Received 4 January 2024, accepted 6 February 2024, date of publication 13 February 2024, date of current version 23 February 2024.

Digital Object Identifier 10.1109/ACCESS.2024.3365712

RESEARCH ARTICLE

SGANFuzz: A Deep Learning-Based MQTT Fuzzing Method Using Generative Adversarial Networks

ZHIQIANG WEI[1], XIJIA WEI[2], XINGHUA ZHAO[1], ZONGTANG HU[1], AND CHU XU[1]
[1]China Mobile (Suzhou) Software Technology Company Ltd., Suzhou 215004, China
[2]Department of Computer Science, University College London, WC1E 6BT London, U.K.

Corresponding author: Zhiqiang Wei (weizhiqiang51@gmail.com)

ABSTRACT As the Internet of Things (IoT) industry grows, the risk of network protocol security threats has also increased. One protocol that has come under scrutiny for its security vulnerabilities is MQTT (Message Queuing Telemetry Transport), which is widely used. To address this issue, an automated execution program called fuzz has been developed to verify the security of MQTT brokers. This program is provided with various random and unexpected input data and monitored for different responses, such as acknowledgments, crashes, failures, or memory leaks. To generate a significant number of realistic MQTT protocols, we have proposed a Generative Adversarial Networks (GAN)-based protocol fuzzer called SGANFuzz. Our experimental results show that SGANFuzz has successfully detected 6 vulnerabilities among 7 MQTT implementations, including 3 CVE bugs. Compared to the state-of-the-art fuzzing tools, SGANFuzz has proven to be the most efficient fuzzing tool in terms of vulnerability detection and has expanded the feedback coverage by receiving more unique network responses from MQTT brokers.

INDEX TERMS MQTT, fuzz test, generative adversarial networks, time-series models, transformer, vulnerability detection.

I. INTRODUCTION

As mobile network access capabilities continue to advance, the Internet of Things is becoming increasingly prevalent. To ensure the security and efficiency of communication between machines, researchers are concentrating on network layer communication protocols. Message Queuing Telemetry Transport(MQTT) is one of the well-known industrial network protocols that can be straightforward to deploy on various cloud platforms and operating systems [1].

Security concerns surrounding MQTT have been a major focus for many companies and research institutions. To investigate its security, we have utilized the fuzz test technique on the MQTT broker. In this paper, we use the fuzz test technique to investigate the security of MQTT brokers. Fuzz test is a testing method proposed by Miller in 1990 [2], which is developed to discover software weaknesses. It is designed as an automatically running program that covers random input to specific applications for possible system feedback detection. A wide range of test cases, such as program files, code, and payload packets, can be used as fuzzing data.

Researchers have developed various applications for revealing security loopholes in fuzzing network protocols. There are many generation-based methods for protocol fuzzing tasks. For example, a fuzzer for the Open Platform Communications protocol is created by Wang et al. [3], while Profinet Discovery and Configuration Protocol is the focus of a fuzzing approach proposed by Zhang et al. [4]. In industrial environments, a vulnerability detector SimaticScan is designed for Siemens SIMATIC Programmable Logic Controllers(PLC) [5]. Mutation-based fuzzers can also be utilized in fuzzing applications. AFLNet, an extension of AFL-class fuzzer, can fuzz specific protocols with additional network applications support [6], [7].

Although these research studies have contributed significantly to fuzz test development, there are still some limitations to these fuzzers. Many of them are only versatile for specific protocols, and building a protocol generation system is difficult because the tool requires knowledge of the protocol's grammar and format. Establishing protocol data

The associate editor coordinating the review of this manuscript and approving it for publication was N. Ramesh Babu.

should be time and resource-consuming as it requires manual understanding and analysis of the protocol. Motivated by these issues, we design and implement SGANFuzz, a GAN-based MQTT fuzzer that does not require knowledge of the protocol grammar and rules. SGANFuzz can generate a considerable number of formatted and valid MQTT protocol data by learning syntax from MQTT training data on its own. During the evaluation process, we test our fuzzing tool on 7 different open-source MQTT implementations and detect a total of 6 vulnerabilities, including 3 CVE bugs. Meanwhile, we demonstrate SGANFuzz's superior performance in triggering unique network responses compared to other state-of-the-art fuzzing tools. The source code and data of this study are available at https://github.com/PeterWeiJust/SGANFuzz. The main contributions of this paper are as follows.

1. A novel GAN-based fuzzing framework called SGANFuzz is proposed. We first illustrate the principle of SeqGAN, then design and implement its framework using deep learning models like GRU and transformer. After that, we implement SGANFuzz, which consists of three parts: test case generation, fuzzing tools, and log system, respectively.

2. During the test case generation process, we have found that fine-tuning the training parameters of SeqGAN leads to the best performance convergence. A comparative study is also conducted on the training performance of SeqGAN and other sequence generation architectures, such as Seq2Seq and RankGAN. To evaluate the quality and diversity of generated data, we use text generation metrics such as BLEU and Self-BLEU, along with a novel metric called N-Jaccard, which we have proposed.

3. In all, we discover 6 vulnerabilities in 7 MQTT implementations, including 3 CVE bugs. SGANFuzz can detect these bugs more efficiently than other state-of-the-art fuzzing tools. Experimental results show that SGANFuzz receives the most unique network response and highest generation data acceptance rate among other popular protocol-based fuzzing methods.

II. PRELIMINARY

In this section, we will introduce the MQTT protocol, including its structure, control packets, and message format. Additionally, we cover the principle of fuzz test, specifically protocol fuzzing.

A. MESSAGE QUEUING TELEMETRY TRANSPORT

1) STRUCTURE

MQTT is a lightweight protocol designed for devices with limited resources. When communicating through MQTT, the client will establish an authenticated connection with the broker and then send messages. These messages can be published with a specific topic, and brokers will facilitate client communications while addressing session requirements. The main structure of MQTT is shown in table. 1. MQTT protocol structure includes three parts: Fixed header, Variable header, and Payload. While the Fixed Header is the required part

TABLE 1. The structure of MQTT protocol.

Fixed Header	Bits	7	6	5	4	3	2	1	0
	Byte1	Control Packet Type				Packet Flag			
	Byte2...	Remaining Length							
Variable Header	Byte1	Variable Header content							
	Byte2...	Properties							
Payload	Byte1	Payload content							
	Byte2...	Will Properties							

TABLE 2. MQTT control packets.

Packet Type	Value	Direction	Discription
CONNECT	1	C to S	Setup connection
CONNACK	2	S to C	Acknowledge connection
PUBLISH	3	Bi-direction	Send messages to client
PUBACK	4	Bi-direction	Acknowledge publish(QoS1)
PUBREC	5	Bi-direction	Acknowledge publish(QoS2)
PUBREL	6	Bi-direction	Acknowledge pubrec(QoS2)
PUBCOMP	7	Bi-direction	Acknowledge pubrel(QoS2)
SUBSCRIBE	8	C to S	Request to subscribe topic
SUBACK	9	S to C	Acknowledge subscribe
UNSUBSCRIBE	10	C to S	Stop subscribe to a topic
UNSUBACK	11	S to C	Acknowledge unsubscribe
PINGREQ	12	C to S	Ping the server
PINGRESP	13	S to C	Acknowledge ping
DISCONNECT	14	C to S	Request to disconnect

of all packets, variable header and payload are optional. Packets such as PING and PINGRESP do not have a variable header, while others like CONNACK and PUBACK do not have a payload. The property field, which contains user information, is unique to MQTT version 5.x and only appears in CONNECT packets. Note that in our research, both the MQTT data collected from cloud systems and data generated for fuzz test are in version 3.1.1.

2) CONTROL PACKET

There are a total of 14 distinct packets present in MQTT, with each packet serving a unique communication purpose. The details surrounding the type, identifier, and intended purpose of each control packet are showcased in table. 2. These packets demonstrate a balanced distribution, with four being transmitted from the server to the client, five provided by the client, and the remaining packets sent in both directions. The MQTT protocol also offers three levels of quality of service(QoS) for message delivery, which include at most once, at least once(PUBACK), and exactly once(PUBREC, PUBREL, PUBCOMP).

3) USER ACTIONS AND MESSAGE FORMAT

The MQTT protocol uses a publish/subscribe message pattern that allows for efficient distribution of messages across multiple applications. To establish a connection with the MQTT broker, users must first send a CONNECT packet. Once the broker receives this packet and responds with a CONNACK packet, the client-server connection is established. Messages can be sent to the broker via a PUBLISH packet, which requires a user-defined topic. The broker will respond according to the number of QoS identifiers in the PUBLISH packet. In MQTT, clients connected to a

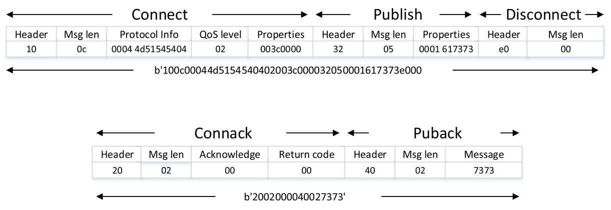


FIGURE 1. Message and response format of MQTT.

central broker can either publish messages or subscribe to topics. A subscriber can receive those published messages by sending a SUBSCRIBE packet with the topic name. Once there is no need to receive messages, an UNSUBSCRIBE packet should be sent by the subscriber for unbinding specific topics. Clients should only send a DISCONNECT packet if the connections are closed.

