CGFuzzer: A Fuzzing Approach Based on Coverage-Guided Generative Adversarial Networks for Industrial IoT Protocols

Zhenhua Yu, *Member, IEEE*, Haolu Wang, Dan Wang, *Member, IEEE*, Zhiwu Li, *Fellow, IEEE*, and Houbing Song, *Senior Member, IEEE*

***Abstract*—With the widespread application of the Industrial Internet of Things (IIoT), industrial control systems (ICSs) greatly improve industrial productivity, efficiency, and product quality. However, IIoT protocols as the bridge of different parts of ICSs are vulnerable to be attacked due to their vulnerabilities. To reduce cyberattack threats, we need to find the vulnerabilities of IIoT protocols by using efficient vulnerability mining methods, such as fuzzing. Fuzzing is often used to mine vulnerabilities for IIoT protocols. However, the traditional fuzzing methods for IIoT protocols have a low passing rate and low code coverage. To solve these problems, we propose a generative adversarial network (GAN), here referred to as coverage-guided GANs (CovGAN), which aims to generate test cases with a high passing rate and code coverage by learning IIoT protocol specifications. Based on the CovGAN, we construct a fuzzing framework (CGFuzzer) for IIoT protocols. Finally, we design a protocol simulator to verify the CovGAN performance. Experimental results show that the proposed methodology outperforms approximately 5%, 7%, and 39% of the passing rate of GANFuzz, SeqFuzzer, and Peach, respectively. In addition, CGFuzzer has a significant improvement in code coverage, which is about 17%, 24%, and 31% higher than GANFuzz, SeqFuzzer, and Peach, respectively.**

***Index Terms*—Coverage guided, fuzzing, generative adversarial networks (GANs), industry control protocol, vulnerability mining.**

I. Introduction

INDUSTRIAL control systems (ICSs) are automatic control systems composed of computer equipment and industrial control components, which are widely applied to water treatment, chemical industry, manufacturing, and other critical fields [1]. With the proposal of "Internet+," the Industrial Internet of Things (IIoT) [2] and ICSs are tightly integrated. While industrial productivity and efficiency are being improved by the IIoT, the ICSs maybe expose inherent vulnerabilities to attackers, which seriously threatens their security [3]. The Stuxnet attacked Iran's Bushehr nuclear power plants in 2010 by using an existing vulnerability to tamper with a programmable logic controller, eventually causing physical damage to centrifuges [4]. On August 3rd, 2018, as a new equipment was installed without prior quarantine and offline security checks, the ICSs of Taiwan Semiconductor Manufacturing Company were infected with the Wannacry virus, which affected three factories [5]. These attacks were executed through potential vulnerabilities in protocols, PLCs, remote control units, or other devices, posing a serious threat to the economy and society.

IIoT protocols are developed to interconnect various parts of ICSs, which are usually implemented with security vulnerabilities and flaws [6]. Consequently, they have become the key target of cyberattacks [7]. According to different industry fields, there exist different IIoT protocols, such as IEC60870-5-101/104 [8], distributed network protocol version 3 (DNP3) [9], Modbus [10], OPC, and Profinet [11]. Due to the transmission with clear messages, IEC60870-5-101/104 has risks of being eavesdropped and sniffed. DNP3 can be attacked by crafted frames. Modbus, OPC, and Profinet also face threats from spoofing, replay, and Denial-of-Service (DoS) attacks [12]. As one of the most important protocols in the IIoT, DNP3 is widely applied to smart grids and the water industry [13]. If there are vulnerabilities in the DNP3 protocol, they will cause significant damages to the systems. Therefore, the study of DNP3 security is particularly important [14].

DNP3 includes an application layer, a transport layer, and a data link layer [15]–[17]. It provides functions, such as error detection, remote key update, and security statistics to avoid vulnerabilities [14] caused by unexpected data, authorization failure, response timeout, etc. However, there are still some potential vulnerabilities in DNP3. The extension and update of DNP3 functions in IIoT networks lead to more vulnerabilities, which pose a major threat to ICSs. The threat model of DNP3 is shown in Fig. 1. DNP3 does not have security authentication, which makes DNP3 easy to be attacked during communication. Attackers can find its potential vulnerabilities by eavesdropping, man in the middle, spoofing, or replay attacks [14]. Through using exploit scripts of vulnerability, attackers can implant viruses to devices of

Manuscript received 14 March 2022; revised 16 April 2022 and 17 May 2022; accepted 5 June 2022. Date of publication 16 June 2022; date of current version 24 October 2022. This work was supported in part by the National Natural Science Foundation of China under Grant 61873277, and in part by the National Science Foundation under Grant 2150213. *(Corresponding authors: Zhiwu Li; Houbing Song.)*

Zhenhua Yu, Haolu Wang, and Dan Wang are with the College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an 710054, China (e-mail: zhenhuayu@xust.edu.cn; 19308208016@stu.xust.edu.cn; jojowd@xust.edu.cn).

Zhiwu Li is with the Institute of Systems Engineering, Macau University of Science and Technology, Macau, China (e-mail: zwli@must.edu.mo).

Houbing Song is with the Security and Optimization for Networked Globe Laboratory, Embry-Riddle Aeronautical University, Daytona Beach, FL 32114 USA (e-mail: h.song@ieee.org).

Digital Object Identifier 10.1109/JIOT.2022.3183952

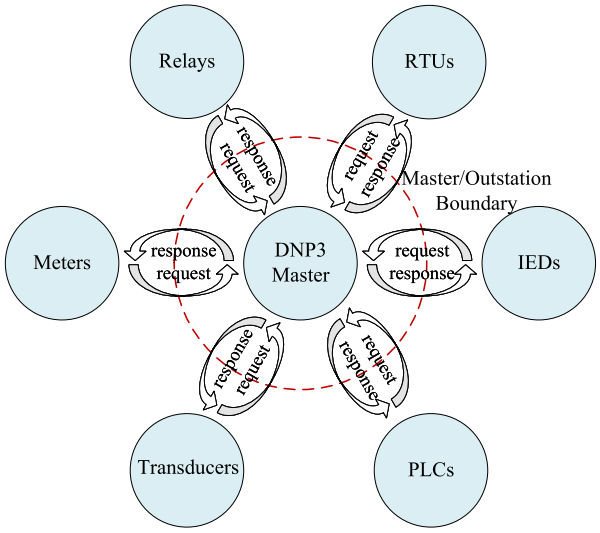

Fig. 1. Threat model of DNP3.

DNP3-based industrial control networks. Once the DNP3 master is implanted a virus, other devices may also be implanted, such as RTUs, IEDs, PLCs, etc. In addition to using an efficient access control to improve protocol security [18], [19], it is necessary to employ vulnerability mining techniques to find out potential vulnerabilities in DNP3 for preventing ICSs from being attacked. Vulnerability mining can be divided into static [20] and dynamic methods. As a dynamic method, fuzzing plays a vital role in mining vulnerabilities [21]. It has been employed to discover vulnerabilities for network protocols, which is implemented by sending many nonstandard data to industrial control devices [22]. By causing device communication errors in fuzzing, it is possible to determine whether the protocols are vulnerable through analyzing those errors.

The existing fuzzing methods for IIoT protocols have some problems, such as low passing rate and code coverage of test cases, and high redundancy of test cases. It is necessary to present a fuzzing method that considers the characteristics of DNP3 to efficiently discover the potential vulnerabilities in DNP3. In this article, we propose a new fuzzing framework for DNP3 that improves passing rate and code coverage, where a coverage-guided generative adversarial network is developed to learn the features of the protocol header to improve the passing rate, and a multiarmed bandits algorithm is employed to improve code coverage. The main contributions of this article are summarized as follows.

1) A coverage-guided generative adversarial network (CovGAN) for DNP3 is presented, which combines upper confidence bound applied to tree [23] (UCT) with sequence generative adversarial networks [24] (SeqGAN) for improving passing rate and code coverage.
2) We propose a fuzzing framework CovGANs fuzzer (CGFuzzer) based on the CovGAN to improve the fuzzing automation of DNP3. To evaluate its effectiveness, we develop some performance metrics, such as the number of triggering exceptions, code coverage, and passing rate of test cases.
3) A DNP3 simulator is constructed to compare the performance of CGFuzzer with Peach, GANFuzz, [25] and SeqFuzzer [26]. The simulation results show that CGFuzzer has a higher number of triggering exceptions, code coverage, and passing rate of test cases than other methods.

The remainder of this article is organized as follows. The related works are discussed in Section II. Section III details the architecture of CovGAN. In Section IV, we present the fuzzing framework of DNP3 based on CovGAN. Section V introduces metrics for evaluating results against other methods. Section VI reaches a conclusion and makes a prospect of the research.

II. Related Works

Fuzzing is a common and effective vulnerability analysis technique, which inputs random and unexpected data into a program to detect its vulnerabilities [27]. The methods of traditional fuzzing can be divided into generation-based and mutation-based approaches.

A. Generation-Based Approaches

Generation-based fuzzing generates the test cases that meet the basic syntactic and semantic of a target system through designing models (model-based) or grammar rules (syntax-based). Peach [28] and Spike [29] are typical model-based fuzzing tools, where the former also has the capability to mutate test cases. Peach specifies the types and data by writing a state model and a data model that is customized with input format and generates test cases in combining with a mutation strategy. Unlike Peach writing a configuration file to implement fuzzing, Spike needs to use a programming interface to constrain the format of inputs. These two tools generate test cases with high legitimacy and have achieved good results in practice. However, there are some limitations for fuzzing with unknown data models, which require reverse parsing to get the data models.

