DSS: Discrepancy-Aware Seed Selection Method for ICS Protocol Fuzzing

Shuangpeng Bai[1,2], Hui Wen[1,2(✉)], Dongliang Fang[1,2], Yue Sun[1,2], Puzhuo Liu[1,2], and Limin Sun[1,2]

[1] School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, China
{baishuangpeng,wenhui,fangdongliang,sunyue0205,liupuzhuo,sunlimin}@iie.ac.cn

[2] Beijing Key Laboratory of IOT Information Security Technology, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100093, China

Abstract. Industrial Control System (ICS), as the core of the critical infrastructure, its vulnerabilities threaten physical world security. Mutation-based black-box fuzzing is a popular method for vulnerability discovery in ICS, and the diversification of seeds is crucial to its performance. However, the ICS devices are dedicated devices whose programs are challenging to get, protocols are unknown, and execution traces are hard to obtain in real-time. These restrictions impede seed selection, thereby reducing the efficiency of fuzzing. Therefore, it has become our primary goal to select a high-quality seed set containing as few seeds as possible with extensive triggered traces.

In this paper, we present a novel automatic seed selection method called DSS, selecting high-quality seeds for improving fuzzing efficiency. The method is based on the observation that dissimilar response messages are generated by different device execution processes in most cases, which helps us build the connection of messages discrepancy and execution traces discrepancy to guide DSS. Expressly, we point out that dissimilar messages are effective indicators of different execution paths. Therefore, choosing ICS messages with high discrepancy as seeds can bring more initial execution traces and fewer seeds with the same semantic, which are essential to black-box fuzzing. Our experiments show that the quantity of seeds selected by DSS is significantly less than the traditional method when achieving the same trace coverage.

Keywords: ICS protocol · Fuzzing · Seed selection

1 Introduction

Industrial Control System (ICS) is a system that combines software and hardware, and is widely used in critical infrastructure, such as critical manufacturing, energy, and other fields. If the security risks are not adequately handled, it will pose a severe threat to the real world. ICS protocol is a set of special rules used for the interaction between industrial software in supervisory control layer

K. Sako and N. O. Tippenhauer (Eds.): ACNS 2021, LNCS 12727, pp. 27–48, 2021.
https://doi.org/10.1007/978-3-030-78375-4_2

and industrial devices in control layer. It works on monitoring remote physical devices' status and controlling remote physical devices. ICS protocols include open protocols and proprietary protocols. The former includes IEC 61850 and Modbus, etc.; the latter includes S7comm and FINS, etc. The ICS protocol has high control authority over the device, but the limited security protection puts the industrial device at risk. In recent years, the frequency and severity of attacks on industrial control systems have increased [18]. For example, national power grids of Venezuela [19] and Ukraine [2] were attacked causing widespread power outages, and the Stuxnet virus [11] attacked Iran's nuclear facilities. These events have shown how essential is the security of ICS devices to the real world. Therefore, ensuring the correct implementation of the protocols in the device is of great significance for protecting critical infrastructure.

Many traditional vulnerability discovery methods (such as static analysis and fuzzing) have achieved good results. However, ICS device security analysis has some restricted conditions, including industrial device programs are challenging to get, protocols are unknown, and execution traces are hard to obtain in real-time. Due to limited conditions, most traditional methods fail, except for black-box fuzzing. However, as a pointless method, black-box fuzzing needs information as a guide to test enough code traces in a limited time.

Rebert [16] points out that the quality of seeds is one of the decisive factors for the effect of mutation-based fuzzing. While more seeds can trigger more internal execution processes of the device, the resulting high cost of computing resources is disproportionate to improved effectiveness. Considering this contradiction, we define high-quality seeds as seeds with small quantities and extensive triggered traces. Accordingly, it is our goal to select high-quality seeds from messages with unknown semantics.

To achieve this goal, we propose DSS, a method to select high-quality seeds for fuzzing proprietary ICS protocols. DSS reduces redundant test cases by excluding seeds with repeated meanings, thereby improving the efficiency of black-box fuzzing. We find that similar messages correspond to similar program execution paths. Conversely, messages with large differences correspond to different program execution paths. Based on this observation, DSS select dissimilar messages as high-quality seeds, which contains different triggered traces.