We present an example of the MQTT protocol format in fig. 1 for facilitating understanding. In this scenario, protocols consist of concatenated packets, each containing various data fields. The header and properties of the MQTT protocol vary from different control packets. The first sequence indicates a client connecting to the broker, publishing a message, and then disconnecting using a control packet. The feedback from the broker contains a CONNACK packet, confirming a successful connection, and a PUBACK packet, due to the QoS1 field setting in the CONNECT packet. All control packet fields are formatted in hexadecimal characters and concatenated as input sequences for the deep learning model in the experiment chapter below.

It is of great importance to acknowledge that messages obtained from cloud systems usually align with the user's MQTT actions, which can activate network responses rather than being dismissed by the network layer. For example, according to the MQTT protocol specifications, the protocol payload always begins with a CONNECT packet, not a PUBLISH or SUBSCRIBE packet. Once a client sends a DISCONNECT packet to the broker, no further user action packets will be sent until the next network connection is established. Therefore, after sufficient training, our GAN-based model will generate messages that adhere to the standard order of MQTT packets. This will be reflected in some of the fuzzing metrics we proposed later in the experiment section.

B. FUZZ TEST ABOUT NETWORK PROTOCOL

The fuzzing process phase is highly variable and depends on many factors, such as the applications being tested and the testing approach of the programmers. There are always certain basic steps that a network protocol fuzzer should follow, including the following phases.

1) IDENTIFY TARGET OBJECTIVE

To begin the protocol-based fuzz test, the first step is to identify the target. This target could be an application that implements the protocol [8]. For instance, open-source MQTT implementations like Mosquitto [9], EMQX [10], and HiveMQ [11] can serve as the target objective. These brokers act as message intermediaries that facilitate the exchange of MQTT messages between clients. They can be easily deployed using scripts on various operating systems.

2) DESIGN FUZZING STRATEGY

Modern fuzzing methods are mainly divided into two types, mutation-guided and generation-guided [12]. The generation-guided technique generates input data based on user-defined rules or knowledge of the target protocol's specifications. This approach should be appropriate when the user is well-versed in the protocol's syntax. Conversely, mutation-guided fuzzing involves using test cases from a previously collected corpus of input data. This method is effective especially when the target protocol is not well-understood or the user lacks access to source code. However, it's challenging to design a suitable fuzzing strategy for network protocols since the fuzzer should balance fuzzing coverage and efficiency.

3) GENERATE AND EXECUTE TEST CASES

This process is closely related to the previous one and involves automatically generating a high volume of test data for the fuzzing process. This includes sending the appropriate data packets to the target objective. Generating numerous cases is vital for a successful fuzzing process.

4) DISCOVER VULNERABILITY

Fuzz test is a vital technique for detecting vulnerabilities in a given target. Consequently, a reliable monitoring system is essential throughout the entire process. The monitoring system should be designed to capture data packets transmitted between clients and the broker. Once the target objective happens to hang or crash after sending particular data packets, the fuzzer must take note of such behavior and use the monitoring system to maintain a comprehensive record of the fuzzing process.

C. TIME-SERIES DEEP LEARNING MODELS

1) RECURRENT NEURAL NETWORK

RNN is a typical time-series network that deals with sequential data generation or prediction tasks [13], [14], [15], [16]. Numerous studies have proven its advantages in natural language generation, such as machine translation and speech recognition [17]. RNN is also used in future data prediction scenarios, such as indoor localization track and weather forecasting [18], [19], [20]. The unfolded base structure of RNN is displayed in fig. 2. The weight matrices U, V, and W are depicted in the image, while h represents the hidden state for every time step. Equations from 1 to 2 offer numerical definitions of the output o at time t. Notably, each matrix

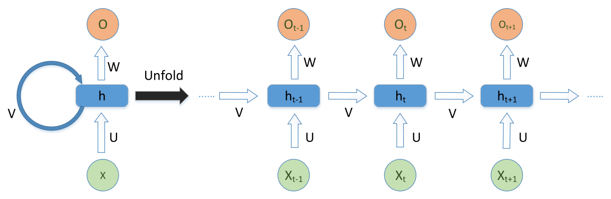


FIGURE 2. Unfolded structure of RNN.

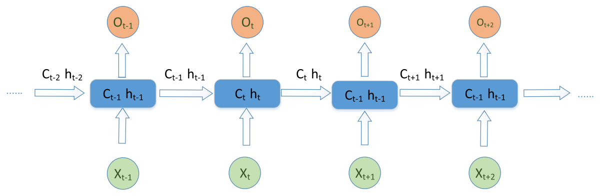


FIGURE 3. Unrolled chain of LSTM, using the same block containing current new observation and internal cell state of previous time-step.

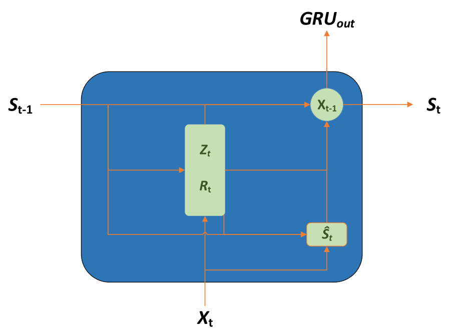


FIGURE 4. Prototype GRU cell architecture.

shares identical parameters at varying time steps.

$$h_t = f(Vh_{t-1} + Ux_t) \tag{1}$$

$$o_t = f(Wh_t) \tag{2}$$

2) VARIANTS OF RNN

LSTM is one of the RNN variants proposed to deal with the problem of vanishing gradients [21]. RNNs struggle to capture relationships between samples over long distances when processing sequential data. The LSTM handles this issue by incorporating a forget gate. The forget gate determines whether the previous "memory" should be kept or discarded in the next time step. Fig. 3 illustrates the unrolled chain of the LSTM network, which includes an additional cell state c compared to the RNN unrolled structure present in fig. 2.

$$R_t = \sigma(U_R x_t + W_R s_{t-1}) \tag{3}$$

$$Z_t = \sigma(U_Z x_t + W_Z s_{t-1}) \tag{4}$$

$$\tilde{s}_t = \phi(W(R_t \odot s_{t-1}) + Ux_t) \tag{5}$$

$$GRU_{out} = s_t = Z_t \odot s_{t-1} + (1 - Z_t) \odot \tilde{s}_t \tag{6}$$

Much like LSTM, GRU is a type of RNN that has been optimized to overcome the issue of vanishing gradient.

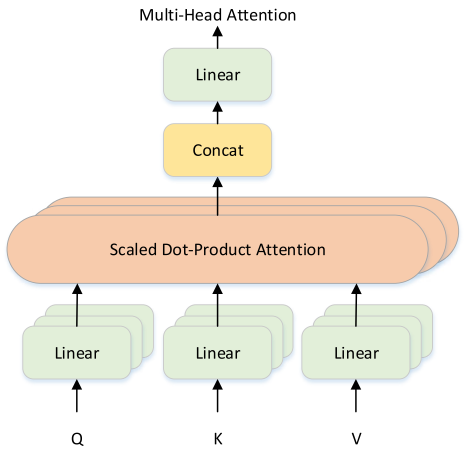


FIGURE 5. The structure of multi-head attention.

However, while the LSTM has a cell state, the GRU has a distinct internal structure that eliminates it and modifies the number of gates present [22]. From fig. 4, we find that there are only two gates inside a GRU unit: the update gate and the reset gate. Equations from 3 to 6 perform the definitions in GRU units. These gates determine what information to retain and what to include, as well as how much previous information to discard. Due to its reduced number of tensor operations, the GRU is quicker to train than the LSTM.

D. TRANSFORMER

In recent years, the Transformer has become the most successful deep learning architecture either in the field of natural language processing or computer vision, owing to its remarkable success. Unlike the recurrent neural network, the Transformer leans heavily on attention mechanisms. The multi-head attention, which is depicted in fig. 5, is the core of the Transformer model.

$$Attention(Q, K, V) = softmax(\frac{QK^T}{\sqrt{d_k}})V \tag{7}$$

The multi-head attention technique employs weight matrices for Q, K, and V. As illustrated in the accompanying image, these matrices undergo processing via several linear layers before undergoing matrix operations in the Scaled Dot-Product Attention component, as shown in equation 7. Input scaling is implemented using the value of d_k, representing the dimension of K. Multi-head attention consists of several attention layers that can run in parallel. It will concatenate these attention outputs together and apply linear layers to get the final multi-head attention output.

III. METHODOLOGY

In this part, we introduce the deep-learning models we used in our fuzzing experiment. Besides, we propose SGANFuzz, a deep learning-based method for fuzzing MQTT implementations with the objective of vulnerability detection.

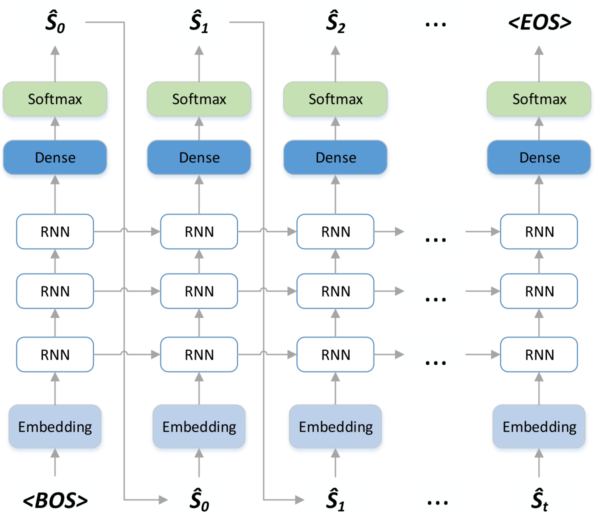


FIGURE 6. The RNN-based free-running training process.