In addition to the model-based tools described above, there are also some syntax-based approaches. LangFuzz [30] learns code fragments from a test set based on syntax and reassembles fragments to generate new test cases. With respect to test cases selection, the assumption is that test cases obtained by problematic test sets are more likely to trigger program exceptions than those obtained by random collection. Spandan *et al.* [31] used context-independent language grammars to generate parsing trees. Test cases are generated by using genetic evolutionary algorithms for the reorganization of code fragments. A grammar tree is introduced in Superion [32], which mutates and filters its samples with the help of American fuzzy lop (AFL) [33]. After applying tree-based mutation in using another tool for language recognition (ANTLR), Superion improves the code coverage and bug-finding capability over AFL and jsfunfuzz [34].

B. Mutation-Based Approaches

AFL uses an evolutionary algorithm to generate test cases and evaluates their quality through the feedback of the execution path. Then, AFL will preserve and mutate test cases

that trigger new paths. However, AFL suffers from the problem of blindly selecting the location and operation of mutation. The fuzzing problem can be regarded as a Markov model, such as AFLFast [35]. It uses a specific strategy to guide AFL to prefer low-frequency paths for mutation, which can explore more paths in the same test time. AFLGo [36] plans to add a simulated annealing algorithm for assigning higher energy to test cases that can approximate a specific target location and give priority to high-energy test cases. AFLNET [37] models the state transfer of protocols by using state machines to improve the capabilities of vulnerability mining with different protocol states. It improves the efficiency of test cases generation by generating state-specific test cases and finds states that can cause vulnerabilities through analyzing the server's response codes. AFLNET has discovered two new CVEs that are classified as critical (CVSS score CRITICAL 9.8). EPF [38] is also a coverage-guided fuzzing framework, and it applies population-based simulated annealing to heuristically schedule packet types for increasing fuzzing effectiveness.

To address the problem of blindly obtaining mutation location, BuzzFuzz [39] applies dynamic taint analysis techniques to automatically locate fields in test cases that affect program vulnerabilities. Then, it only mutates the data of the located fields and retains the content of other fields. BuzzFuzz completes the mutation while passing the syntax checking, which improves the efficiency of hitting target programs in vulnerability mining. Bigfuzz [40] builds the execution logic of applications deployed in big data computing frameworks as directed acyclic graphs and incorporates schema-aware data mutation operators based on their in-depth study of data-intensive scalable computing (DISC) application error types. In contrast to the random mutation, it can reduce fuzzing time, improve code coverage, and achieve five times improvement in detecting application errors. RESTler [41] generates test cases by inferring the dependencies of the request types and analyzing the dynamic feedback values of the response messages. It has found 28 bugs in GitLab and several bugs in Azure and Office 365. Considering different types of attacks, Ori [42] divides the fuzzing into an attack mode and structural mutation to verify the security of scalable service-oriented middleware over IP (SOME/IP) applications.

Although the above works have promoted the testing technology development of IIoT protocols, most of them are extended based on the traditional fuzzing framework and require a tedious design of test cases that results in increasing test costs and being more prone to error.

C. Artificial Intelligence-Based Approaches

As deep learning algorithms are applied to various fields, many researchers use existing techniques to improve the efficiency of vulnerability mining tools. Patrice *et al.* [43] proposed a sequence-to-sequence model to learn the input grammar of PDF objects to help produce fuzzing data for the PDF parser. She *et al.* [44] employed a deep neural network to guide the generation of test cases. Their Neuzz uses convolutional neural networks (CNNs) to approximate the actual logic in a target program and guides the generation by finding the gradient of the network to achieve a higher branch coverage for the target program. Comparing with AFL, Neuzz finds 70 times more branches and 36 more exceptions in different programs.

Based on the generators and discriminators of generative adversarial networks (GANs), we can generate fake but plausible messages of IIoT protocols, which is one of the goals of fuzzing. GAN allows feeding the data to the network without modification. So, we can directly train the IIoT protocol with a few labels. Meanwhile, the GAN structure can apply any network as a generator without a particular functional form [45]. In this framework, we employ long short-term memory neural networks as a generator to generate valid protocol, which improves the passing rate during fuzzing. A Wasserstein GAN (WGAN) is used to generate fuzzing data of industrial control protocols [46], which can learn the structure and distribution of real-world data frames and generate similar data frames without knowing the detailed protocol specification. Zhang *et al.* [47] proposed a CAGFuzz for fuzzing image-based deep learning systems. DLFuzz [48] introduces fuzzing into the adversarial testing of deep learning systems. By considering the sequence features of a protocol, Hu *et al.* [25] employed a GAN using long short-term memory (LSTM) [49] as a generator to fuzz IIoT protocols. Mohit *et al.* [50] improved the fuzzing effectiveness of AFL by predicting reasonable mutation, which is different from using deep learning to generate test cases. To accelerate the mutation process of AFL, RapidFuzz [51] applies GANs to generate potential seeds.

However, the fuzzing effectiveness of deep learning methods may be influenced by the training data scale. Since these methods focus on learning IIoT protocol specifications, their diversity of generated test cases is not high enough.

III. Coverage-Guided Generative Adversarial Networks

A SeqGAN employs LSTM as a generator, which improves the efficiency of sequence data processing. SeqGAN consists of a generator G and a discriminator D. Given an input sequence x, the discriminator aims to maximize $\log D(x)$ and minimize $\log(1 - D(G(s)))$. The generator G generates input by $G(s)$ to maximize $\log(1 - D(G(s)))$ [45], where s is a start token of generated data, i.e.,

$$\min_G \max_D \; V(G, D) = \mathop{\mathbb{E}}_{x \sim P_R(x)} \left[\log D(x)\right] + \mathop{\mathbb{E}}_{s \sim P_G(s)} \left[\log(1 - D(G(s)))\right] \quad (1)$$

where $P_R(x)$ and $P_G(s)$ represent the probability distribution of real data x and the probability distribution of generated data $G(s)$, respectively. Note that $\mathbb{E}[\log D(x)]$ represents expectation of $\log D(x)$ and $\mathbb{E}[\log(1 - D(G(s)))]$ represents expectation of $\log(1 - D(G(s)))$. The loss function $V(G, D)$ will be minimaxed in each training epoch. With the G being replaced by LSTM, the propagation of gradient will become a challenge. SeqGAN uses Monte Carlo tree search and reinforcement learning to overcome this problem. In this article, we propose a coverage-guided GAN, which is more suitable

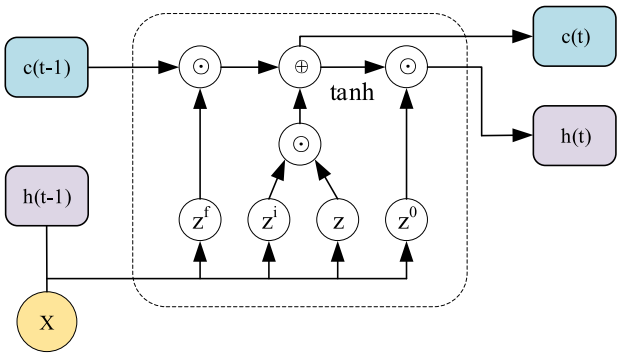

Fig. 2. Single LSTM cell.

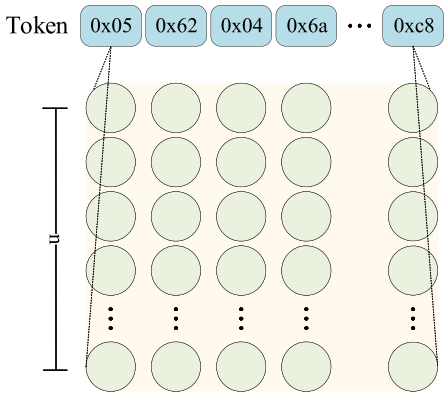

Fig. 3. Embedding layer.

for fuzzing. It uses UCT algorithm and code coverage for evaluating the test cases, which aims to reduce the influence of training data for training effectiveness and increase the diversity and passing rate of test cases.

A. Generator

LSTM is a kind of recurrent neural networks, which completes the selective memory and forgetting of data through forgetting gates and memory gates. Fig. 2 shows the detail of an LSTM cell, where X is input data, described as $X = x^1, \ldots, x^t, \ldots, x^n$, where x^t represents the current input at t moment, h represents a hidden state, and c^t holds the state of the cell at t moment. First, the four states are calculated by the current input x^t with h^{t-1} coming the previous cell. The formulas of the four states are listed follows:

$$z = \tanh\left(W * \left[x^t, h^{t-1}\right]\right) \tag{2}$$

$$z^f = \sigma\left(W^f * \left[x^t, h^{t-1}\right]\right) \tag{3}$$

$$z^i = \sigma\left(W^i * \left[x^t, h^{t-1}\right]\right) \tag{4}$$

$$z^o = \sigma\left(W^o * \left[x^t, h^{t-1}\right]\right) \tag{5}$$

where z^f, z^i, and z^o are obtained by mapping values between 0 and 1 through a sigmoid activation function after merging weight matrixs. Note that z chooses tanh as an activation function

$$c^t = z^f \odot c^{t-1} + z^i \odot z \tag{6}$$

$$h^t = z^o \odot \tanh(t). \tag{7}$$

LSTM cell obtains selective memory from c^t and forgetting from z^i and z^f. The hidden state h^t is determined by z^o and c^t that is scaled with a tanh activation function.

B. Discriminator

SeqGAN has a better performance by using a CNN as a discriminator. As a CNN has advantages in text feature extraction [52], CovGAN also uses it as a discriminator. To make CNN train vector data, an embedding layer with a token sequence is vectorized from $n \times 1$ to $n \times n$, which reduces dimensions compared with one-hot encoding. We can treat this 2-D sequence as a matrix. An embedding layer is shown in Fig. 3.