The experiment shows that our method can select high-quality seeds from ICS messages with repeated meanings. Moreover, when the same amount of execution paths is reached, the quantity of seeds provided by our method is significantly less than the random method, and the proportion is only 0.7% in the optimal situation.

Contributions. In summary, we make the following main contributions.

– We point out that dissimilar messages are effective indicators of different execution paths. This observation provides information to reduce duplicate seeds, thereby reducing similar test cases.

– We propose a seed selection method by analyzing the discrepancy between messages. A small number of seeds with different characteristics are obtained, which can be used as a fuzzing corpus.
– We evaluate the effect of seed selection on Modbus and S7comm protocols. The seeds selected by our method is significantly less than the traditional method.

Roadmap. The remainder of this paper is organized as follows. In Sect. 2, we provide background on the industrial control system, ICS protocol, and related work. Then we propose our approach based on messages discrepancy in Sect. 3. We evaluate our method and verified its effectiveness in Sect. 4. Finally, we introduce how to apply our method to security analysis in Sect. 5.

2 Background

2.1 Industrial Control System

A typical industrial control system includes two parts: industrial software in supervisory control layer and industrial devices in control layer, as shown in Fig. 1. The industrial software sends a network request to the industrial device to control it and requires the industrial device to return information, such as start-stop status, current I/O value. The industrial device executes the instructions sent by the industrial software, converts the request message into a series of operations, and sends the response message back through the network.

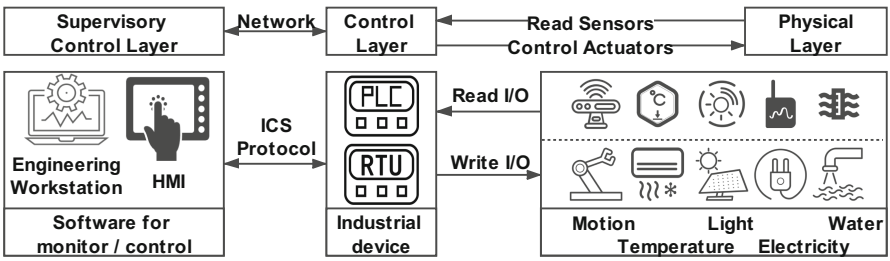

Fig. 1. Industrial control system framework.

The ICS protocol is used for the communication between industrial software and industrial devices, and it specifies the mapping of messages to functions. Apart from the standard protocol defined functions, many industrial manufacturers have expanded their products' functions, which exceed the established scope of the original protocol. Specifically, though some public and standard protocols such as Modbus and IEC 61850 are widely used, many proprietary protocols are proposed for customized features (such as controlling the start

and stop, connection management, file transfer, firmware update). As a result of being defined by various manufacturers separately, these implementations lack the support of well-tested underlying libraries. Furthermore, some manufacturers firmly believe that their products are only used in a network-isolated environment. Therefore, security issues are not considered well in implementation, and the implementation has not undergone adequate safety testing. The intellectualization of industrial control systems requires breaking the isolation of the network. It is possible to expose the communication interface of the industrial device, which brings opportunities to the attacker.

2.2 Obstacles in ICS Protocol Vulnerability Discovery

Industrial device is a kind of dedicated device, so that the vulnerability discovery of ICS devices often faces the following situations: some internal programs of the devices are challenging to get, protocols are unknown, and execution traces are hard to obtain in real-time. These restricted conditions cause traditional software vulnerability discovery methods to fail for industrial devices.

Static Binary Analysis Methods. First, static binary analysis methods, such as static symbolic execution and static taint analysis, analyze binary programs to find vulnerabilities. Some firmware are hard to obtain through official channels, although some ICS manufacturers provide device firmware (such as Schneider). Some even need to be read from flash using JTAG, which is also challenging for industrial device. Second, the firmware needs to be decompressed to get the binary program. Some tools can analyze standard file systems, such as binwalk [7]. However, the extracting difficulty has increased because of the emergence of encrypted firmware and private format firmware (such as the Schneider Modicon series). These problems lead to the inability to guarantee the acquisition of the program in the device. In this case, the static analysis method is not suitable. Besides, path explosion is also one of the limitations of static analysis methods.