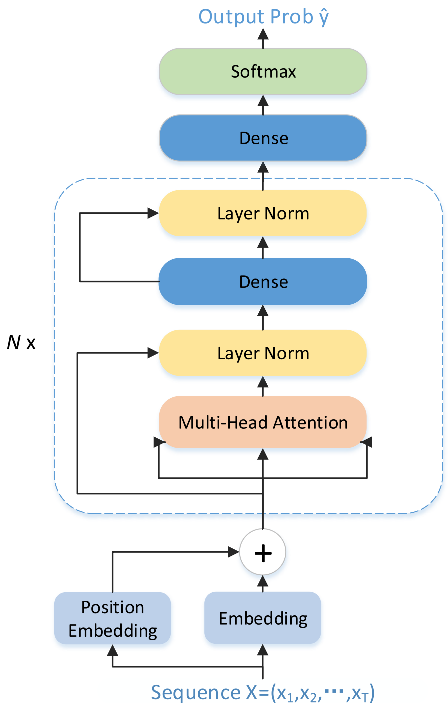


FIGURE 7. The encoder part of transformer architecture, applied for sequence classification.

A. GENERATIVE MODEL FOR SEQUENCE

The standard way of training a time-series model is using maximum likelihood estimation(MLE). First, an input sequence $\mathbf{S} = (S_1,S_2,\ldots,S_T)$ is transformed by an embedding layer into a layer of hidden states. Next, using the equation 8, a GRU generator G produces a probabilistic output y at time t by incorporating a softmax layer after the GRU gate output s_t, which is defined in equation 6.

$$p(y_t|S_1, S_2, \ldots, S_t) = softmax(s_t) \tag{8}$$

Fig. 6 showcases the training process of an RNN-based model, where the sequence $\mathbf{y} = (y_1,y_2,\ldots,y_T)$ is generated through auto-regression. Before mapping the input sequence into an embedding vector, a start flag BOS is appended at the beginning of the sequence. The expected output is the shifted original input sequence appended by an end flag EOS. The anticipated output is the original input sequence shifted by one and supplemented with an end flag EOS. Therefore, the GRU network estimates the probability of S_t at time step $t-1$ based on previous hidden states and current input S_{t-1}. During free-running, the RNN utilizes its previous output $\hat{S}_{t-1}$ as its input at the next time step t [23]. The loss of the sequence is calculated using the log-likelihood method in equation 9. In this scenario, The training process of MLE involves minimizing the negative log-likelihood, and the generator will update its weights through back-propagation through time(BPTT) at every time step using chain rules [23].

$$\mathcal{L} = -\sum_{t=1}^{T} logp(y_t = S_t|S_1, S_2, \ldots, S_{t-1}) \tag{9}$$

To address the problem of gradient vanishing in RNN architecture, we will incorporate its enhanced versions, namely LSTM and GRU, into the generator training process. These variants offer the benefit of long-term memory design, allowing them to proficiently capture long sequence feature information and generate superior, extensive texts. Furthermore, we adopt the teaching force during MLE generator training, which utilizes ground truth data as input instead of model output from a previous time step [24]. This method promotes increased stability and accelerated convergence while training recurrent neural networks.

B. DISCRIMINATIVE MODEL FOR SEQUENCE

A large number of machine learning models could be utilized for sequence classification. Traditional methods like logistic regression(LR) and support vector machine(SVM) have been proposed for binary classification [25], [26] tasks. In the last few years, deep learning-based classification methods have gradually replaced traditional ones because of their excellent performance. Models like CNN, RCNN, and LSTM, have shown impressive accuracy in classifying sequence data [27], [28]. The Transformer, an encoder-decoder architecture based on a self-attention mechanism, has become the cornerstone of many large language models like BERT and GPT-3 [29], [30]. In contrast to the generative models discussed earlier, the discriminator takes the entire sequence as an input vector and predicts the probability that the sequence is real. We present an input sequence $x_1,x_2,\ldots,x_T$ as below.

$$X_{1:T} = x_1 \oplus x_2 \oplus \ldots \oplus x_{T-1} \oplus x_T \tag{10}$$

The concatenation operator, denoted by $\oplus$ in equation 10, creates a vector incorporating the entire input sequence. In contrast to the generator, the output layer of the discriminator employs a sigmoid activation function. This enables the discriminator to provide the probability that an input sequence is real or synthetic. The objective function for optimizing the discriminative model is cross-entropy loss.

$$\hat{y} = T_{X_{1:T}} \tag{11}$$

$$\mathcal{L}(y, \hat{y}) = -ylog\hat{y} - (1-y)log(1-\hat{y}) \tag{12}$$

The framework of the transformer encoder is shown in fig. 7. The encoder mainly contains embedding layers, self-attention blocks, and linear feedforward layers. Unlike RNN, the self-attention mechanism in the transformer does not involve time steps. Therefore, in addition to regular embedding, the transformer introduces position embedding to enrich the position information of the sequence [31]. The internal structure of the Multi-Head Attention layer is the same as we have illustrated before. Layer normalization normalizes each of the inputs in the batch independently across all features, resolving the small batch sizes issue in batch normalization [32]. Equations from 11 to 12 provide numerical definitions of the transformer encoder's output and loss function. Encoder model T will produce an output label $\hat{y}$ when fed with input sequence $X_{1:T}$. $\mathcal{L}$ indicates the cross-entropy loss between ground truth label y and the predicted one $\hat{y}$.

C. CALIBRATE GAN FOR SEQUENCE DATA GENERATION

GAN is a deep learning framework proposed for generating highly realistic data. The vanilla structure of GAN consists of a generator and a discriminator [33]. The generator, denoted as G, is designed to capture the key features of the input data and generate samples that closely resemble the original data distribution. The goal is to fool the discriminator D into thinking that the generated samples are real rather than synthetic [34]. Meanwhile, the discriminator will try to identify whether its input comes from raw or synthetic data. Both of them are trained simultaneously during the adversarial training process until the generation loss converges. Optimized GANs, like DCGAN and WGAN, are proposed for enhancing unsupervised learning representations. Conditional GAN was proposed for image translation [35] and TAC-GAN was proven effective in the multimodal task of synthesizing images from text descriptions [36].

1) SeqGAN ARCHITECTURE

While GAN has achieved great success in image processing fields, generating text presents challenges due to its discrete output, making gradient updates difficult. SeqGAN is an improved architecture that combines deep learning strategy with reinforcement learning methods for sequence data generation [37]. Real data is gathered from an open device for training, while the generator produces fake data from randomly initialized vectors. After that, the discriminator classifies the combined real and fake data. During adversarial training, the generator creates a batch of samples and the discriminator calculates a numerical reward for each of them based on Monte Carlo Search [38]. The generator will then update its policy gradient [39] until the generated sentence reaches the user-defined length. The generator should be a time-series deep model as the sequence is generated step by step. The discriminator should accurately classify input data as real or synthetic. We take the MQTT protocol in fig. 1 as an example of sequence generation. To begin, we initialize the sequence with a BOS tag. The sequence is then converted into a vector and fed to the pre-trained generator. Fig. 8 illustrates the next character prediction process. The generator will predict the next character and append it at the end of the sequence. The updated sequence will then serve as the input of the generator at next time step. We repeat this process until the sequence reaches its maximum length. Finally, we append an EOS tag to indicate the end of the sequence.

Algorithm 1 SeqGAN for Fuzzing MQTT Protocol

```
Input: Pre-Generator $G_1$, weights $\alpha$; Generator $G_2$, weights
    $\beta$; Discriminator $D$, weights $\gamma$; Sequence data $P = X_{1:T}$;
    GAN training epochs $E$; Generator training gstep,
    Discriminator training dstep
 1: Pre-train $\alpha$ via $P$, using MLE method
 2: $\beta \leftarrow \alpha$
 3: Generate samples $N$ using $G_\alpha$
 4: Pre-train $\gamma$ via $N$, able to minimizing cross-entropy loss
 5: for $i = 1$ to $E$ do
 6:     for gstep do
 7:         Generate $Y_{1:T} = (y_1, y_2, \ldots, y_T)$ using $G_\alpha$,
            sequence generation process is shown in fig. 8
 8:         for $i = 1$ to $T$ do
 9:             Compute $Q(a = y_t; s = Y_{1:T-1})$ by Eq. 15
10:         end for
11:         Refresh $G_2$ parameters using policy gradients by
            Eq. 17
12:     end for
13:     for dstep do
14:         Generate batch sequence using $G_\alpha$ and combine
            these negative samples with positive data $P$
15:         Train $D_\gamma$ using above data for $p$ epochs using Eq. 16
16:     end for
17:     $\beta \leftarrow \alpha$
18: end for
19: Generate batch of payload sequence $S_b$ using trained
    SeqGAN model $sgan$
Output: Numbers of payload sequence $S_b$
```

2) TRAINING PROCESS

We have leveraged sequence generation architecture to design an optimized model capable of producing realistic MQTT protocol data. In order to guarantee the reliability and effectiveness of SeqGAN training, we have adopted a pre-training approach prior to the adversarial training loop. This accelerates the convergence of the generator and the discriminator since their weights are not initialized randomly. The detailed steps involved in SeqGAN training for MQTT