The CNN uses 12 combinations of 2-D convolution and pooling layers. A dropout layer is added to improve the training efficiency by reducing a part of training data. A full connection layer provides the ability of classification to the CNN. A convolution operation includes a 2-D filter $K \in R^{k_r \times k_c}$, which is applied to a $k_r \times k_c$ matrix. For example, a feature $F_{w,h}$ is generated from a matrix $Map_{w:w+k_r-1,h:h+k_c-1}$ through

$$F_{w,h} = f\left(K \cdot Map_{w:w+k_r-1,h:h+k_c-1} + b\right) \tag{8}$$

where f is a ReLU function, w is between 1 and $(n - k_r + 1)$, h is between 1 and $(n - k_c + 1)$, $\cdot$ is dot product, and $b \in R$ represents a bias. Note that $k_r \times k_c$ represents the size of convolution kernel [6].

C. Parameter Optimization

1) Generator Parameter Optimization: As the test cases generation is converted to a sequence generation problem, it can be formulated as follows. Given a data set of real-world structured sequences, a θ-parameterized generative model G_θ is trained to produce a sequence $Y_{1:T} = (y_1, \ldots, y_t, \ldots, y_T)$, $y_t \in \mathcal{Y}$, where $\mathcal{Y}$ is the vocabulary of candidate tokens. To overcome the problem of intractable gradients, the work in [24] formulates sequence generation as a Markov decision process and trains a GAN using policy gradient [53].

With the policy gradient of reinforcement learning, a reward of the current sequence can directly optimize the parameter of LSTM in the generator. Generating one token can be represented as an action. Due to the constant output length of networks, the reward can only be calculated after the generation of a complete sequence, and we cannot get a reward for each action.

To solve the above problems, we apply the UCT algorithm to fill in an incomplete sequence. It chooses an action through upper confidence bound (UCB) values, which is different from the Monte Carlo tree search in SeqGAN. According to the task of fuzzing, the coverage of a target program is added to the calculation of a reward. The structure of CovGAN is shown in Fig. 4. For a complete sequence, a reward can be obtained immediately. However, the reward of an incomplete sequence is calculated by executing UCT for N times, where N is also called a rollout number. For example, a sequence of length

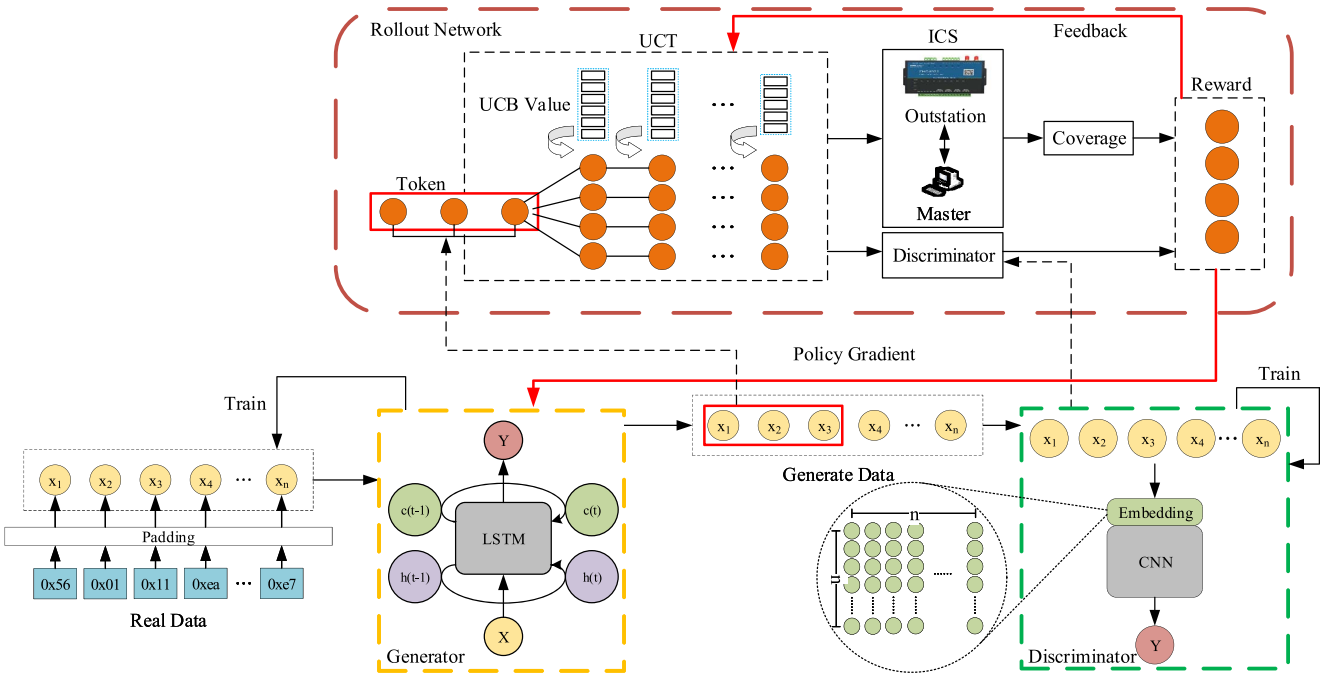

Fig. 4. Parameters optimization of CovGAN.

TABLE I
PARAMETERS

Parameters	Notes
G_θ	A generator in a GAN
G_β	A generator in Rollout network
$x_{i,j}$	The average reward after simulations
$V_{i,j}$	The total number of choices $[token]_{i,j}$
D_ϕ	A discriminator in a GAN
$Q_{D_\phi}^{G_\theta}$	The action-value function for training generator

T is expected and t tokens are currently generated. Then, the UCT will be used to simulate the remaining $T-t$ tokens for N times. We add a threshold th in UCT to decide to generate sequence by generator G_β or UCT, where G_β is a generator for UCT. Through feeding these sequences into the discriminator D_ϕ and the ICS based on DNP3, the reward and code coverage can be obtained. The parameters are summarized in Table I.

The specified UCT of CovGAN is as follows.

1) Selecting the next token with the max UCB value until the sequence length is T.
2) By repeating (1) N times, we can expand N sequences.
3) Feed N sequences into the discriminator and the DNP3 simulator.
4) Updating UCB values by rewards.

In the UCT algorithm, the relationship between the previous decision and the current decision can be regarded as a tree structure, where node $node_{i,j}$ represents a decision. Variable i is a depth of node $node_{i,j}$, and j is an index of node $node_{i,j}$ in depth i. The next decision depends on the UCB values of node $node_{i+1,j}$. The formula of UCB values [23] is as follows:

$$\mathrm{UCB}\big([\mathrm{token}]_{i-1,m}, [\mathrm{token}]_{i,j}\big) = x_{i,j} + c \times \sqrt{\frac{2\ln\left(\sum_{p\in\eta_{i,j}} V_{i+1,p}\right)}{V_{i,j}}} \tag{9}$$

where $\mathrm{UCB}([\mathrm{token}]_{i-1,m}, [\mathrm{token}]_{i,j})$ represents the UCB value of choosing current token $[\mathrm{token}]_{i,j}$ after choosing previous token $[\mathrm{token}]_{i-1,m}$, and $i \in \{1 < i \leq T+1, i \in \mathbb{N}^+\}$ also can be treated as a position that will be generated a token. $\eta_{i,j}$ denotes a set of child nodes of node $node_{i,j}$. $x_{i,j}$ represents the average reward of choosing token $[\mathrm{token}]_{i,j}$ after N simulations, $V_{i,j}$ is the number of choices on $[\mathrm{token}]_{i,j}$, and c controls the balance between $x_{i,j}$ and $V_{i,j}$. $\sum_{p\in\eta_{i,j}} V_{i+1,p}$ is the number of all chosen $[\mathrm{token}]_{i+1,p}$. We define UCB_i to save the UCB values of all token at position i.

Parameters $x_{i,j}$ and $V_{i,j}$ are updated through simulation or update. If node $node_{i,j}$ is the final decision, $x_{i,j}$ and $V_{i,j}$ are gotten by simulation. Otherwise, $x_{i,j}$ and $V_{i,j}$ are obtained by $x_{i+1,p}$ and $V_{i+1,p}$, respectively, where $p \in \eta_{i,j}$. The formulas of $x_{i,j}$ and $V_{i,j}$ are as follows:

$$V_{i,j} = \sum_{p\in\eta_{i,j}} V_{i+1,p} \tag{10}$$

$$x_{i,j} = \frac{\sum_{p\in\eta_{i,j}} x_{i+1,p} V_{i+1,p}}{V_{i,j}}. \tag{11}$$

The policy gradient uses the idea of action value. We model sequence generation as a reinforcement learning process. Based on the idea of action value, the reward of each token generated by GAN's generator can be estimated. Action a is a generating token based on the current sequence that is called state s, and its value is the probability of this action. $D_\phi(Y_{1:T})$ is a probability indicating whether a sequence $Y_{1:T}$ is from real sequence data or not. $Cov(UCT^{G_\beta})$ is a function of obtaining the code coverage and UCT^{G_β} is a collection of