Generation-Based Fuzzing. Generation-based fuzzers, such as Peach [4], and Sulley [1], need expert knowledge about protocol information, including field structure division, range of possible values, and data dependence among fields. Some works use automated or manual methods to analyze traffic and reverse the protocol, such as PULSAR [5], which uses the Markov model representing the state machine of the protocol. However, these approaches may introduce new problems. If the protocol reverse is not comprehensive enough, the template's expression ability will be limited, and some input space will be missed. If there is a misunderstanding in the reverse engineering result, it will lead to error accumulation, resulting in many invalid mutations. Comprehensive and accurate protocol reverse requires manual analysis, which leads to the high cost.

Gray-Box Fuzzing. A gray-box fuzzer obtains the program execution paths triggered by the current input in real-time, thereby guiding the mutation with high efficiency. However, because it is difficult to obtain the industrial device program's execution paths in real-time, gray-box fuzzing methods, such as AFL [20], cannot be used directly unless the firmware image is emulated correctly. Some works emulate the firmware of embedded devices, such as Firmadyne [3]. Furthermore, some works combine emulation and fuzzing, such as FirmAFL [22], BaseSAFE [15] and Frankenstein [17]. However, these works are mainly focused on Linux and some specific RTOS systems, but not ICS devices. Emulating some system-independent tasks functions rather than the whole system is simpler and enough for fuzzing. The emulation needs function addresses to hook system functions. But for VxWorks, the commonly used ICS operating system, it is difficult to automatically get system function addresses because of the mixing of task code and kernel code. Therefore, the correct emulation of the ICS device requires manual analysis of the function address.

Some works use path coverage information to guide a gray-box fuzzing, focusing on ICS protocol code libraries. Polar [13] based on static code analysis and dynamic taint analysis technique, locates the function code processing statements and some security-sensitive points. And then use the knowledge of these key locations to guide the fuzzing of the ICS protocol code libraries. Peach* [14] identifies valuable data covering the new code area based on the path information collected during the fuzzing and then constructs a puzzle corpus to optimize the input generation process based on cracked packet pieces. The above methods are based on the execution path coverage for fuzzing. These methods apply to ICS protocol libraries, but not to black-box devices. As a result, coverage-guided gray-box fuzzing is not yet applicable.

Black-Box Fuzzing. Original black-box fuzzer generates test cases by mutating existing messages. Due to its wide application range and low cost, this method is often used in ICS scenarios. Because pointless mutation generates a high percentage of invalid test cases, it is necessary to analyze existing information to improve efficiency.

Some emerging black-box fuzzers for the ICS protocol generate test cases based on learning existing messages. SeqFuzzer [21] extracts format information of the EtherCAT protocol based on deep learning and generates EtherCAT test cases. GANFuzz [8] uses Generative Adversarial Network (GAN) to train a generative model on Modbus protocol data, learn protocol syntax and generate Modbus test cases. The above methods use machine learning to analyze traffic, learn protocol knowledge, and automatically generate test cases. These methods are suitable for black-box fuzzing, but these methods' versatility needs to be evaluated when applied to proprietary protocols. In addition, the above methods have potential problems. When the data distribution is unbalanced, some information hidden in the less frequent messages can be easily missed, although it may represent some essential special functions.

Also, Kim [9] put forward a fuzzing tool for Modbus protocol. It updates the seeds pool during the test in two conditions. The first is that the number of the changed bytes in requests is not equal to that of responses. The second is that response time has a significant change. However, this condition is not applicable in some situations. For example, when reading two registers and eight registers in Modbus, the request has only a one-byte difference in value, but the response changes more. The two messages are considered to have passed different program areas in the method, but they have the same function and execution process.

Table 1. Comparison of vulnerability discovery methods for ICS device

Method	Requirement	Challenge
Static binary analysis	Acquisition of firmware/program	① ②
Generation-based fuzzing	Expert knowledge about ICS protocol	③
Gray-box fuzzing	Acquisition of firmware/program, emulation	① ④
Black-box fuzzing	Messages between software and device	⑤

① Non-public firmware or program. ② Path explosion. ③ High cost of comprehensive and accurate protocol reverse. ④ Difficulty of emulation for ICS device. ⑤ Low efficiency without guidance.