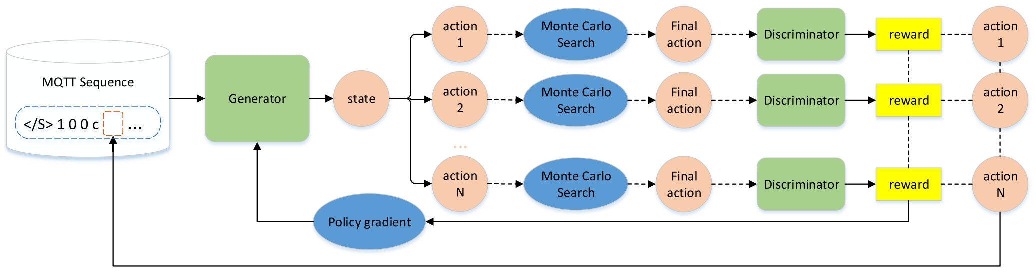


FIGURE 8. Generate the next character of MQTT sequence data by SeqGAN, using monte carlo search for actions estimation and policy gradient for weights update.

data can be found in algorithm 1. The pre-training process has been described in the sequence generation section.

$$J(\alpha) = \sum_{y1} G_\alpha(y_1|s_0) * Q_{D_\gamma}^{G_\alpha}(s_0|y_1) \quad (13)$$

During the adversarial training process, we first use a pre-trained generator for sequence generation and then compute its reward by discriminator D_γ in Eq. 13.

$$Q_{D_\gamma}^{G_\alpha}(s = Y_{1:T-1}, a = y_T) = D_\gamma(Y_{1:T}) \quad (14)$$

The function $Q_{D_\gamma}^{G_\alpha}(s, a)$ calculates the rewards earned by following policy G_α for taking action a in state s. Then, we use equation 14 to estimate the probability of whether the discriminator input is real data or not, by evaluating $D_\gamma(Y_{1:T})$.

$$Q_{D_\gamma}^{G_\alpha}(s = Y_{1:T-1}, a = y_T) = \begin{cases} \frac{1}{N}\sum_{n=1}^{N} D_\gamma(Y_{1:T}^n), \\ \\ Y_{1:T}^n \in MC(Y_{1:t}; N), t < T \\ \\ D_\gamma(Y_{1:t}), t = T \end{cases} \quad (15)$$

The discriminator only provides a reward considering a finished sequence. Since it is of great importance for us to care about the long-term reward, we should take the future outcome into consideration. We use N-times Monte Carlo Search $MC^{G_\alpha}(Y_{1:t}; N)$ to sample unknown tokens at last $T - t$ position. When the generator aims at generating a sequence $Y_{1:T}$, it will sample the t-length sequence using G_α and sample $Y_{t+1:T}$ sequence based on both G_α and current state. For a batch size data set N, we iteratively define the reward calculation starting from state $s = Y_{1:t}$ to sequence end in Eq. 15. The loss function is determined as follows.

$$\min_\gamma -\mathbb{E}_{Y\sim p_{data}}[\log D_\gamma(Y)] - \mathbb{E}_{Y\sim G_\alpha}[\log(1 - D_\gamma(Y))] \quad (16)$$

Once the Generator G_2 is trained for *gstep* times, we update its parameters α by calculating gradients of the objective function outlined in Eq. 13. This step is essential because, after the generator has undergone *gstep* training sessions, the discriminator requires re-training to remain aligned with the generator. The gradient update function is shown in Eq. 17, where lr denotes the learning rate. This parameter is utilized in various algorithms like RMSprop, SGD, and Adam.

$$\theta \leftarrow \theta + lr\nabla_\alpha J(\alpha) \quad (17)$$

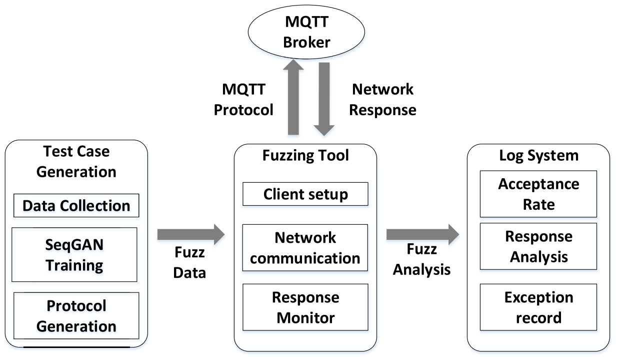


FIGURE 9. SGANFuzz architecture.

D. DESCRIPTION OF SGANFuzz ARCHITECTURE

Fig. 9 illustrates the overall architecture of SGANFuzz. The fuzzer mainly contains three parts. They are test case generation, fuzzing tool, and log system, respectively.

1) TEST CASE GENERATION

During this part, we first collect MQTT sequence data from industrial cloud systems using a specific software application, Wireshark. By connecting to the MQTT broker, the software can trace the control packets that are exchanged between clients and servers. After running Wireshark for days, a significant number of messages with the format in fig. 1 are gathered. It is worth noting that these payloads typically follow the user's actions, which can trigger various network responses instead of being rejected by the broker. For instance, a DISCONNECT packet will always appear at the end of the message sequence, and a PUBLISH packet will only occur before a CONNECT packet is dispatched from the client.

After data collection, we will apply SeqGAN architecture to train a GAN-based deep learning model capable of generating protocols. Our training approach will follow the algorithm 1, using the MQTT data we have collected. We will continue the training process until the training loss converges to a relatively small floating-point number. Given the difficulty in training GAN-based models, we will need to fine-tune the model multiple times. In the experiment section below, we will provide detailed information regarding how we intend to adjust the parameters during the fine-tuning process. Our ultimate aim is to generate realistic but fake MQTT protocol sequences.

In the testing phase of the MQTT protocol, a generator is utilized to produce test cases in the form of MQTT protocol messages. The generator has been expertly calibrated to generate various formal MQTT protocol sequences effortlessly. While most of the generated messages conform to the MQTT protocol specifications, the broker may consider certain messages informal communication protocols. The generated message adheres to a similar structure as the example protocol illustrated in fig. 1, initiating with a CONNECT packet and incorporating message delivery actions such as PUBLISH and SUBSCRIBE.

2) FUZZING TOOL

We have designed and implemented a software application that can execute test cases in addition to the protocol generation component. Before starting this application, we will deploy an MQTT service. Various popular MQTT implementations will be tested as brokers providing port 1883 for listening. After the deployment, a local client will establish a socket connection between the broker and itself. The client will then send protocol messages to the central broker as test cases. By listening to the port of the MQTT broker, the client can observe the network response from the broker and understand its current status.

All communication messages between clients and brokers will be tracked during the fuzzing process. Clients will monitor the status of brokers by recording corresponding responses after sending MQTT packets to them. The example response in fig. 1 shows that a broker returns a CONNACK confirmation and a PUBACK packet to a client. These byte data will be concatenated into a string representing a series of responses to this client request.

3) LOG SYSTEM

We have developed a comprehensive log system not only to track all communication between clients and brokers but also to check the status of the MQTT service. During the fuzz test process, clients may send requests that deviate from the protocol specifications due to a range of fuzzing strategies. However, these malformed requests will be rejected by the client and won't pass the network layer. We will determine the percentage of test messages that pass or fail the syntax check and use this data as one of the performance indicators to evaluate the effectiveness of the fuzz test systems.

Meanwhile, the clients may occasionally send harmful requests to the broker, which can result in unexpected service disruptions. To combat this, we have implemented a cutting-edge log system that can pinpoint the source of any issues that may arise, such as connection timeouts or denial of service. In extreme cases, the server may even crash as a result of unreasonable requests, such as trying to subscribe to a topic with no name or publishing a specific topic with no messages. Some of these exceptions are included in CVE, a program that identifies and defines publicly disclosed cybersecurity vulnerabilities. Furthermore, numerous protocol fuzzers have been created to detect CVE bugs.

It is worth noting that some of the responses may appear redundant. For example, a PUBACK packet that is technically distinct by mutating some values in its packet fields, may not contain novel information. To tackle this problem, we have created a response parser that can extract every field from a response message. SGANFuzz will identify any response with a distinct number or name of fields as a unique response and duly log it into the system.

IV. EXPERIMENTS

In this section, we implement the test case generation, fuzzing tool, and log system described in the previous part. We will present the performance comparison between popular fuzzing methods and evaluate our SGANFuzz for vulnerability detection.

A. DATA COLLECTION AND ENVIRONMENT BUILDING

1) DATA COLLECTION

To capture MQTT packets in the industrial cloud system, we employ the Wireshark software. It produces a file of byte-string data, representing each protocol sequence payload. The initial protocol messages are in byte-string format, which we convert into hexadecimal bytes data for processing. After that, we use the decode function in Python to convert these bytes of data to strings. In section II, we have chosen a message data sample from the dataset displayed in fig. 1. Overall, we have gathered around 30,000 MQTT sequence protocol data samples, which can be used in the forthcoming experiment section.

We first divide the initial data into a training set and a test set, using a ratio of 5:1. Then, to prepare the protocol data for input into the neural network, we establish a consistent sequence length. To facilitate training on MQTT data, we insert a BOS tag at the beginning of the sequence and an EOS tag at the end. After that, we append PAD tags to sequences that fell short of the expected length to ensure uniform sequence length. We utilize one-hot encoding to transform all protocol data into vectors, enabling us to commence with the training process at last.

2) FUZZING ENVIRONMENT BUILDING

Before executing the fuzz test, we set up the software environment and deploy the MQTT server locally. The next step is to generate a batch of MQTT sequence data and send it to MQTT brokers.

TABLE 3. MQTT brokers for testing.