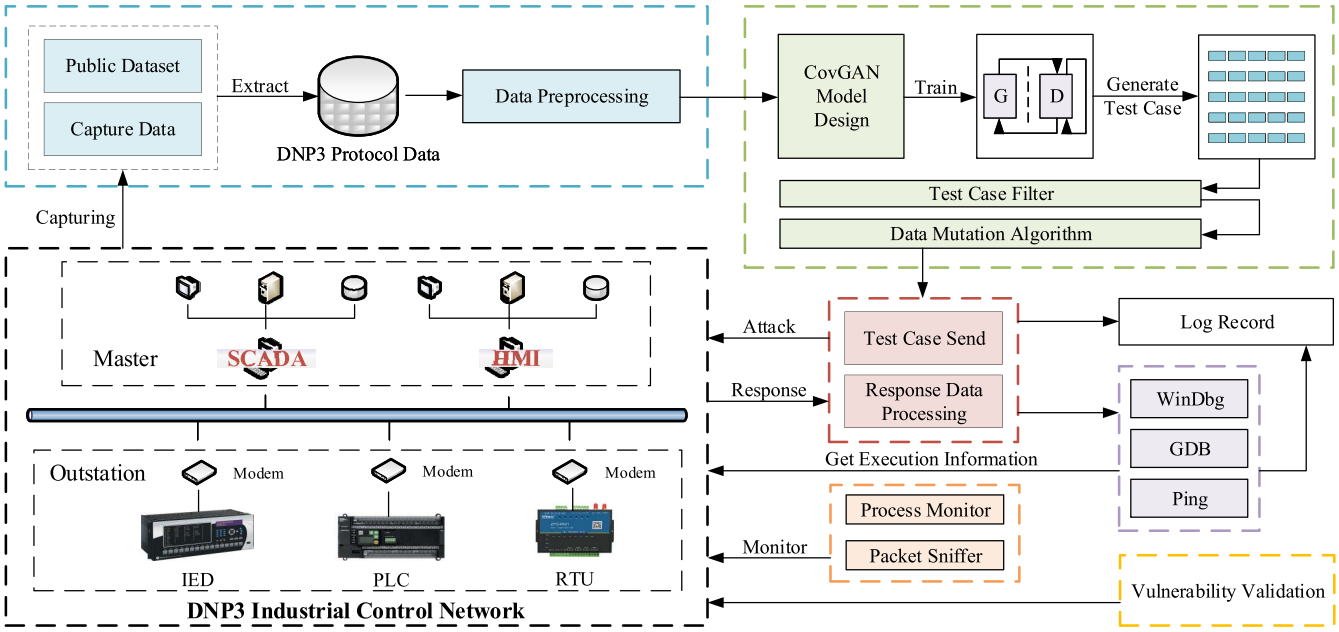

Fig. 5. Fuzzing framework of DNP3 based on CovGAN.

TABLE II
CALCULATION OF REWARD

Algorithm: Reward calculation based on the UCT and code coverage

Input: Data X, Rollout Number N, Sequence Length T, Discriminator D_ϕ, Generator G_β , threshold th
Output: Reward R
while $i = 1, 2, 3, ..., N$ **do**
  while $j = 1, 2, 3, ..., T$ **do**
    if $T < th$ **then**
      UCB($[token]_{j-1,m}$,$[token]_{j,n}$) $\leftarrow max(UCB_j)$
      Select $[token]_{j,n}$ with UCB($[token]_{j-1,m}$,$[token]_{j,n}$)
      Concatenate $X_{1\sim j-1}$ with $[token]_{j,n}$
    else
      Generate $[token]_{j,n}$ by G_β
      Concatenate $X_{1\sim j-1}$ with $[token]_{j,n}$
    end if
    Compute $D_\phi(X)$ and $Cov(X)$
    $R \leftarrow R + D_\phi(X) + Cov(X)$
    Update $V_{i,j}$ by Eq. (10)
    Update $x_{i,j}$ by Eq. (11)
    Update UCB$[token]_{j-1,m}$,$[token]_{j,n}$) by Eq. (9)
  end while
end while
$R \leftarrow R \div N$
return R

sequences that are generated by UCT of CovGAN. Table II shows the calculation of reward through action-value function. The following is the action-value function $Q_{D_\phi}^{G_\theta}$:

$$Q_{D_\phi}^{G_\theta}(s = Y_{1:t-1}, a = y_t) = \begin{cases} \frac{1}{N}\sum_{n=1}^{N} \left(D_\phi\left(Y_{1:t}^n\right) + Cov\left(UCT^{G_\beta}\right)\right) \\ Y_{1:t}^n \in UCT^{G_\beta}(Y_{1:t}; N), & \text{for } t < T \\ D_\phi(Y_{1:t}) + Cov\left(UCT^{G_\beta}\right), & \text{for } t = T. \end{cases} \quad (12)$$

2) Discriminator Parameter Optimization: We train a ϕ-parameterized discriminative model D_ϕ to provide a guidance for improving generator G_θ. Note that Y is the sequence that is generated by UCT or G_β. The discriminator parameters are optimized by a stochastic gradient algorithm. The loss function of D_ϕ is $V(D_\phi)$. The objective of the discriminator [24] is shown as follows:

$$\min \quad V(D_\phi) = -\mathbb{E}_{Y\sim P_{data}}\left[logD_\phi(Y)\right] - \mathbb{E}_{Y\sim G_\theta}\left[\log\left(1 - D_\phi(Y)\right)\right]. \quad (13)$$

By backpropagating the loss of CNN, we can optimize the parameters ϕ_D. The highway network decreases the difficulty of gradient backpropagation in CNN [54]. It controls the nonlinear transformation of each layer by using gates. $H(x, W_H)$ is a nonlinear transform and $T(x, W_T)$ is a transform gate in a highway network. The parameters of $H(x, W_H)$ and $T(x, W_T)$ are represented as W_H and W_T, respectively. The output y in a layer of highway network is formulated as follows [54]:

$$y = H(x, W_H) \cdot T(x, W_T) + x \cdot (1 - T(x, W_T)). \quad (14)$$

If $T(x, W_T) = 0$, the output is x; if $T(x, W_T) = 1$, the output is x that is transformed.

IV. FUZZING FOR DNP3 PROTOCOL

In this section, a fuzzing framework named CGFuzzer is designed for the DNP3 protocol, which is shown in Fig. 5.

A. Design of Fuzzing Framework

The fuzzing framework of CGFuzzer includes a data acquisition module, a test case generation module, a debugging module, a monitoring module, and a vulnerability verification module.

Data Acquisition: Collecting the protocol data of DNP3. This module outputs the traffic captured at the preset IP and port into files.

Test Case Generation: A module is employed to generate, filter, and mutate DNP3 data. According to the specific format of the protocol, the data are extracted and separated. Then, they

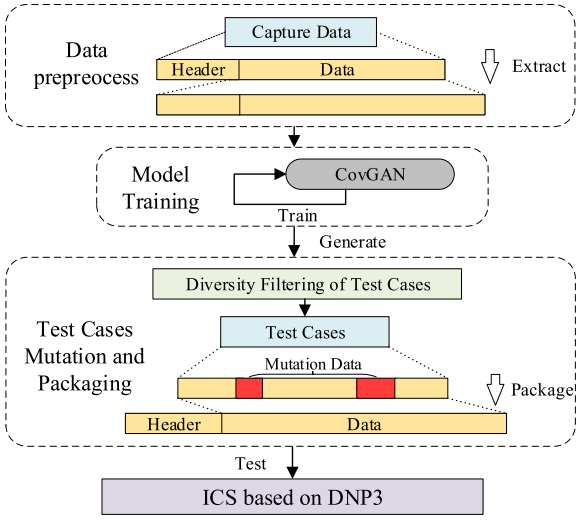

Fig. 6. Process of the test case generation module.

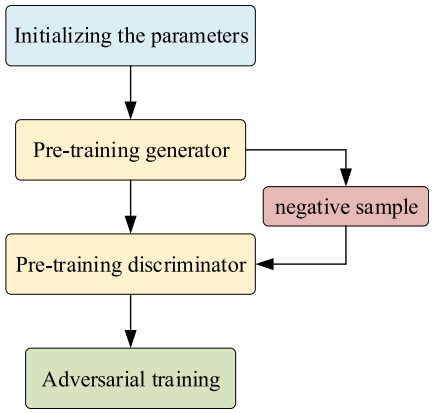

Fig. 7. Process of training.

are preprocessed by some methods, such as integer overflow or string overflow. Before a filter algorithm with similarity is executed, the preprocessed data are inputted into CovGAN for training. The data generated by CovGAN can be mutated with a random mutation strategy for higher diversity. The process of the test case generation module is shown in Fig. 6.

Debugging: Using debugging tools to detect exceptions during testing. We can record system information and the context of ICSs based on DNP3 by GDB, Ping, and WinDbg in fuzzing, which is helpful to vulnerability analysis.

Monitoring (Monitoring System Status): It includes a monitoring packet method, process monitoring, and sniffing. The monitoring packet method is to send normal data to ICSs based on DNP3 at intervals to check whether they are normal or not. Process monitoring determines the status of ICSs by obtaining information about CPU and memory. Monitoring packets in the ICS of DNP3 can be implemented by using a sniffer.

Vulnerability Verification: Verifying whether the abnormal data in the process of fuzzing are vulnerabilities.

B. Data Preprocessing

Since the data set is collected from a network card port, the DNP3 data need to be extracted according to the PCAP [55] file format. The data are divided into several segments, which facilitates the model training. DNP3 has three layers: 1) a data link layer; 2) a transport layer; and 3) an application layer. The first layer consists of start bytes, length, destination, source, and checksum. The second layer is used to indicate packet status information, such as packet ID. The third layer contains control fields, function codes, and data objects.

The training data consist of normal data and attack data, and we feed these into CovGAN separately. After dividing data into different segments and mutating some of them, the payload data are split into 2-bit hexadecimal due to the uncertain length, which is different from other parts. The hexadecimal data of the attack data are converted into decimal after boundary value mutation. The normal data are only converted to decimal. As the input layer dimension of CovGAN is fixed, the incomplete data are necessary to fill values.

C. Training of Model

In order to improve training efficiency, a process of adversarial training is designed, as depicted in Fig. 7. First, the generator generates random DNP3 protocol data by executing UCT for N times. Then, the rollout network calculates the reward using the code coverage and the output of the discriminator. The discriminator maps the generated data to 2-D by an embedding layer. Finally, the full connection layer in the discriminator outputs the loss values to optimize the classification effect of the discriminator.