In Table 1 we summarize the vulnerability discovery methods, their requirements when applied to ICS device, and the main challenges.

2.3 Seed Selection for Improving Fuzzing Efficiency

Black-box fuzzer is suitable for ICS test scenarios but lacks information guidance, resulting in low efficiency. Rebert [16] studied the influencing factors of the effect of mutation-based fuzzing, in which the quality of seeds is one of the decisive factors. While more seeds can trigger more internal execution processes of the device, the resulting high cost of computing resources is disproportionate to improved effectiveness. Moreover, compared with traditional software testing, since network-based testing is slow, there are higher requirements for fuzzing efficiency. To ensure that, black-box fuzzers need a small number of seeds that retain high execution path coverage.

Seeds selection in gray-box fuzzers such as AFL tools generates the seed set using software instrumentation technique to obtain the execution traces and use the greedy algorithm to select the optimal set. Since the program is inside the industrial device, it is challenging to obtain instrumentation information except for emulating firmware. Considering that the industrial device emulation is still difficult, but the network traffics are easy to get, it is feasible to select seeds by analyzing the relationship of message and execution trace.

3 Approach

In this section, we describe our observation and method. In Sect. 3.1, we elaborate our observation on the relationship between ICS messages and execution paths. In Sect. 3.2, we propose calculation methods to measure the difference between messages. In Sect. 3.3, we design a seed selection method based on the comparison of message discrepancy.

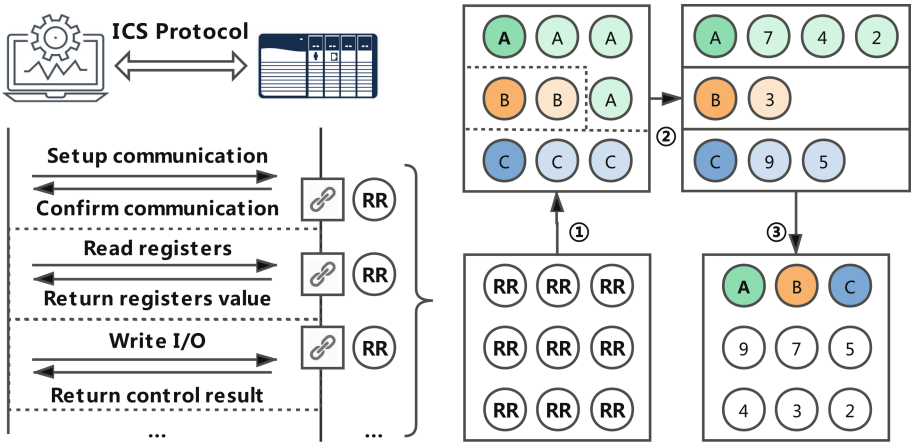

Fig. 2. Discrepancy-aware seed selection method.

In order to facilitate subsequent explanations, the combination of a response message and its request message is named an **RR** (request-response). ①*Selecting Typical RRs.* Choose typical RRs and distribute the others to the most similar typical RR. In the Fig. 2, different letters mean different types. A darker circle represents a typical RR, and the other lighter circles of the same color series are similar RRs of the typical RR. ②*Scoring all RRs.* For each typical RR, score its similar RRs by comprehensively considering the discrepancy between requests and the discrepancy between responses in the same type. In the Fig. 2, the number in the circle means the score of the RR. The larger the number, the lower the similarity within the same type. ③*Sort RRs.* Sort RRs in descending order of score and take out RRs one by one from the list until the expected number is reached.

3.1 Technique Foundation

Key Observation. Our key observation is that dissimilar response messages are generated by different device execution processes in most cases. Our strategy is choosing dissimilar response messages for a greater probability of triggering non-repetitive execution traces based on this observation. It should be noted that

dissimilar response messages are not a sufficient condition for different execution traces. Dissimilar messages may also trigger the same execution process.

In more detail, we divide the situation into the following four categories according to whether the response messages in the ICS protocol are similar and whether the traces corresponding to the messages are the same.