Broker	Version
Mosquitto	1.6.2/2.0.2/2.0.14
HiveMQ	4.12
EMQX	5.20
VerneMQ	1.11.8
hrotti	Commit 087b33bb
mqttools	0.47
moquette	0.16

TABLE 4. SeqGAN pre-training parameters setting.

Parameters	Settings
Embedding size(both G and D)	64
Latent size(both)	64
Dropout rate(both)	0.1
Batch size(both)	32
Epochs(both)	50
Max Length(both)	50
GRU layers(G)	3
Learning rate(G)	1e-2
Learning rate(D)	1e-4
Transformer block number(D)	1
Multi-Head Attention number(D)	8

a: SOFTWARE ENVIRONMENT

Our SGANFuzz is constructed using Python 3.8, and our two designed brokers operate on Windows 10, with a RAM of 16GB. To accommodate certain MQTT implementations that require a Linux environment, we establish a virtual computer with the CentOS 7 operating system. The virtual machine contains 8 GB of RAM and 4 processor cores, which we believe should be sufficient for our testing.

b: TARGET BROKERS

MQTT implementations for vulnerability detection tests are outlined in table. 3. We conduct fuzzing on each broker, employing approximately 20000 MQTT protocol data generated by the innovative SeqGAN system. Among all these implementations, we use the commit ID from Github as its hrotti version since it lacks an official version. During the fuzzing process, our program listens to the MQTT port and sends synthetic data SeqGAN generates to the server.

B. TRAINING SeqGAN FOR MQTT DATA GENERATION

Fig. 10 shows the experimental results of training MQTT data using the SeqGAN algorithm. As outlined in algorithm 1, the algorithm utilizes three variables, *gstep*, *dstep* and *p*, during the training phase. The loss metric measures the negative log-likelihood value of the generator during the generation process. The dashed lines in the picture divide the process into pre-training and adversarial training stages. Our content generation system employs a GRU neural network as the generator and a transformer encoder model as the discriminator. Teaching force strategy will be adopted to pre-train the generator. The specific parameters used in the SeqGAN training process can be found in table. 4.

We find that the stability of SeqGAN heavily relies on the fine-tuning of specific parameters. More specifically, adjusting *gstep*, *dstep*, and *p* parameters has a noticeable effect on training loss and convergence. In fig. 10(a), both the generator and the discriminator take one step in every epoch, with the discriminator trained only once during the generation phase. As seen in fig. 10(b) and fig. 10(c), the loss curve in fig. (10a) oscillates obviously and decreases slowly. This is because the discriminator cannot get fully trained and will provide a misleading signal gradually. When *dstep* is set with 5 or *p* is set with 10, the discriminator will be trained more stable even though the negative log-likelihood loss remains high. Interestingly, as shown in fig. 10(d) and fig. 10(e), increasing the number of generation steps helps to alleviate the instability in training.

Based on the results shown in fig. 10(d), adjusting the parameters to set *gsteps* at 10, *dsteps* at 5, and *p* at 1 resulted in SeqGAN exhibiting improved performance as indicated by its lower loss value. This can be attributed to the fact that the transformer model utilized by SeqGAN is more intricate and efficient than standard time-series models, thus requiring a higher number of steps to generate high-quality samples capable of deceiving the discriminator. These findings are further supported by the loss outcomes displayed in fig. 10(e), where the generator takes 20 steps for fake data generation before the discriminator is trained for only 1 step and 1 epoch. Given these conditions, SeqGAN can learn stably and achieve superior performance.

Please note that setting a high value for the *gstep* variable is not recommended. As depicted in fig. 10(f), SeqGAN faces the problem of over-fitting when *gstep* is set to 50. Although its loss decreases quickly and stably in the initial 30 epochs of adversarial training, it rises after 40 epochs of adversarial training. As a result, during the fine-tuning process, we must let the generator and the discriminator adapt to each other's learning pace.

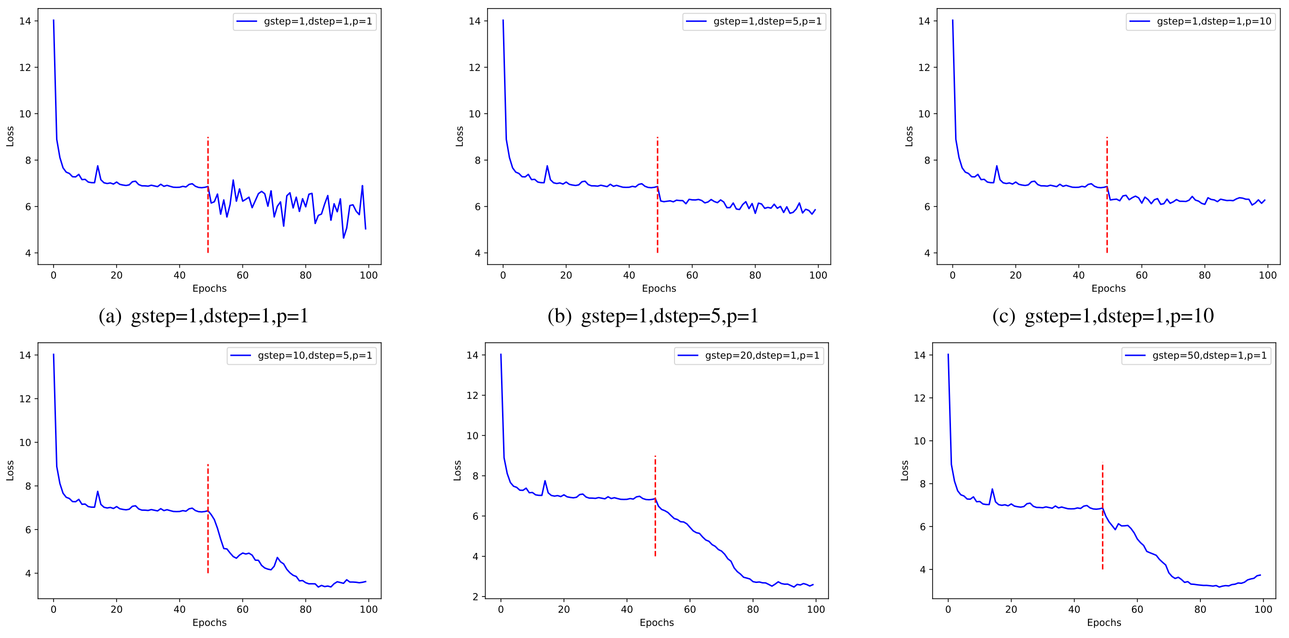
Aside from SeqGAN, we also explore RankGAN, which employs a discriminator that ranks rather than labels [40]. Meanwhile, Seq2Seq is another widely-used framework used to generate sequence data, particularly in natural language generation and speech processing. A comparative study has been conducted to evaluate the quality and diversity of generated data.

C. COMPARATIVE STUDY ON DEEP LEARNING METHODS

During our deep learning analysis, we utilize multiple sequence generation models, including SeqGAN. To measure the quality of the generated data, we employ standard natural language processing metrics such as BLEU and Self-BLEU, frequently used in machine translation tasks. Additionally, we introduce a novel metric called N-Jaccard to assess the diversity of generated MQTT data. The significance of this lies in the correlation between varied data and diverse fuzzing outcomes.

1) N-Jaccard

We propose a new metric for measuring the similarity of two text sets. Inspired by the Jaccard index, we determine the



(a) gstep=1,dstep=1,p=1 (b) gstep=1,dstep=5,p=1 (c) gstep=1,dstep=1,p=10

(d) gstep=10,dstep=5,p=1 (e) gstep=20,dstep=1,p=1 (f) gstep=50,dstep=1,p=1

FIGURE 10. The converge performance of negative log-likelihood applying different training skills. The vertical dashed lines in the middle indicate the start of adversarial training.

TABLE 5. Performance of different models in MQTT protocol generation task, evaluated on test MQTT data.

Methods	NJ2	NJ3	NJ4	NJ5	BL2	BL3	BL4	BL5	SBL2	SBL3	SBL4	SBL5
Real data	0.513	0.507	0.481	0.439	0.874	0.801	0.75	0.693	0.751	0.664	0.625	0.483
MLE	0.385	0.343	0.305	0.247	0.787	0.736	0.621	0.529	0.745	**0.698**	0.634	**0.530**
Seq2Seq	0.324	0.309	0.278	0.219	0.742	0.713	0.601	0.458	**0.759**	0.697	0.625	0.474
RankGAN	0.431	0.428	0.414	0.385	**0.803**	**0.773**	0.684	0.537	0.740	0.68	**0.641**	0.483
SeqGAN	**0.465**	**0.437**	**0.424**	0.385	0.792	0.771	**0.695**	**0.540**	0.751	0.672	0.625	0.489

similarity by calculating the ratio of the number of words that appear in both sets to the total number of words across both sets. This approach is known as the N-Jaccard similarity, represented by equation 18. The equation uses S_1 and S_2 to represent the two text sets and G_n to refer to the union set of n-grams found in both sets.