1) *Initialize the Parameters of Generator and Discriminator in CovGAN:* The dimension of hidden layers is 12, sequence length is 26, and batch size is 64. The number of LSTM cells is 26.
2) *Pretrain the Generator by Maximum-Likelihood Estimation:* As the structure of the CovGAN network is complex, using maximum-likelihood estimation to estimate parameters is beneficial to training.
3) *Pretrain the Discriminator:* The data generated by the generator are labeled as a negative sample and input into the discriminator for pretraining.
4) *Adversarial Training:* After using normal data for training, we add attack data for training again to improve the effect of training. The parameters θ of the generator and the discriminator are optimized by the Adam optimizer [56]. The learning rates of the generator and the discriminator are set to 0.01 and 0.0001, respectively.

D. Test Cases Generation

There are some problems of using data generated by CovGAN on fuzzing directly, which are duplication of data and insufficient diversity. Therefore, diversity filtering and random mutation strategy are conducted to solve the above problems.

1) Diversity Filtering Algorithm: Levenshtein distance [57] refers to the minimum number of edit operations between two strings, which include replacing, inserting, and deleting. Similar test cases are filtered by similarity based on

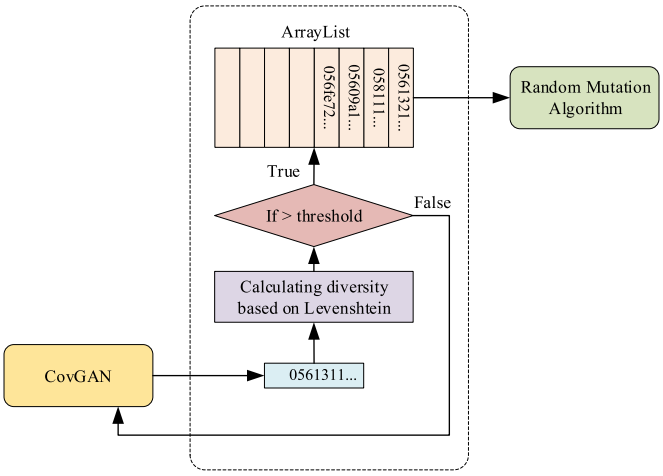

Fig. 8. Process of the diversity filtering algorithm.

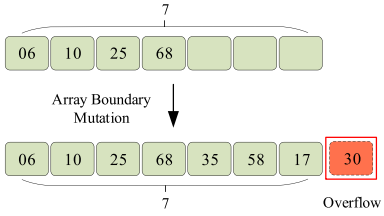

Fig. 9. Array boundary overflow.

Levenshtein distance. The function of similarity is designed as follows:

$$s_{i,j} = 1 - \frac{\text{edit}(str_i, str_j)}{\max(len(str_i), len(str_j))} \quad str_i \neq str_j. \tag{15}$$

We calculate the similarity $s_{i,j}$ between the test cases str_i and the str_j according to (15). Note that edit(str_i, str_j) is the number of steps to transform str_i to str_j, $len(str_i)$ represents the length of str_i, and max($len(str_i)$, $len(str_j)$) is a function to obtain the maximum value between $len(str_i)$ and $len(str_j)$. Through setting a similarity threshold Δ, we filter similar test cases and store other test cases that are not similar in a candidate queue *TCA* for mutating. Fig. 8 depicts the process of a diversity filtering algorithm.

The test cases that are less than a threshold of similarity are queued by a diversity filtering algorithm. We can apply multithreading to speed up the filtering algorithm with synchronization that avoids dirty data.

2) Random Mutation Algorithm: It completes the mutation of test cases by calling the mutation approaches, which are described as follows.

1) *Boundary Value Mutation:* It randomly mutates the array length and the number value. The purpose is to cause array overflow or integer overflow exceptions in the program. Array boundary overflow is shown in Fig. 9.
2) *Character Encoding Mutation:* It aims at causing the exception in decoding. Fig. 10 shows the details of this mutation.
3) *Character Replication and Exchange Mutation:* It is a mutation method of randomly copying and exchanging of characters. This method can improve the diversity of test cases and increase the probability of triggering exceptions.

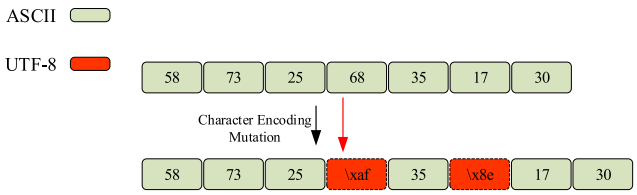

Fig. 10. Character encoding mutation.

TABLE III
FUNCTIONS OF MUTATION OPERATIONS

Function	Note
$mangleResize$	Fill spaces into random positions
$mangleByte$	Fill uint8 data into random positions
$mangleBit$	Flip bit in random positions
$mangleBytes$	Fill 2-4 bytes data into random positions
$mangleMagic$	Fill random boundary value
$mangleNegByte$	Data Inversion
$mangleIncByte$	Data plus 1
$mangleDecByte$	Data minus 1
$mangleMemMove$	Move data with random lengths
$mangleCloneByte$	Exchang data at random locations
$mangleShrink$	Drop data with random lengths

TABLE IV
STRATEGY OF RANDOM MUTATION

Algorithm: Strategy of random mutation

Input: Test Case Array TCA, Mutation Factor F, and Functions $Func = \{func_1, func_2, ..., func_n\}$
1: Initialize parameters of mutation function
2: **while** TCA is not NULL **do**
3: Get $test$ from TCA
4: **while** $i = 1, 2, 3, ..., F$ **do**
5: Select mutation function $func_i$ from $Func$
6: $test \leftarrow func_i(test)$
7: **end while**
8: Package and Send $test$ to DNP3 simulator
9: Capture responding $resTest$
10: **if** $resTest$ is exception **then**
11: Insert $test$ into TCA
12: **end if**
13: **end while**

Based on the three approaches described above, we implement mutation functions based on the AFL. The detailed functions are shown in Table III.

The mutation is accomplished by executing these functions randomly. In order to reduce the time complexity, we do not utilize the coverage-guided approach to guide the mutation that is different from the strategy of AFL. The logic of the random mutation strategy is presented in Table IV.

First, we initialize each mutation function to specify the length of the mutation and the range of random values. When the candidate queue *TCA* that stores test cases is not empty, the mutation function fun_i is randomly selected from the collection of mutation operations *Func* to mutate test cases *test* until the mutation factor F becomes 0. Note that the input and the output of each function in *Func* are test cases. Then, we convert the *test* to hexadecimal and package *test* to send to a DNP3

simulator. With the captured response information *resTest* from the DNP3 simulator, we can determine whether the test cases are causing exceptions. Since the log of exceptions is not constant, we set multiple fields to match the log, e.g., exception, offline, close, stop, reset, etc. Finally, the test cases that cause the exception are added to the candidate mutation queue *TCA* and wait for the next mutation.

V. Experimental Evaluation and Analysis

In this section, we compare CGFuzzer with GANFuzz [25], SeqFuzzer [26], and PeachFuzzer [28], respectively. A part of the data set is captured in a DNP3 simulator by Wireshark [58] and its number is about 1500. We set this data set as the pretraining data set. The DNP3 simulator is constructed by the opendnp3 library. The other part of the data set is obtained from a public data set [59], and its number is about 9500. The data of the public data set are hexadecimal and its length is between 96 and 192 bit. This public data set includes DNP3 traffic packets with various attacks, such as flooding attack, injection attack, replay attack, masquerading, etc. We employ the public data set as adversarial training data set. We set 80% of these two data sets as a training set, and the remaining 20% as a test set, respectively. Details of the experimental environment are as follows: a machine running Windows 10, with 16-GB of RAM and Intel Core i5-6300HQ CPU @ 2.30-GHz processor. Some requirements, such as OpenCppCoverage, Cmake, Opendnp3, ASIO, etc., are employed to build a DNP3 simulator and statistics code coverage. The training of CovGAN based on Python 3.6.9 and TensorFlow 1.7(GPU) is utilized.

We use the following metrics to measure the fuzzing effectiveness.

1) *Test Case Recognition Rate (TCRR): TCRR* refers to the percentage of test cases recognized by a test target. It indicates the proportion of valid test cases. In the fuzzing of IIoT protocols, we consider that the test case format is correct if the target can recognize and respond to this test case. A higher *TCRR* indicates that more generated test cases are similar to the real-world sequences in format. Conversely, the lower *TCRR* means more test cases are dropped directly by the target, which indicates that generator G needs to be adjusted or retrained. The formula is shown as follows:

$$TCRR = \frac{nRecognized}{nSent} \times 100\% \tag{16}$$

where *nRecognized* is the total number of test cases recognized and *nSent* is the total number of test cases that are sent to IIoT systems.

2) *Number of Exception Triggering (NET): NET* refers to the specific exceptions found. It is an important metric of the fuzzer efficiency. The specific formula is as follows:

$$NET = nExceptions \tag{17}$$

where *nExceptions* indicates the number of exceptions found.

3) *Diversity of Generated Data (DGD): DGD* refers to the ability to maintain the training data diversity. More diversity of generated test data frames is likely to cause more exceptions. This metric focuses on the distribution between the number of the generated data that is less than the threshold of similarity and itself. It is also an important metric for the fuzzing method's effectiveness

$$DGD = \frac{nDiversity}{nGenerated} \times 100\% \tag{18}$$

where *nDiversity* is the total number of the generated data that is less than the threshold of similarity, and *nGenerated* is the total number of generated data.

