms of TIRR and AoVD values, it enjoys the shortest training time.

In conclusion, although GANFuzz is still in its prototype version, it is able to reveal errors in implementations of INPs. The favorable TIRR and AoVD value we achieved also demonstrate the potential and effectiveness of our method. What's more, the diversity of the errors found indicates that the generated test cases are diverse and well-structured, which shows that the clustering strategy we present in Section 3.2 can achieve favorable code coverage and testing depth in reality. As we said earlier, the AdvancedClustering clustering strategy outperforms the other two strategies. And for SameLengthClustering, even though it is simple, but it works well. To sum up, all of these three clustering strategies are effective.

6 RELATED WORK

Miller et al. presented the first fuzzing tool for testing the UNIX programs in their early work and subsequently enhance their research by examining more UNIX utilities and services [19, 20]. As an extension of Miller's work, [8, 9] utilized fuzzing to test the robustness of numerous of Windows NT software. These early efforts mainly focused on testing software but not network protocols.

The state-of-art fuzzing method today is the model-based fuzzing that leverages the input model, which covers the grammar or structural information of the SUT's inputs, to guide the generation of test cases. The approach present in [10] use the grammar of the valid inputs of the application to enhance the white-box fuzzing. This work aims to test applications like compilers which accept highly-structured and complicated inputs. Their experiments on JavaScript interpreter embedded in Internet Explorer 7 achieves the best code coverage and depth compared to other fuzzing approaches before. Their work is contributive and exciting, and have shown great potential and effectiveness of the model-based fuzzing.

In the context of network protocol, model-based fuzzing has been applied widely. Successful fuzzers, such as Peach[7], SPIKE[14], which describe the protocol specification as XML files and templates, respectively for generation of test cases. These work promote the development of network protocol testing, but the limitation is obvious. Whenever we intend to test a new protocol, we have to interpret the protocol specs and establish some form of model manually, which is very laborious and needs a lot of time. When there

is no specification available, the mainstream approach is to reverse engineer the protocol grammar from network traces. Discoverer [6] deprived the application-level protocol message formats automatically from network traces by inferring common protocol idioms. Paolo Milani et al. [5] presented the Prospex system to extract both protocol specs and state machine. Prospex not only operates on the network traces but also records the SUT execution to figure out how the network inputs are dealt with to obtain better precision. Their work is similar to [17, 29]. In [28], the authors applied the HMM model to achieve automated protocol learning. This work infers the ϵ-machine–a form of HMM–from network traffic data, with which we are able to perform smart fuzzing and anomaly detection. These work pave the way for inferring the grammar from network traces, but none of them use deep learning techniques for this problem. Compared to these work, our approach is more straightforward and simple, but effective. All we have to do is to capture a bunch of network traces and extract the protocol-specific protocol messages, then learn a generative message model over the protocol messages. Afterward, We can use the generative model to produce test inputs.

More recently, deep learning techniques are also carried out in the field of fuzzing. Patrice et al. [11] utilized the seq2seq deep learning model to infer the grammar PDF objects, and applied the learned model to generate new PDF objects to test the PDF parser embedded in the Microsoft Edge browser. Their work shows the vast potential of using deep learning techniques for fuzz testing. Their work is close to ours, but their focus is not on network protocols. In our work, we employ the GAN model to learn the grammar of industrial network protocol.

7 CONCLUSION AND FUTURE WORK

In this paper, we propose a test case generation approach for testing implementations of industrial network protocols. Our approach learns the protocol grammar by training a generative model in a generative adversarial network to estimate the underlying distribution function of protocol messages. Having such a generative model let us able to generate well-formed test cases. To improve code coverage and testing depth, we present three clustering strategies for dividing the training data. Based on our method, we present an automatical fuzzing framework named GANFuzz, which can be applied to either public or private industrial protocol, outperforming numbers of previous fuzzing tools. Finally, we evaluate our approach by testing applications of Modbus-TCP protocol.

In the future, our work will be carried out in the following directions. Firstly, we plan to enhance our approach with the Wasserstein GAN[2]. Secondly, we intend to apply our method to stateful protocols, such as Session Initiation Protocol. Lastly, we want to develop a user-friendly interface for GANFuzz to make it easier to use.

8 ACKNOWLEDGMENTS

This work is partially supported by STCSM Project (No. 16DZ1100600 and No. 18QB1402000), SHEITC Project (No. 160602), NSFC Project (No. 61602178), and National Defense Basic Scientific Research Program of China (No. JCKY2016204B503).