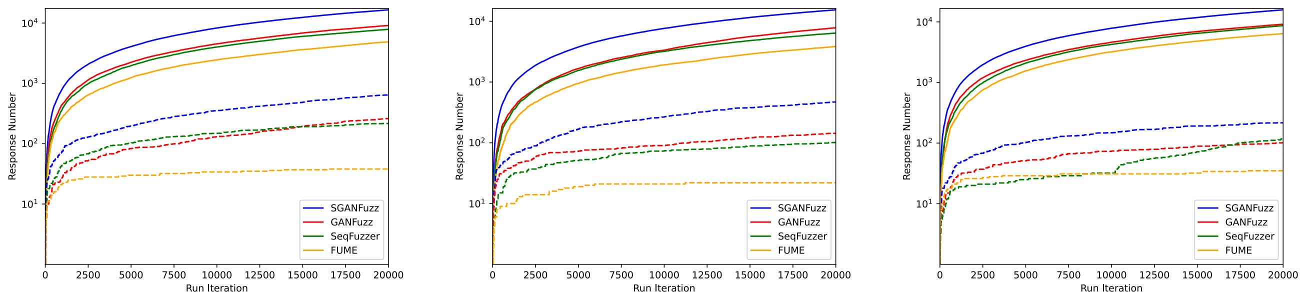
$$score_n = \frac{\sum_{g \in G_n} min\{C_n(g, S_1), C_n(g, S_2)\}}{\sum_{g \in G_n} max\{C_n(g, S_1), C_n(g, S_2)\}} \tag{18}$$

The equation involves calculating the similarity between two sets based on the frequency of n-grams, with $score_n$ being the resulting score. A low N-Jaccard score indicates that the generated text differs markedly from the actual texts in terms of distribution, suggesting a lack of diversity or quality in the generated sequence. It is worth noting that a higher N-Jaccard score is preferable, as it indicates better results.

2) BLEU AND Self-BLEU

. In natural language generation tasks, BLEU is often the metric of choice when evaluating the quality of generated sequence data. Original BLEU is designed for machine translation tasks [41] and calculates the average similarity score between the candidate data and a reference set of all sequences in the test set. In contrast to BLEU, Self-BLEU measures the diversity between generated and reference sequences. During the Self-BLEU calculation process, each generated sequence is given a BLEU score by comparing it to other generated sequences. The average of these scores provides the Self-BLEU metric, where a lower value indicates greater diversity [42].

Results of generating MQTT data using various approaches are displayed in table. 5. Our approach involves using n-gram values ranging from 2 to 5, with the MQTT test data as our reference, which we refer to as Real data. After we use different training techniques to generate sequence data batches, we compare them to the Real data. From the table, we know that with an increasing number of n, the diversity and similarity metrics of all methods show a downward trend. Moreover, based on the N-Jaccard metric, SeqGAN outperforms the other methods. RankGAN also gets high BLEU scores compared to other methods. All four methods show comparable Self-BLEU scores, but SeqGAN's value is



(a) Mosquitto response detection (b) EMQX response detection (c) HiveMQ response detection

FIGURE 11. Comparison of unique response number detection in different fuzzing methods. Dashed lines represent unique response, while solid lines indicate total response.

TABLE 6. Fuzzing methods acceptance rate.

Methods	Mosquitto	EMQX	HiveMQ
SGANFuzz	85.11%	83.27%	86.5%
FUME	53.91%	49.93%	52.14%
SeqFuzzer	72.72%	74.12%	76.9%
GANFuzz	70.11%	66.6%	72.7%

slightly lower, indicating that it produces data with greater diversity.

D. RESPONSE FEEDBACK

We build the log system to monitor the number of approved fuzzing data and the feedback response status provided by the broker. To determine the efficiency of the protocols generated that bypass the protocol format check and are sent to the broker, we have introduced a metric called acceptance rate. In this test, fuzzers with a higher acceptance rate are considered more skilled than those with a lower rate. To test the acceptance rate produced by MQTT, we use the modified edition(2.0.14) of Mosquitto. We also carry out tests on the EMQX and HiveMQ scenarios to assess this measure.

Table. 6 displays the acceptance rates of SGANFuzz and several protocol-based fuzzers, including FUME [43], SeqFuzzer [44], and GANFuzz [45]. The results demonstrate that SGANFuzz has the highest acceptance rate, generating more than 80% illegal communication protocols in all test brokers. It appears that most of the MQTT protocols generated align with the users' actions, adhering to the appropriate packet sequence. Nevertheless, a portion of these protocols are rejected due to malformed sequence data. This is due to the GAN-based model introducing some degree of randomness to the sequence data generation process, causing unexpected tokens to be included. FUME, on the other hand, has a relatively lower acceptance rate, with around half of the generated protocols being rejected during transmission. This should be reasonable because grammar-based fuzz methods generate compliant protocols, while FUME sometimes mutates protocols with unexpected changes. Our proposed model shows a higher acceptance rate than GANFuzz, outperforming it by 15%, 16.6%, and 13.8% in all test brokers. This is expected since GANFuzz uses the vanilla version of SeqGAN, which performs worse on sequence generation tasks than our model. Also, SGANFuzz has proved superior to SeqFuzzer in all three testing scenarios.

1) UNIQUE NETWORK RESPONSE

The uniqueness of a network response is determined by the values found in its control packet fields. If the response has distinct field names or numbers, it will be logged as a unique response. Throughout the fuzzing process, we monitor the broker's status by keeping track of the number of unique network responses. Fig. 11 illustrates the detection of unique responses, with dashed lines indicating distinct and solid lines representing all network responses, including redundancies. The iteration number denotes the quantity of fuzzing data generated by tested fuzzing tools.

2) MOSQUITTO DISCOVERY

Among all the fuzzing tools, SGANFuzz stands out for detecting the highest number of responses. The number of responses detected in Mosquitto broker is present in fig. 11(a). In this test, during 20000 fuzzing iterations, SGANFuzz detects approximately 16000 responses, with the number consistently increasing. While GANFuzz and SeqFuzzer also identify a noteworthy number of responses, they are not as many as SGANFuzz. FUME has the least number of responses among all the fuzzers. These tools rank in the same order when it comes to unique response detection. SGANFuzz detects over 600 unique responses, while GANFuzz and SeqFuzzer detect roughly 300 each. However, the rate of detecting unique responses is slower than overall response detection. Fuzzers like FUME even have difficulty detecting unique responses after 20000 fuzzing iterations.

3) EMQX DISCOVERY

The information in fig. 11(b) indicates that response detection ranking remains consistent between the EMQX and Mosquitto scenarios. Among the tools evaluated, SGANFuzz proves to be the most successful in identifying the greatest number of both overall and distinct responses. In contrast,

TABLE 7. Vulnerabilities found in fuzzing test.

Index	Broker	Error details	CVE
0	Mosquitto	client sends a crafted CONNACK	Yes
1	Mosquitto	too long will delay interval	Yes
2	Mosquitto	PUBLISH with topic length 0	Yes
3	hrotti	UNSUBSCRIBE with topic length 0	No
4	hrotti	client sends a crafted PUBLISH	No
5	hrotti	client sends a crafted CONNECT	No

TABLE 8. Time to find vulnerabilities. Numbers in the table mean the time of fuzzing iteration.

Index	SGANFuzz	AFLNet	BooFuzz	FUME	SeqFuzzer
0	230	23	N/A	149	643
1	3120	N/A	N/A	8345	N/A
2	154	748	N/A	255	N/A
3	43	N/A	3768	25	2341
4	32	12	N/A	17	412
5	38	5	N/A	19	564

GANFuzz and SeqFuzzer, achieve a relatively small number of response detection, with less than 100 unique responses identified compared to SGANFuzz's roughly 500.

4) HiveMQ DISCOVERY

Finally, fig. 11(c) plots the number of responses detected by these fuzzers in HiveMQ broker. The results reveal that SGANFuzz detects a lower number of unique responses, compared to Mosquitto and EMQX scenarios. However, the performance gap between the other fuzzers is not so obvious that most of them discover around 100 unique responses. It's worth mentioning that SGANFuzz demonstrates its ability to generalize as it performs the best in all test brokers.

Through analysis of three different scenarios, it has been determined that SGANFuzz outperforms other methods with respect to response detection. Additionally, when evaluating the acceptance rate of test suites, it can be concluded that the test cases produced by SGANFuzz are more realistic than those generated by alternative methods. FUME, one of the three fuzzes examined, mutates existing MQTT corpus based on specific rules, but may create malformed protocols if certain fields, such as protocol version and packet suffix, are altered. In contrast, learning-based methods like SeqFuzzer are good at generating sequences that closely align with the protocol specification due to their auto-regression architectures. But this maybe constrained in their ability to produce diverse request sequences, which ultimately restricts the number of unique responses they can identify. GAN-based fuzzer, however, can generate sequences that are both high quality and diverse, provided that the generator has been sufficiently trained. The generator is presented with the entire sequence data and generates one token at each timestep, utilizing global information to focus on the current token generation. SGANFuzz has made significant strides when compared to ordinary GANFuzz, primarily due to its effective model architecture selection and training strategy, which enables it to produce realistic protocols.

E. VULNERABILITY FINDINGS

We discover 6 vulnerabilities among MQTT implementations in table. 3 during our fuzzing test. All vulnerabilities we find here cause the immediate termination of the MQTT broker, and some of them lead to the denial of service. We listed these bugs in the table. 7 below.

1) MOSQUITTO

We find 3 vulnerabilities when fuzzing Mosquitto clients using SGANFuzz, which are indexed from 0 to 2. The first vulnerability is found in version 2.0.2 when an authenticated client connected with MQTT v5 sends a crafted CONNACK message to the broker. It has been reported to Eclipse and assigned to CVE-2021-28166. The second bug is reported when fuzzing mosquitto 1.6.2. An MQTT client connected to the broker sets a will delay interval longer than the session expiry interval. It has been assigned to CVE-2019-11778. Another Mosquitto vulnerability is related to topic length that the server crashes when the client tries to send a PUBLISH packet with zero topic length. Likewise, this vulnerability is assigned to CVE-2021-34432.