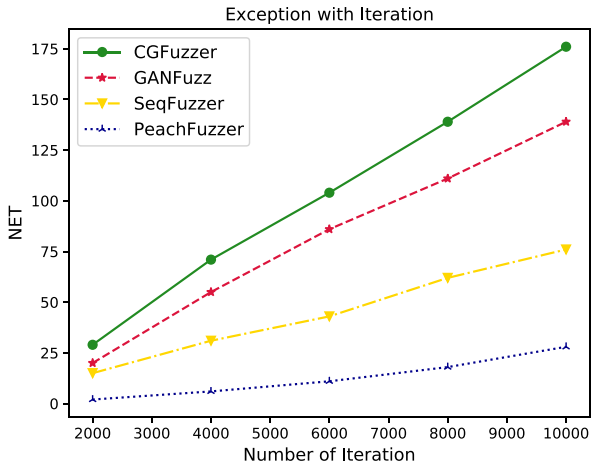

Fig. 11. Number of triggering exceptions.

Then, we train the CovGAN with $epoch = 100$, discriminator learning rate $lr = 0.0001$, and dropout probability $dp = 0.75$.

During the communication between a master and a slaver of DNP3, Scapy [60] is applied to send data to the port for fuzzing. It is reasonable to use the number of exceptions to evaluate the effectiveness of the method. After fuzzing with 10 000 test cases in CGFuzzer, GANFuzz, SeqFuzzer, and PeachFuzzer, respectively, we compare testing logs and find that all results are connection interrupt and reset. The results are summarized as follows.

The test results are recorded every 1000 iterations. We can see that the vulnerability mining ability of CGFuzzer is higher than other methods in Fig. 11. The record values of CGFuzzer increase between 20 and 30, while the values of other methods are less than that of CGFuzzer. We define a structure of DNP3 in XML and set the mutation range of the data according to PeachFuzzer documents. The lack of self-learning ability in peachfuzzer makes the number of exceptions fluctuate between 0 and 5.

The test execution time also reflects the fuzzing efficiency. In 10 000 iterations, the execution time of Peach is the longest, which is more than 500 min. The other methods including CGFuzzer are almost 80 min. Fig. 12 shows the time cost for different methods.

The functions of recurrence and analysis in PeachFuzzer affect the above test results, and the experiments below are conducted to exclude these factors by using the time of test cases generation as a metric. CGFuzzer, GANFuzz, and SeqFuzzer count the generation time in the code of generation and mutation. Since the Peach installation is not at a source code level, the POPEN module will be used to obtain the time of generating test cases, and the results are shown in Fig. 13.

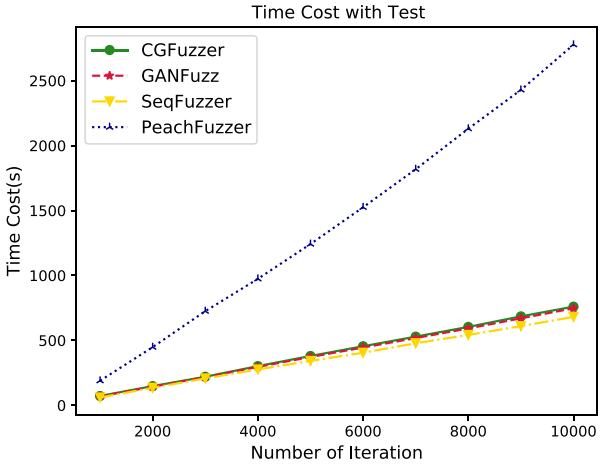

Fig. 12. Time cost in different methods.

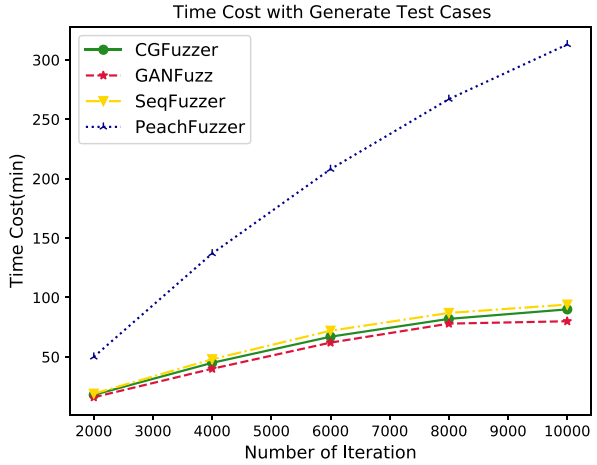

Fig. 13. Time cost in generating test cases.

After just calculating the generation time, the time growth of PeachFuzzer decreases significantly as the number of iterations increases, but it is still slower than the other three algorithms. Fuzzing requires a high diversity of test cases to improve the efficiency of vulnerability mining. However, this requirement will bring a large number of invalid test cases, which seriously reduces the passing rate of test cases. Therefore, a comparison of the passing rate is necessary. When the start bytes length exception and header check error are raised, it means that the check of the header is not passed, and the test cases are not inputted to the system. We calculate the passing rate by counting the number of these two exceptions, and the results are depicted in Fig. 14. The passing rate of CGFuzzer is about 92%, which is higher than Peach and other deep learning methods.

Diversity means, to some extent, the degree of mutation between test cases. This part is only for grammar learning algorithms, as Peach is a variable-length test cases generation method that affects the diversity statistics. As is shown in Fig. 15, CGFuzzer always has the highest diversity in 10 000 tests, thanks to the diversity filtering algorithm.

We compare the effectiveness of CGFuzzer with different parameters. The results are shown in Table V.

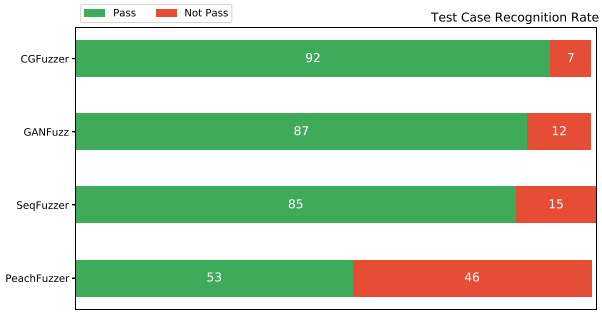

Fig. 14. Comparison of passing rate.

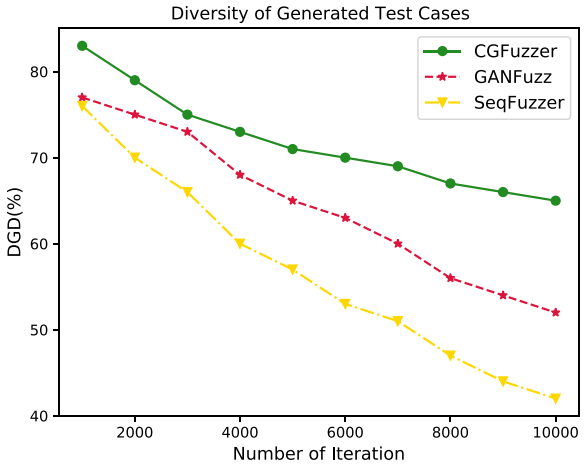

Fig. 15. Line graph of diversity changing in 10 000 fuzzing.

TABLE V
RESULT UNDER DIFFERENT PARAMETERS

Parameters	$TCRR$	NET	DGD
$epoch$ = 50, lr = 0.0001, dp = 0.5	87.25%	167	69.81%
$epoch$ = 100, lr = 0.0001, dp = 0.5	91.06%	171	65.22%
$epoch$ = 50, lr = 0.0003, dp = 0.5	89.78%	171	68.47%
$epoch$ = 100, lr = 0.0003, dp = 0.5	90.11%	159	65.61%
$epoch$ = 50, lr = 0.0001, dp = 0.75	86.37%	163	68.70%
$epoch$ = 100, lr = 0.0001, dp = 0.75	92.26%	172	65.32%
$epoch$ = 50, lr = 0.0003, dp = 0.75	90.16%	168	67.14%
$epoch$ = 100, lr = 0.0003, dp = 0.75	87.84%	162	65.51%

When *epoch* is 100, we can obtain a higher passing rate (*TCRR*). Since the features of the protocol header are fitted by the CovGAN. If $epoch = 100$, $lr = 0.0001$, and $dp = 0.75$, the number of triggering exceptions (*NET*) and the passing rate (*TCRR*) is more than that of other parameters, respectively.

Code coverage means the depth and width of code execution branches, and a higher code coverage increases the probability of mining vulnerabilities. To obtain the coverage more easily, we calculate coverage on a DNP3 decoder. Code coverage statistics are implemented by using the OpenCppCoverage tool. To ensure the accuracy of the results, 1000 items are randomly sampled from 10 000 items of test cases for calculating code coverage. The code coverage results are shown in Fig. 16, where the code coverage of CGFuzzer is 68%.

Fig. 16 shows the code coverage, but does not reflect the specific execution depth. Therefore, the experiments for the execution depth are conducted. LPDU, TPDU, and APDU

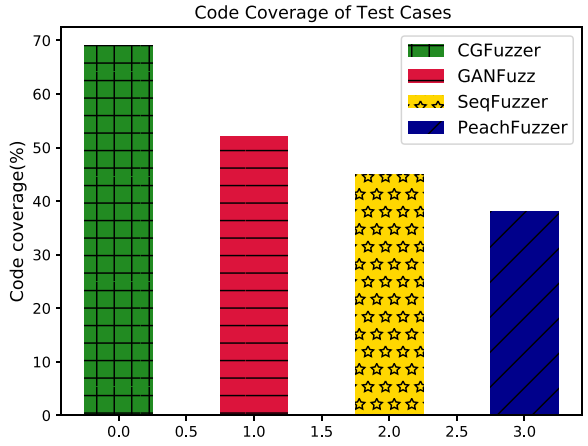

Fig. 16. Comparison of code coverage.