Same Trace with Similar Response. This type includes reading values multiple times, such as reading I/O or time. Except for a small number of changes, the response messages of the same function remain similar. Selecting dissimilar responses can exclude responses with the same function and ensure that the seed set size is small.

Different Trace with Dissimilar Response. This type includes establishing a connection, reading time, reading I/O value, writing I/O value, etc. Different functions correspond to dissimilar responses, which means that this type of response is bound to its function. Selecting dissimilar responses can select messages with different functions, ensuring that more original input execution paths are covered.

Same Trace with Dissimilar Response. This type includes the function of reading a large amount of data at one time, such as reading the value of multiple consecutive addresses or downloading programs from the device. Because the data part in protocol is easy to change and the structure part is relatively stable. Taking Modbus as an example, when the request messages contain different reading bytes, industrial devices will give back responses with different lengths. The similarity of these responses is low, but the execution traces are basically the same. However, in this case, request messages with the same function are similar. Moreover, the dissimilar response is often caused by long messages, which are rare because industrial devices need to ensure real-time performance, and most of the packets are small in length.

Different Trace with Similar Response. This type includes some controlling commands and writing operations. Although the functions are different, the response messages only contain similar and simple confirmation information. In this case, only choosing dissimilar response messages may result in missing response with different functions. However, if there are discernible differences between the request messages of different functions, we can get different execution paths by selecting dissimilar requests. Otherwise, some functions may be ignored, in the rare case where the request message and the received message are both very similar to other messages, but the function is unique.

Statistical Results. We have counted the average execution path differences corresponding to response messages with different similarities. In the S7comm

protocol, the top 20% most dissimilar response messages have an average execution path difference of 58.18%, and the top 20% most similar response messages have an average execution path difference of 29.28%. In the Modbus protocol, the top 20% most dissimilar response messages have an average execution path difference of 69.23%, and the top 20% most similar response messages have an average execution path difference of 33.80%.

3.2 Calculation of Discrepancy

In this part, we propose a normalized discrepancy calculation formula between messages based on text distance. Levenshtein distance [12] refers to the minimum number of editing operations (replacing, inserting, and deleting a character) required to convert two strings from one to the other. The hamming distance [6] refers to the number of different bytes at the same position in two equal-length strings.

First, we propose a general discrepancy calculation method based on text distance.

$$discrepancy = distance(str_1, str_2) \tag{1}$$

Second, $discrepancy_{lev}$ and $discrepancy_{ham}$ are designed based on levenshtein distance and hamming distance. The range of $discrepancy_{lev}$ is between 0 and the maximum length of two strings. Hamming distance requires two strings to be equal in length so that the longer string needs to be curtailed to the same length as the other before calculating. The range of $discrepancy_{ham}$ is between 0 and the minimum length of two strings.

$$discrepancy_{lev} = distance_{lev}(str_1, str_2) \tag{2}$$

$$discrepancy_{ham} = distance_{ham}(str_1, str_2) \tag{3}$$

Third, due to the considerable value of the distance between long messages, our method normalizes the distance to eliminate the influence of long messages. The discrepancy ranges from 0 to 1.

$$discrepancy_{lev} = \frac{distance_{lev}(str_1, str_2)}{len(str_1) + len(str_2) + 1} \tag{4}$$

$$discrepancy_{ham} = \frac{distance_{ham}(str_1, str_2)}{min(len(str_1), len(str_2))} \tag{5}$$

3.3 Seed Selection Method Based on Discrepancy Comparison

We propose a seed selection method based on discrepancy comparison.

Preprocessing. The preprocessing process generates RRs from existing network traffic by completing the following steps. ①Divide traffic into different streams according to IP address and port. ②Remove duplicate request messages and their response. ③If there are a huge number of messages with basically repeated content, these are heartbeat packets used to confirm the survival status of the device in the ICS protocol. These packets are deleted and will not be analyzed later. ④By binding the request message with its response, a series of RRs are obtained for subsequent selection.

Selecting Typical RRs. Our method calculates and compares discrepancy between response messages and constructs a typical-set containing RRs which have typical responses. Each typical RR in the typical-set has a similar-list containing RRs similar to this typical RR. Similar-list will be used to score RRs later.