2) HROTTI

In the hrotti scenario, we uncover 3 vulnerabilities not associated with CVE. Like Mosquitto, the fourth and fifth vulnerabilities are activated by transmitting a specially crafted control packet. In particular, index 3 highlights a vulnerability that results in the hrotti broker crashing when a client tries to unsubscribe a topic with zero length.

Furthermore, we evaluate the discovery speed of these six vulnerabilities across different fuzzing engines. Our analysis encompasses SGANFuzz, BooFuzz, AFLNet, FUME, and SeqFuzzer. The time taken to identify each vulnerability is shown in table. 8. Among these fuzzing methods, BooFuzz employs a generation-based approach to generate test cases from a given input corpus, whereas AFLNet is a mutation-based MQTT fuzzer. FUME is implemented with both mutation and generation techniques. SeqFuzzer is an industrial protocol fuzzer based on the seq2seq generation framework.

We have found that when dealing with crafted packets, mutation-based fuzzers are more efficient in identifying vulnerabilities than generation-based ones. AFLNet discovers the Mosquitto crafted packet bug only 23 fuzzing iterations, while hrotti bugs are identified even faster (12 and 5 iterations). The reason for this is that modifying a control packet that matches the content of raw packet data from the input corpus is relatively easy. This makes mutation-based fuzzing method effective in detecting crafted packets vulnerabilities. However, generation-based approaches tend to generate hundreds of valid packets with similar contents. Generation-based approaches show their advantages in handling topic-length vulnerabilities since they can quickly generate PUBLISH or UNSUBSCRIBE packets. On the other hand, mutation-based approaches must successfully mutate the ''topic length'' field to zero without corrupting other contents. A shift in perspective reveals that learning-based models excel at pinpointing vulnerabilities resulting from specific modifications to protocol specifications. This is due to their familiarity with the syntax and structure of complete protocol sequences, increasing the likelihood of generating the token '0' at the topic length field through randomization. On the other hand, non-learning-based models like AFLNet and BooFuzz often encounter difficulties in identifying such scenarios. It can be time-consuming for them to change the length of the topic field, and the rules for generating the topic field usually do not permit it to be left empty. FUME detects the fourth hrotti bug in only 25 iterations, whereas AFLNet cannot detect this. SGANFuzz effectively detect both types of vulnerabilities and even triggers the second vulnerability, which other methods find challenging to detect. SGANFuzz takes 3120 fuzzing iterations to find the second bug, much faster than FUME, which takes 8345 iterations. Compared to SeqFuzzer, SGANFuzz can find more vulnerabilities due to the advantages of SeqGAN. It can generate MQTT data with greater diversity, as shown in table. 5. This diversity helps SGANFuzz change the information in different fields, such as will delay interval and topic length, without compromising the quality of the generated data.

V. RELATED WORK AND DISCUSSION

Fuzzing has been a popular technology for discovering software vulnerabilities for many years. Miller, the pioneer of the fuzz test, conducted his research on fuzzing the UNIX program in 1990 [2]. Since then, fuzzing has been significantly improved, and many applications of fuzzing network protocol have been built up. Model-based fuzzing is a well-known type of method for protocol fuzzing research. It covers the knowledge of grammar and structure of the (system under test) SUT's inputs and leverages a trained model for test case generation. Applications like Peach [4] and SPIKE [46] describe the protocol specification as XML files and templates, respectively, for test case generation. Aitel introduces a block-based approach that splits a protocol into several blocks and fuzzes the target by removing factors in this protocol [47]. FUME is a Markov model MQTT fuzzer that can generate realistic but fake data [43] by following the syntax and rules of MQTT protocol. However, these fuzzing methods require manual effort to extract the protocol grammar from network traces and reconstruct fuzzing data according to the protocol specification. Mutation-based methods, like AFL-type fuzzers, are almost the most popular class of fuzzing frameworks. The fundamental component is a coverage-based grey box fuzzer(CGF), which instruments the fuzz target by injecting some instructions into the source code at compile time. Whenever a new path is reached in the code, the instrumented target will inform AFL during fuzzing time.

Deep learning technology has been applied in fuzz tests to automate the reconstruction engineering process. For instance, Nichols' et al. GANFuzz framework leverages GAN structure to fuzz Modbus-TCP protocol [48], while Chockalingam's deep learning-based fuzzer detects abnormalities in the CAN protocol [49]. Additionally, Zhao's SeqFuzzer utilizes a seq2seq framework to fuzz the EtherCAT Ethernet protocol [44]. These methods employ the RNN architecture for protocol data generation, making them user-friendly as they do not require knowledge of the protocol specification or the test broker. However, optimal results can be challenging to achieve due to the heavy reliance on input corpus data and the model training strategy, despite the time and labor-saving benefits of these fuzzers.

SGANFuzz has many advantages over other protocol fuzzers. Firstly, it uses an optimized SeqGAN framework that outperforms the GAN framework used in GANFuzz for sequence data generation. Its fuzzing data is highly diverse, making it more effective at identifying unique feedback from the broker than other deep learning-based methods. Secondly, SGANFuzz features a response feedback system that previous protocol-based fuzzing tools lacked. This system enables the investigation of the status of the broker and the monitoring of feedback coverage. Additionally, SGANFuzz can also be used for fuzz tests on other real-world network protocols, provided sufficient sequence data is available for training.

However, the disadvantages of SGANFuzz cannot be ignored. As a black-box fuzzer, SGANFuzz has no access to the source code of MQTT brokers, which means it can only estimate the feedback coverage. In contrast to SGANFuzz, grey-box fuzzers like AFLs can monitor their source code coverage directly using instrumentation. Compared to other deep learning-based fuzzing frameworks, the SeqGAN architecture in SGANFuzz is also challenging to fine-tune, often requiring a significant amount of time to achieve optimal loss convergence. Also, SGANFuzz performs poorly when generating long sequence data. This problem can be solved by dividing the MQTT sequence into several subsequences according to the control packets, generating them separately, and finally concatenating them.

VI. CONCLUSION

To enhance the efficiency of fuzz testing, we have developed SGANFuzz, a deep learning-based framework that can independently generate MQTT protocols with great accuracy. With SeqGAN, a powerful sequence generation model, SGANFuzz can be expertly trained on MQTT data to optimize the efficiency of fuzzing testing. Throughout rigorous testing, we have uncovered a total of 7 vulnerabilities in 7 open-source MQTT implementations, including 3 CVE bugs. The experimental results of discovery speed indicate that SGANFuzz surpasses other state-of-the-art fuzzers by detecting MQTT vulnerabilities with more precision and speed. Also, SGANFuzz is proven to be more powerful than existing methods of MQTT fuzz testing thanks to its capability to identify a more significant number of unique network responses.

REFERENCES

[1] D. Soni and A. Makwana, "A survey on MQTT: A protocol of Internet of Things (IoT)," in *Proc. Int. Conf. Telecommun. Power Anal. Comput. Technol. (ICTPACT)*, vol. 20, 2017, pp. 173–177.

[2] P. Mell and T. Grance, "Use of the common vulnerabilities and exposures (CVE) vulnerability naming scheme," NIST Special Publication, 2002, vol. 800, p. 51.

[3] T. Wang, Q. Xiong, H. Gao, Y. Peng, Z. Dai, and S. Yi, "Design and implementation of fuzzing technology for OPC protocol," in *Proc. 9th Int. Conf. Intell. Inf. Hiding Multimedia Signal Process.*, Oct. 2013, pp. 424–428.

[4] D. Zhang, J. Wang, and H. Zhang, "Peach improvement on profinet-DCP for industrial control system vulnerability detection," in *Proc. 2nd Int. Conf. Electr., Comput. Eng. Electron.*, Atlantis Press, 2015, pp. 1622–1627.

[5] R. Antrobus, S. Frey, B. Green, and A. Rashid, "SimaticScan: Towards a specialised vulnerability scanner for industrial control systems," in *Proc. 4th Int. Symp. ICS SCADA Cyber Secur. Res.*, vol. 4, 2016, pp. 11–18.

[6] V.-T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox fuzzer for network protocols," in *Proc. IEEE 13th Int. Conf. Softw. Test., Validation Verification (ICST)*, Oct. 2020, pp. 460–465.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2016, pp. 1032–1043.

[8] S. Hernández Ramos, M. T. Villalba, and R. Lacuesta, "MQTT security: A novel fuzzing approach," *Wireless Commun. Mobile Comput.*, vol. 2018, pp. 1–11, Jan. 2018.

[9] R. A. Light, "Mosquitto: Server and client implementation of the MQTT protocol," *J. Open Source Softw.*, vol. 2, no. 13, p. 265, May 2017.

[10] S. Gao and Z. Shi, "Industrial internet cloud platform system based on emqx," in *Proc. Chinese Intell. Syst. Conf.* Springer, 2022, pp. 183–192.

[11] F. Antonielli, "Development and comparison of MQTT distributed algorithms for HiveMQ," Tech. Rep., 2021.

[12] C. Chen, B. Cui, J. Ma, R. Wu, J. Guo, and W. Liu, "A systematic review of fuzzing techniques," *Comput. Secur.*, vol. 75, pp. 118–137, Jun. 2018.

[13] X. Wei, Z. Wei, and V. Radu, "Sensor-fusion for smartphone location tracking using hybrid multimodal deep neural networks," *Sensors*, vol. 21, no. 22, p. 7488, Nov. 2021.

[14] X. Wei and V. Radu, "Leveraging transfer learning for robust multimodal positioning systems using smartphone multi-sensor data," in *Proc. IEEE 12th Int. Conf. Indoor Positioning Indoor Navigat. (IPIN)*, Sep. 2022, pp. 1–8.