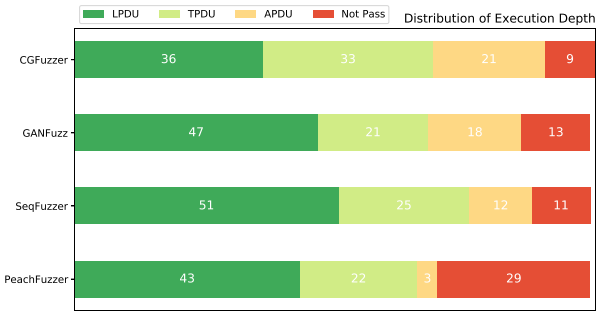

Fig. 17. Distribution of execution depth in different methods.

represent the link protocol data unit, transmission protocol data unit, and application protocol data unit in the process of DNP3 decoding, respectively. The specific results are shown in Fig. 17.

It can be seen that the highest percentages of TPDUs and APDUs are achieved in CGFuzzer, accounting for 33% and 21%, respectively; both are higher than that of other methods. This means that CGFuzzer can generate test cases with a higher execution depth. It should be noted that the percentage of executing depth only indicates that the current layer is executed. For example, the percentage of TPDU represents that only TPDU is executed. However, the test cases executed to the APDU layer have also been executed TPDU.

After experiments, we verify the effectiveness of the CGFuzzer. Although GANFuzz and SeqFuzzer learn the features of the protocol by using deep learning, there are less considerations for promoting the diversity of test cases. When a test case is being generated, they lack the guidance of code coverage and strategy of balancing exploitation and exploration. This makes SeqFuzzer and GANFuzz generate lower *NET* and code coverage test cases than CGFuzzer. The lack of feature learning for protocol makes the passing rate and code coverage of PeachFuzzer not as good as those of other methods to be compared. CovGAN can preserve features of protocol very well. A protocol header with a high passing rate can be obtained. Fuzzing should focus on high diversity rather than high similarity. CovGAN combines UCT with code coverage to balance high diversity and similarity. This method only consumes partial resources to guide the generator to generating high diversity and code coverage test cases, while maintaining the performance of the generator.

VI. Conclusion

This article proposes a CovGAN for IIoT protocols, which combines UCB apply to a tree with SeqGANs for improving passing rate and code coverage. Based on the above network, we present a fuzzing framework named CGFuzzer for IIoT protocols, which can be used as a reference for other IIoT protocols. We ultimately evaluate this framework by comparing it with the existing methods. The results obtained indicate the application potential of the proposed method.

In future studies, we expect to build a smarter and more efficient fuzzing framework for IIoT protocols. First, the sequence structure of other IIoT protocols can be introduced through transfer learning to improve the training effect with a few samples. Second, we can simplify the network structure in the generator to reduce the training time, and combining fuzzing with other methods in vulnerability mining is also a feasible option. Finally, mutation strategies can introduce artificial intelligence algorithms to make the fuzzing learn regular mutation. To further conduct the security analysis of IIoT protocols, we employ formal protocol verifier tools in the future, such as Proverif [61]. The proposed algorithm can be applied to cyber–phsycial sytems.

References

[1] S. Keith, J. Falco, and K. Scarfone, "Guide to industrial control systems (ICS) security," Nat. Inst. Stand. Technol., Gaithersburg, MD, USA, document NIST SP 800-82, 2011.

[2] W. Sun, S. Lei, L. Wang, Z. Liu, and Y. Zhang, "Adaptive federated learning and digital twin for Industrial Internet of Things," *IEEE Trans. Ind. Informat.*, vol. 17, no. 8, pp. 5605–5614, Aug. 2021.

[3] Z. H. Yu, H. X. Gao, D. Wang, A. A. Alnuaim, M. Firdausi, and A. M. Mostafa, "SEI2RS malware propagation model considering two infection rates in cyber-physical systems," *Physica A Stat. Mech. Appl.*, vol. 597, Jul. 2022, Art. no. 127207.

[4] S. Lüders, "STUXNET and the impact on accelerator control systems," in *Proc. Int. Conf. Accelerator Large Exp. Phys. Control Syst.*, 2011, pp. 1285–1288.

[5] K. Mohit. "TSMC Chip Maker Blames WannaCry Malware for Production Halt." The Hacker News. 2018. [Online]. Available: https://thehackernews.com/2018/08/tsmc-wannacry-ransomware-attack.html

[6] W. Y. Lv, J. W. Xiong, J. Q. Shi, Y. H. Huang, and S. C. Qin, "A deep convolution generative adversarial networks based fuzzing framework for industry control protocols," *J. Intell. Manuf.*, vol. 32, no. 2, pp. 441–457, 2021.

[7] M. Kravchik and A. Shabtai, "Efficient cyber attack detection in industrial control systems using lightweight neural networks and PCA," *IEEE Trans. Dependable Secure Comput.*, early access, Jan. 8, 2021, doi: 10.1109/TDSC.2021.3050101.

[8] K. Srinivasan and C. Rosenberg, "How Internet concepts and technologies can help green and smarten the electrical grid," in *Proc. 1st ACM SIGCOMM Workshop Green Netw.*, 2010, pp. 35–40.

[9] "DNP3 Official Website." DNP3. [Online]. Available: https://www.dnp.org/About/Overview-of-DNP3-Protocol (Accessed: Mar. 16, 2021).

[10] S. Andy, *Open Modbus/TCP specification*, Schneider Electr., Rueil-Malmaison, France, 1999.

[11] "Profinet Official Website." Profibus. [Online]. Available: https://www.profibus.com/ (Accessed: Mar. 16, 2021).

[12] S. Figueroa-Lorenzo, J. Añorga, and S. Arrizabalaga, "A survey of IIoT protocols: A measure of vulnerability risk analysis based on CVSS," *ACM Comput. Surveys*, vol. 53, no. 2, pp. 1–53, 2020.

[13] "DNP3." [Online]. Available: https://en.wikipedia.org/wiki/DNP3 (Accessed: Mar. 16, 2021).