REFERENCES

[1] MartÄśn Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S Corrado, Andy Davis, Jeffrey Dean, and Matthieu Devin. 2016. TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems. (2016).
[2] Martin Arjovsky, Soumith Chintala, and Léon Bottou. 2017. Wasserstein gan. *arXiv preprint arXiv:1701.07875* (2017).
[3] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *ISC*, Vol. 4176. Springer, 343–358.
[4] Tong Che, Yanran Li, Ruixiang Zhang, R Devon Hjelm, Wenjie Li, Yangqiu Song, and Yoshua Bengio. 2017. Maximum-Likelihood Augmented Discrete Generative Adversarial Networks. *arXiv preprint arXiv:1702.07983* (2017).
[5] Paolo Milani Comparetti, Gilbert Wondracek, Christopher Kruegel, and Engin Kirda. 2009. Prospex: Protocol specification extraction. In *Security and Privacy, 2009 30th IEEE Symposium on*. IEEE, 110–125.
[6] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces.. In *USENIX Security Symposium*. 1–14.
[7] Michael Eddington. 2011. Peach fuzzing platform. *Peach Fuzzer* (2011), 34.
[8] A Ghosh, Viren Shah, and Matt Schmid. 1998. An approach for analyzing the robustness of windows NT software. In *21st National Information Systems Security Conference, Crystal City, VA*, Vol. 10.
[9] Anup K Ghosh, Matthew Schmid, and Viren Shah. 1998. Testing the robustness of Windows NT software. In *Software Reliability Engineering, 1998. Proceedings. The Ninth International Symposium on*. IEEE, 231–235.
[10] Patrice Godefroid, Adam Kiezun, and Michael Y Levin. 2008. Grammar-based whitebox fuzzing. In *ACM Sigplan Notices*, Vol. 43. ACM, 206–215.
[11] Patrice Godefroid, Hila Peleg, and Rishabh Singh. 2017. Learn&Fuzz: Machine Learning for Input Fuzzing. *CoRR* abs/1701.07232 (2017). http://arxiv.org/abs/1701.07232
[12] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Aaron Ozair, Sherjil anFd Courville, and Yoshua Bengio. 2014. Generative adversarial nets. In *Advances in neural information processing systems*. 2672–2680.
[13] Sepp Hochreiter and Jürgen Schmidhuber. 1997. Long short-term memory. *Neural computation* 9, 8 (1997), 1735–1780.
[14] immunitysec. [n. d.]. SPIKE. ([n. d.]). http://www.immunitysec.com/resources/papers-presentations.html
[15] Yoon Kim. 2014. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882* (2014).
[16] Matt J Kusner and José Miguel Hernández-Lobato. 2016. GANS for Sequences of Discrete Elements with the Gumbel-softmax Distribution. *arXiv preprint arXiv:1611.04051* (2016).
[17] Zhiqiang Lin, Xuxian Jiang, Dongyan Xu, and Xiangyu Zhang. 2008. Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution.. In *NDSS*, Vol. 8. 1–15.
[18] Richard McNally, Ken Yiu, Duncan Grove, and Damien Gerhardy. 2012. *Fuzzing: the state of the art.* Technical Report. DEFENCE SCIENCE AND TECHNOLOGY ORGANISATION EDINBURGH (AUSTRALIA).
[19] Barton P Miller, Louis Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33, 12 (1990), 32–44.
[20] Barton P Miller, David Koski, Cjin Pheow Lee, Vivekananda Maganty, Ravi Murthy, Ajitkumar Natarajan, and Jeff Steidl. 1995. *Fuzz revisited: A re-examination of the reliability of UNIX utilities and services.* Technical Report. Technical Report CS-TR-1995-1268, University of Wisconsin.
[21] Andrew Y Ng. 2004. Feature selection, L 1 vs. L 2 regularization, and rotational invariance. In *Proceedings of the twenty-first international conference on Machine learning*. ACM, 78.
[22] RiptideIO. 2017. github. (mar 2017). Retrieved Jan 3, 2018 from https://github.com/riptideio/pymodbus
[23] Jason Sanders and Edward Kandrot. 2010. *CUDA by example: an introduction to general-purpose GPU programming*. Addison-Wesley Professional.
[24] CORE Security. [n. d.]. pcapy. ([n. d.]). https://github.com/CoreSecurity/pcapy
[25] Nitish Srivastava, Geoffrey E Hinton, Alex Krizhevsky, Ilya Sutskever, and Ruslan Salakhutdinov. 2014. Dropout: a simple way to prevent neural networks from overfitting. *Journal of machine learning research* 15, 1 (2014), 1929–1958.
[26] Richard S Sutton and Andrew G Barto. 1998. *Reinforcement learning: An introduction*. Vol. 1. MIT press Cambridge.
[27] Richard S Sutton, David A McAllester, Satinder P Singh, and Yishay Mansour. 2000. Policy gradient methods for reinforcement learning with function approximation. In *Advances in neural information processing systems*. 1057–1063.
[28] Sean Whalen, Matt Bishop, and James P Crutchfield. 2010. Hidden Markov Models for Automated Protocol Learning.. In *SecureComm*. Springer, 415–428.
[29] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. 2008. Automatic Network Protocol Analysis.. In *NDSS*, Vol. 8. 1–14.
[30] Lantao Yu, Weinan Zhang, Jun Wang, and Yong Yu. 2017. SeqGAN: Sequence Generative Adversarial Nets with Policy Gradient.. In *AAAI*. 2852–2858.
[31] Xiang Zhang, Junbo Zhao, and Yann LeCun. 2015. Character-level convolutional networks for text classification. In *Advances in neural information processing systems*. 649–657.