[15] X. Wei, T. Olugbade, F. Shi, S. Wu, A. William, N. Gold, Y. Cho, K. Chetty, and N. Bianchi-Berthouze, "Leveraging WiFi sensing toward automatic recognition of pain behaviors," in *Proc. 11th Int. Conf. Affect. Comput. Intell. Interact. Workshops Demos (ACIIW)*, Sep. 2023, pp. 1–8.

[16] T. Olugbade, L. Lin, A. Sansoni, N. Warawita, Y. Gan, X. Wei, B. Petreca, G. Boccignone, D. Atkinson, Y. Cho, S. Baurley, and N. Bianchi-Berthouze, "FabricTouch: A multimodal fabric assessment touch gesture dataset to slow down fast fashion," in *Proc. 11th Int. Conf. Affect. Comput. Intell. Interact. (ACII)*, Sep. 2023, pp. 1–8.

[17] G. Saon, Z. Tüske, D. Bolanos, and B. Kingsbury, "Advancing RNN transducer technology for speech recognition," in *Proc. IEEE Int. Conf. Acoust., Speech Signal Process. (ICASSP)*, Jun. 2021, pp. 5654–5658.

[18] X. Wei and V. Radu, "Calibrating recurrent neural networks on smartphone inertial sensors for location tracking," in *Proc. Int. Conf. Indoor Positioning Indoor Navigat. (IPIN)*, Sep. 2019, pp. 1–8.

[19] W. Fang, Y. Chen, and Q. Xue, "Survey on research of RNN-based spatio-temporal sequence prediction algorithms," *J. Big Data*, vol. 3, no. 3, pp. 97–110, 2021.

[20] X. Wei, Z. Wei, and V. Radu, "MM-loc: Cross-sensor indoor smartphone location tracking using multimodal deep neural networks," in *Proc. Int. Conf. Indoor Positioning Indoor Navigat. (IPIN)*, Nov. 2021, pp. 1–8.

[21] R. C. Staudemeyer and E. Rothstein Morris, "Understanding LSTM—A tutorial into long short-term memory recurrent neural networks," 2019, *arXiv:1909.09586*.

[22] R. Dey and F. M. Salem, "Gate-variants of gated recurrent unit (GRU) neural networks," in *Proc. IEEE 60th Int. Midwest Symp. Circuits Syst. (MWSCAS)*, 2017, pp. 1597–1600.

[23] A. Sherstinsky, "Fundamentals of recurrent neural network (RNN) and long short-term memory (LSTM) network," *Phys. D, Nonlinear Phenomena*, vol. 404, Mar. 2020, Art. no. 132306.

[24] S. Rafi and R. Das, "RNN encoder and decoder with teacher forcing attention mechanism for abstractive summarization," in *Proc. IEEE 18th India Council Int. Conf. (INDICON)*, Dec. 2021, pp. 1–7.

[25] X. Zou, Y. Hu, Z. Tian, and K. Shen, "Logistic regression model optimization and case analysis," in *Proc. IEEE 7th Int. Conf. Comput. Sci. Netw. Technol. (ICCSNT)*, Oct. 2019, pp. 135–139.

[26] Q. Li, H. Peng, J. Li, C. Xia, R. Yang, L. Sun, P. S. Yu, and L. He, "A survey on text classification: From traditional to deep learning," *ACM Trans. Intell. Syst. Technol.*, vol. 13, no. 2, pp. 1–41, Apr. 2022.

[27] Y. Luan and S. Lin, "Research on text classification based on CNN and LSTM," in *Proc. IEEE Int. Conf. Artif. Intell. Comput. Appl. (ICAICA)*, Mar. 2019, pp. 352–355.

[28] S. Lai, L. Xu, K. Liu, and J. Zhao, "Recurrent convolutional neural networks for text classification," in *Proc. AAAI Conf. Artif. Intell.*, Feb. 2015, vol. 29, no. 1, pp. 1–7.

[29] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," 2018, *arXiv:1810.04805*.

[30] T. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, and A. Askell, "Language models are few-shot learners," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 33, 2020, pp. 1877–1901.

[31] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[32] J. Lei Ba, J. Ryan Kiros, and G. E. Hinton, "Layer normalization," 2016, *arXiv:1607.06450*.

[33] A. Creswell, T. White, V. Dumoulin, K. Arulkumaran, B. Sengupta, and A. A. Bharath, "Generative adversarial networks: An overview," *IEEE Signal Process. Mag.*, vol. 35, no. 1, pp. 53–65, Jan. 2018.

[34] I. Goodfellow, J. Pouget-Abadie, M. Mirza, B. Xu, D. Warde-Farley, S. Ozair, A. Courville, and Y. Bengio, "Generative adversarial nets," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 27, 2014, pp. 1–9.

[35] P. Isola, J.-Y. Zhu, T. Zhou, and A. A. Efros, "Image-to-image translation with conditional adversarial networks," in *Proc. CVPR*, 2017, pp. 1125–1134.

[36] A. Dash, J. C. B. Gamboa, S. Ahmed, M. Liwicki, and M. Z. Afzal, "TAC-GAN-text conditioned auxiliary classifier generative adversarial network," 2017, *arXiv:1703.06412*.

[37] L. Yu, W. Zhang, J. Wang, and Y. Yu, "SeqGAN: Sequence generative adversarial nets with policy gradient," in *Proc. AAAI Conf. Artif. Intell.*, Feb. 2017, vol. 31, no. 1, pp. 1–7.

[38] G. M. J.-B. C. Chaslot, "Monte-carlo tree search," Tech. Rep., 2010.

[39] R. S. Sutton, D. McAllester, S. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 12, 1999, pp. 1–7.

[40] K. Lin, D. Li, X. He, Z. Zhang, and M.-T. Sun, "Adversarial ranking for language generation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 30, 2017, pp. 1–11.

[41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, "BLEU: A method for automatic evaluation of machine translation," in *Proc. 40th Annu. Meeting Assoc. Comput. Linguistics*, 2002, pp. 311–318.

[42] E. Montahaei, D. Alihosseini, and M. Soleymani Baghshah, "Jointly measuring diversity and quality in text generation models," 2019, *arXiv:1904.03971*.

[43] B. Pearson, Y. Zhang, C. Zou, and X. Fu, "FUME: Fuzzing message queuing telemetry transport brokers," in *Proc. IEEE INFOCOM Conf. Comput. Commun.*, May 2022, pp. 1699–1708.

[44] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *Proc. 12th IEEE Conf. Softw. Test., Validation Verification (ICST)*, 2019, pp. 59–67.

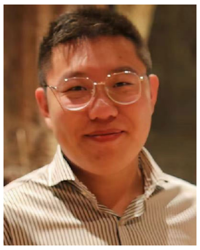
[45] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "GANFuzz: A GAN-based industrial network protocol fuzzing framework," in *Proc. 15th ACM Int. Conf. Comput. Frontiers*, 2018, pp. 138–145.

[46] S. Bradshaw, "An introduction to fuzzing: Using fuzzers (SPIKE) to find vulnerabilities," Tech. Rep., 2010.

[47] D. Aitel, "The advantages of block-based protocol analysis for security testing, 2002," Tech. Rep., 2002.

[48] N. Nichols, M. Raugas, R. Jasper, and N. Hilliard, "Faster fuzzing: Reinitialization with deep neural models," 2017, *arXiv:1711.02807*.
[49] V. Chockalingam, I. Larson, D. Lin, and S. Nofzinger, "Detecting attacks on the can protocol with machine learning," *Annu. EECS*, vol. 558, no. 7, pp. 5–7, 2016.

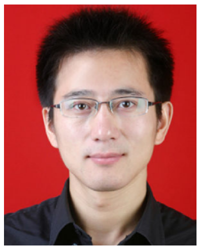
ZHIQIANG WEI received the master's degree in artificial intelligence from The University of Edinburgh, in 2018. He is currently a Researcher with Cmsoft company. Prior to joining Cmsoft, he was a Software Engineer with Microsoft Suzhou, where he was responsible for the development and optimization of Bing map search. His current research interests include using generative adversarial networks and large language models (LLM) to generate fuzzing data for the MQTT protocol.



XIJIA WEI received the B.Eng. degree in electronics and electrical engineering and the M.Sc. degree in artificial intelligence from The University of Edinburgh. He is currently pursuing the Ph.D. degree with University College London. He is investigating self-supervised multimodal deep learning architecture to allow models to understand communicative multisensory representations. Meanwhile, he is also a Visiting Scholar with Bell Labs. His research interest includes multimodal ubiquitous computing.

XINGHUA ZHAO was born in Baoding, China, in 1990. He received the Ph.D. degree in quantum science instruments from the College of Instrumentation Science and Optoelectronics Engineering, Beihang University, Beijing, China, in 2022. Currently, he is a Senior Researcher with China Mobile. His research interests include DPU, RDMA, data center networks, message middleware, and control systems.

ZONGTANG HU is currently a Senior Engineer with Cmsoft company. He has focused on message middleware development and research for eight years. He has contributed many features and docs for many open-source software communities, such as Apache RocketMQ, Apache Pulsar, OpenEuler, Nacos, and SOFAJRaft. He is also a Maintainer and a Committer for these open-source software communities.



CHU XU is a Senior Engineer and the Deputy General Manager of the Platform Product Department, China Mobile Cloud. He leads the research and development of more than 20 big data, data security, and middleware products, for China Mobile Cloud. His main research interests include big data platforms and applications, data security, and middleware.