[14] M. Majdalawieh, F. Parisi-Presicce, and D. Wijesekera, "DNPSec: Distributed network protocol version 3 (DNP3) security framework," in *Advances in Computer, Information, and Systems Sciences, and Engineering*. Dordrecht, The Netherlands: Springer, 2007, pp. 227–234.
[15] *DNP3 Specification Application Layer Part 1: Basics*, vol. 2, DNP User's Group, Raleigh, NC, USA, 2007.
[16] *DNP3 Specification Transport Function*, vol. 3, DNP User's Group, Raleigh, NC, USA, 2007.
[17] *DNP3 Specification Data Link Layer*, vol. 4, DNP User's Group, Raleigh, NC, USA, 2007.
[18] K. Shahzad *et al.*, "An efficient and secure revocation-enabled attribute-based access control for eHealth in smart society," *Sensors*, vol. 22, no. 1, p. 336, 2022.
[19] K. Xue, N. Gai, J. Hong, D. S. L. Wei, P. Hong, and N. Yu, "Efficient and secure attribute-based access control with identical sub-policies frequently used in cloud storage," *IEEE Trans. Dependable Secure Comput.*, vol. 19, no. 1, pp. 635–646, Jan./Feb. 2022.
[20] H. Zhang, Y. Bi, H. Guo, W. Sun, and J. Li, "ISVSF: Intelligent vulnerability detection against Java via sentence-level pattern exploring," *IEEE Syst. J.*, vol. 16, no. 1, pp. 1032–1043, Mar. 2022.
[21] H. Liang, X. Pei, X. Jia, W. Shen, and J. Zhang, "Fuzzing: State of the art," *IEEE Trans. Rel.*, vol. 67, no. 3, pp. 1199–1218, Sep. 2018.
[22] K. Chen, C. Song, L. Wang, and Z. Xu, "Using memory propagation tree to improve performance of protocol fuzzer when testing ICS," *Comput. Security*, vol. 87, pp. 1–13, Nov. 2019.
[23] G. Sylvain and D. Silver, "Combining online and offline knowledge in UCT," in *Proc. 24th Int. Conf. Mach. Learn.*, 2007, pp. 273–280.
[24] L. T. Yu, W. N. Zhang, J. Wang, and Y. Yu, "SeqGAN: sequence generative adversarial nets with policy gradient," in *Proc. AAAI Conf. Artif. Intell.*, vol. 31, 2017, pp. 12–22.
[25] 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. Front.*, 2018, pp. 138–145.
[26] 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*, 2019, pp. 59–67.
[27] C. Chen, B. J. Cui, J. X. Ma, R. P. Wu, J. C. Guo, and W. Q. Liu, "A systematic review of fuzzing techniques," *Comput. Security*, vol. 75, pp. 118–137, Jun. 2018.
[28] "Peach software official website." Peach Tech. [Online]. Available: https://www.peach.tech/ (Accessed: Mar. 16, 2021).
[29] S. Bradshaw. "Spike Software Official Website." [Online]. Available: http://www.immunitysec.com/ (Accessed: Mar. 16, 2021).
[30] H. Christian, K. Herzig, and A. Zeller, "Fuzzing with code fragments," in *Proc. 21st USENIX Security Symp.*, 2012, pp. 445–458.
[31] V. Spandan, S. Rawat, I. Haller, and H. Bos, "Ifuzzer: An evolutionary interpreter fuzzer using genetic programming," in *Proc. Eur. Symp. Res. Comput. Security*, 2016, pp. 581–601.
[32] J. J. Wang, B. H. Chen, L. Wei, and Y. Liu, "Superion: Grammar-aware greybox fuzzing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 724–735.
[33] "AFL Software Official Website." Google. [Online]. Available: http://lcamtuf.coredump.cx/afl/ (Accessed: Mar. 16, 2021).
[34] "Jsfunfuzz." [Online]. Available: https://github.com/MozillaSecurity/funfuzzy (Accessed: Mar. 16, 2021).
[35] B. Marcel, V. T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as Markov chain," *IEEE Trans. Softw. Eng.*, vol. 45, no. 5, pp. 489–506, Dec. 2017.
[36] M. Böhme, V.-T. Pham, M. D. Nguyen, and A. Roychoudhury, "Directed greybox fuzzing," in *Proc. ACM SIGSAC Conf. Comput. Commun. Security*, 2017, pp. 2329–2344.
[37] V. T. Pham, M. Böhme, and A. Roychoudhury, "AFLNET: A greybox guzzer for network protocols," in *Proc. IEEE 13th Int. Conf. Softw. Test. Validation Verification*, 2020, pp. 460–465.
[38] R. Helmke, E. Winter, and M. Rademacher, "EPF: An evolutionary, protocol-aware, and coverage-guided network fuzzing framework," in *Proc. Int. Conf. Privacy Security Trust*, 2021, pp. 1–7.
[39] G. Vijay, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *Proc. IEEE 31st Int. Conf. Softw. Eng.*, 2009, pp. 474–484.
[40] Q. Zhang, J. Y. Wang, M. A. Gulzar, R. Padhye, and M. Kim, "BigFuzz: Efficient fuzz testing for data analytics using framework abstraction," in *Proc. 35th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2020, pp. 722–733.
[41] V. Atlidakis, P. Godefroid, and M. Polishchuk, "RESTler: Stateful rest API fuzzing," in *Proc. IEEE/ACM 41st Int. Conf. Softw. Eng.*, 2019, pp. 748–758.
[42] Y. Li, H. Chen, C. Zhang, S. Xiong, C. Liu, and Y. Wang, "Ori: A greybox fuzzer for SOME/IP protocols in automotive Ethernet," in *Proc. 27th Asia-Pacific Softw. Eng. Conf.*, 2020, pp. 495–499.
[43] G. Patrice, H. Peleg, and R. Singh, "Learn&fuzz: Machine learning for input fuzzing," in *Proc. 32nd IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2017, pp. 50–59.
[44] D. D. She, K. X. Pei, D. Epstein, J. F. Yang, B. Ray, and S. Jana, "NEUZZ: efficient fuzzing with neural program smoothing," in *Proc. IEEE Symp. Security Privacy*, 2019, pp. 803–817.
[45] I. Goodfellow *et al.*, "Generative adversarial nets," in *Proc. 27th Adv. Neural Inf. Process. Syst.*, 2014, pp. 2672–2680.
[46] Z. Li, H. Zhao, J. Shi, Y. Huang, and J. Xiong, "An intelligent fuzzing data generation method based on deep adversarial learning," *IEEE Access*, vol. 7, pp. 49327–49340, 2019.
[47] P. Zhang, B. Ren, H. Dong, and Q. Dai, "CAGFuzz: Coverage-guided adversarial generative fuzzing testing for image-based deep learning systems," *IEEE Trans. Softw. Eng.*, early access, Nov. 2, 2021, doi: 10.1109/TSE.2021.3124006.
[48] J. Guo, Y. Zhao, H. Song, and Y. Jiang, "Coverage guided differential adversarial testing of deep learning systems," *IEEE Trans. Netw. Sci. Eng.*, vol. 8, no. 2, pp. 933–942, Apr.–Jun. 2021.
[49] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.
[50] R. Mohit, W. Blum, and R. Singh, "Not all bytes are equal: Neural byte sieve for fuzzing," 2017, *arXiv:1711.04596*.
[51] A. S. Ye, L. N. Wang, L. Zhao, J. P. Ke, W. Q. Wang, and Q. L. Liu, "RapidFuzz: accelerating fuzzing via generative adversarial networks," *Neurocomputing*, vol. 460, no. 14, pp. 195–204, 2021.
[52] X. Zhang and Y. LeCun, "Text understanding from scratch," 2015, *arXiv:1502.01710*.
[53] R. S. Sutton, D. A. McAllester, S. P. Singh, and Y. Mansour, "Policy gradient methods for reinforcement learning with function approximation," in *Proc. Adv. Neural Inf. Process. Syst.*, vol. 99, 1999, pp. 1057–1063.
[54] R. K. Srivastava, K. Greff, and J. Schmidhuber, "Highway networks," 2015, *arXiv:1505.00387*.
[55] "PCAP." [Online]. Available: https://zh.wikipedia.org/wiki/Pcap (Accessed : Mar. 16, 2021).
[56] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," 2014, *arXiv:1412.6980*.
[57] J. L. Yu and B. Liu, "A normalized levenshtein distance metric," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 29, no. 6, pp. 1091–1095, Jun. 2007.
[58] "Wireshark." [Online]. Available: https://www.wireshark.org/ (Accessed: Mar. 16, 2021).
[59] "DNP3 Cyber-Attack Datasets." [Online]. Available: https://github.com/qut-infosec/2017QUT_DNP3 (Accessed: Mar. 16, 2021).
[60] "Scapy." [Online]. Available: https://scapy.net/ (Accessed: Mar. 16, 2021).
[61] Z. Yu, H. Gao, D. Wang, A. A. Alnuaim, M. Firdausi, and A. M. Mostafa, "SEI^{2}RS malware propagation model considering two infection rates in cyber-physical systems," *Physica A, Stat. Mech. Appl.*, vol. 597, Jul. 2022, Art. no. 127207, doi: 10.1016/j.physa.2022.127207.

Zhenhua Yu (Member, IEEE) received the B.S. and M.S. degrees from Xidian University, Xi'an, China, in 1999 and 2003, respectively, and the Ph.D. degree from Xi'an Jiaotong University, Xi'an, in 2006.

He is currently a Professor with the Institute of System Security and Control, College of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an. He has authored more than 30 technical papers for conferences and journals and holds two invention patents. His research mainly focuses on cyber–physical systems, unmanned aerial vehicles, multiagent systems, artificial intelligence system security, and nonlinear dynamical systems.

Haolu Wang received the bachelor's degree in Internet of Things engineering from Xi'an University, Xi'an, China, in 2018. He is currently pursuing a master's degree in computer science and technology from Xi'an University of Science and Technology, Xi'an.

His current research interests include vulnerability mining, fuzzing, and the security of industry control systems.

Dan Wang (Member, IEEE) received the B.S. and Ph.D. degrees from Xidian University, Xi'an, China, in 2013 and 2019, respectively.

She is currently an Associate Professor with the School of Computer Science and Technology, Xi'an University of Science and Technology, Xi'an. She was a Visiting Researcher with the University of Alberta, Edmonton, AB, Canada, from January 2016 to September 2019. Her current research interests include computational intelligence, data mining, granular computing, and fuzzy system modeling.

Zhiwu Li (Fellow, IEEE) received the B.S. degree in mechanical engineering, the M.S. degree in automatic control, and the Ph.D. degree in manufacturing engineering, respectively, from Xidian University, Xi'an, China, in 1989, 1992, and 1995, respectively.

He joined Xidian University in 1992. He was a Visiting Professor with the Systems Control Group, Department of Electrical and Computer Engineering, University of Toronto, Toronto, ON, Canada. From February 2007 to February 2008, he was a Visiting Scientist with the Laboratory for Computer-Aided Design & Lifecycle Engineering, Department of Mechanical Engineering, Technion—Israel Institute of Technology, Haifa, Israel. From November 2008 to October 2010, he was a Visiting Professor with the Automation Technology Laboratory, Institute of Computer Science, Martin-Luther University of Halle-Wittenburg, Halle, Germany. He is currently with the Institute of Systems Engineering, Macau University of Science and Technology, Macau, China. His research interests include Petri net theory and application, supervisory control of discrete event systems, workflow modeling and analysis, system reconfiguration, game theory, and data and process mining.

Dr. Li is a recipient of an Alexander von Humboldt Research Grant, Alexander von Humboldt Foundation, Germany, and Research in Paris, France. He is listed in Marquis Who's Who in the world, 27th Edition, 2010. He currently chairs the Discrete Event Systems Technical Committee of the IEEE Systems, Man, and Cybernetics Society.

Houbing Song (Senior Member, IEEE) received the Ph.D. degree in electrical engineering from the University of Virginia, Charlottesville, VA, USA, in August 2012.

In August 2017, he joined the Department of Electrical Engineering and Computer Science, Embry-Riddle Aeronautical University, Daytona Beach, FL, USA, where he is currently a Tenured Associate Professor and the Director of the Security and Optimization for Networked Globe Laboratory (www.SONGLab.us). His research has been featured by popular news media outlets, including IEEE GlobalSpec's Engineering360, Fox News, USA Today, U.S. News & World Report, The Washington Times, WFTV, New Atlas, Battle Space, and Defense Daily. His research interests include Internet of Things, cybersecurity, and AI/machine learning/big data analytics.

Dr. Song was a recipient of the Best Paper Awards from CPSCom-2019, ICII 2019, ICNS 2019, CBDCom 2020, WASA 2020, DASC 2021, and GLOBECOM 2021. He has been serving as an Associate Technical Editor for *IEEE Communications Magazine* since 2017, an Associate Editor for IEEE INTERNET OF THINGS JOURNAL since 2020, and IEEE TRANSACTIONS ON INTELLIGENT TRANSPORTATION SYSTEMS since 2021. He is an Editor of eight books, including *Big Data Analytics for Cyber–Physical Systems: Machine Learning for the Internet of Things* (Elsevier, 2019), *Smart Cities: Foundations, Principles and Applications* (Hoboken, NJ, USA: Wiley, 2017), and *Industrial Internet of Things: Cybermanufacturing Systems* (Cham, Switzerland: Springer, 2016). He is the author of more than 100 articles and the inventor of two patents. He is a Highly Cited Researcher identified by Clarivate^{TM} (2021) and a Top 1000 Computer Scientist identified by Research.com. He is a Senior Member of ACM and an ACM Distinguished Speaker.