Algorithm 1: Constructing typical RR set

```
   Input: Total_set containing all RRs, Discrepancy threshold
   Output: Typical_set containing typical RRs
 1 initialize p = first element of total_set;
 2 initialize typical_set = empty set;
 3 add p to typical_set;
 4 p = Next(p);
 5 while p is not the last element of total_set do
 6     initialize q = first element of typical_set;
 7     initialize t = False;
 8     while q is not the last element of typical_set do
 9         d = discrepancy of q.response_message and p.response_message;
10         if d < discrepancy threshold then
11             add p to similar_list of q;
12         else
13             t = True;
14             break;
15         q = Next(q);
16     if t is True then
17         add p to typical_set;
18         initialize similar_list of p = empty list;
19     p = Next(p);
20 return typical_set;
```

In Algorithm 1, add the first RR to the typical-set and traverse the remaining elements. During the traversal, for each element p, calculate the minimum difference between it and all typical-set elements. The difference is calculated by using Eq. 4 in Sect. 3.2. If the difference is greater than the threshold parameter, p is added to the typical-set, and the similar-list of p is initialized to empty.

Otherwise, find the element q in the typical-set that makes the difference smaller than the threshold, and add p to the similar-list of q.

Calculating the Optimal Threshold. In selecting typical RRs, input parameters include a discrepancy threshold used to control the minimum discrepancy of typical RRs' response messages. The minimum discrepancy determines the number of typical RRs. An excessively high threshold will divide RRs with different functions into the same typical RR, causing some omissions of functions. On the contrary, a too low threshold will divide RRs with the same function into different typical RRs, causing repeated selection of the same function.

A smaller threshold will get more typical RRs. ①Use random annealing method to find the threshold that makes the number of RRs approach 100. ②As the threshold decreases by 0.01 each time, Algorithm 1 is called repeatedly, and the number of typical RRs is recorded until the number of typical RRs reaches 300. ③Calculate the proportion of the increase in the number of typical RRs caused by each threshold reduction, and the optimal threshold is the one with the largest increase ratio.

Scoring All RRs. For each typical RR, calculate its minimum difference with other typical RRs as its score. For RRs in similar-lists, our method scores them in Algorithm 2, considering both requesting and responding information.

Algorithm 2: Scoring RRs in similar-lists

Input: Typical_set containing typical RRs, Similar_list of each typical RR
Output: Scores of each RR in similar-lists

```
1  initialize t = first element of typical_set;
2  while t is not the last element of typical_set do
3      initialize similar_list = similar_list of t;
4      initialize p = first element of similar_list;
5      while p is not the last element of similar_list do
6          initialize other_l = similar_list_t without p;
7          p.score_response = min response discrepancy of p and RRs in other_l;
8          p.score_request = min request discrepancy of p and RRs in other_l;
9          p = Next(p);
10     response_l = similar_list sorted in descending order of score_response;
11     request_l = similar_list sorted in descending order of score_request;
12     p = first element of similar_list;
13     while p is not the last element of similar_list do
14         if response_l.index(p) < request_l.index(p) then
15             p.score = p.score_response;
16         else
17             p.score = p.score_request;
18     t = Next(t);
```

For each RR, in Algorithm 2, calculate the minimum discrepancy between the RR and the others in the same similar list. The discrepancy includes the discrepancy between request messages and between response messages. And then, sort the similarity list in descending order to generate request-list/response-list by the minimum discrepancy of request/response messages. Finally, the score is equal to the minimum discrepancy of the request messages if the RR index in the request-list is smaller than the index in the response-list. Otherwise, the minimum discrepancy of the response messages is used as the score. If any of the RR's request or response message is unique, the RR will get a high score through the above steps. Otherwise, our method considers the RR has a low possibility to trigger new execution traces.

Sort RRs. The higher the score of RR is, the more likely it is to be unique. Sort typical RRs in descending order by scores, and do the same for RRs in similar-lists so that RRs with different execution paths are placed first. Concatenate two sorted lists, with typical RRs first, and then our method can take out RRs one by one from the list until the expected number is reached. Choosing more messages can get higher coverage before reaching full coverage. In actual use, the selected number depends on the estimated computing resources and allowable time consumption.

4 Evaluation

4.1 Experiment Setup

Analysis Target. Since the device's program cannot be directly used for analysis, we choose common industrial protocol code libraries for analysis. The public library used here is only to evaluate the effectiveness of our method and does not limit the scope of the application of our method. The ICS protocol library used for the experiment has the requirements, including the library implements a server program, and the server can normally work under instrumentation. For the Modbus protocol, our analysis is based on the unit-test-server provided by the libmodbus library. And for the S7comm protocol, we use the server in the snap7 library.

Legitimate Messages Acquisition. The messages need to match the server software. Otherwise, the requests may look legitimate, but the server cannot parse and respond them correctly. For example, in the Modbus protocol, the used I/O addresses need to be defined in the server; otherwise, an error will be notified. Therefore, many packets that can be correctly parsed by the mentioned software are needed for experiments. To emulate these packets in an industrial control system, we obtain notwork traffic by expanding the original input. ①Perform byte-by-byte mutation on the original industrial control data packet to generate legitimate and illegitimate data packets. ②Send these packets to the server and record the response. ③Mark packets which are parsed as legitimate by Wireshark. Use these marked packets as subsequent candidates, and discard those illegitimate data

packets. The basis for the above operation is that if the device response to our request with a non-abnormal response, it means that the device has correctly executed the request, which also means that the request is legitimate.

Besides, some ICS protocols need to maintain communication status. For example, the S7comm protocol requires "Setup communication" before reading and writing, and the Modbus protocol defines a serial number. For this kind of protocol, we need to send predecessor messages before sending a mutated message to ensure the test case can be parsed correctly. The status may be defined in some protocols but not mandatory to implement in code. For example, the server provided by libmodbus does not handle the serial number; that is, the test case can be sent directly without considering its predecessor data.

For the unit-test-client and unit-test-server provided by the libmodbus library, the experiment mutates the original traffic to expand the execution space. The number of edges before the expansion is 133, and after the expansion is 185, an increase of 38.1%. For the snap7 library, the number of edges changes from 199 to 734, an increase of 268.8%. Table 2 provides a comparison between the original message function and the expanded function. It is worth noting that because the original input "Start upload" function was not implemented in the server, the request for this function returned an error, which was detected by Wireshark and filtered out.

Table 2. Summary of functions under testing

Modbus		S7comm	
Function	Change	Function	Change
Read Coils	Unchanged	Setup communication	Unchanged
Read Discrete Inputs	Unchanged	CPU → Read SZL	Unchanged
Read Holding Registers	Unchanged	Start upload	Disappear
Read Input Registers	Emerge	Read Var	Unchanged
Write Single Coil	Unchanged	CPU → Message service	Emerge
Write Single Register	Unchanged	Write Var	Emerge
Write Multiple Coils	Unchanged	Time → Read clock	Emerge
Write Multiple Registers	Unchanged	PLC Stop	Emerge
Report Slave ID	Emerge	Block → List blocks	Emerge
Mask Write Register	Unchanged	Security → PLC password	Emerge
Read Write Register	Unchanged		

Binary Instrumentation. The execution path triggered by the network request needs to be obtained to evaluate the seed selection method's effectiveness. The instrumentation of ICS protocol libraries is only for the experiment. When faced with real industrial control devices, it is too challenging to achieve.

First, we limit the recorded address to the code segment of the target program. This limitation is to prevent instrumentation of the code of system library

functions. These functions will generate many execution paths with low relevance to ICS protocols, and their security has been extensively studied. If these system library functions are not precluded, irrelevant information will be introduced, which will confuse subsequent analysis.

Second, the industrial control server often has multiple threads working simultaneously, including accepting network requests, running industrial control functions, and sending data packets. Therefore, we instrument all threads and record them separately. For each thread, the execution path is obtained by sequentially recording the basic block's first address.

Third, due to protection measures such as ASLR, the addresses obtained by the instrumentation are random and cannot be directly compared. In order to align these instrumentation results, we process each execution as follows: Take the first address of the first basic block of the first thread as the base address, and record the offset of each address executed to the base address.

Finally, every two consecutive basic blocks' addresses are combined as an **edge**, and all edges executed this time are added to a set. This set represents the code execution path of this message. Remove edges that can be triggered in all messages, which are not related to the function.

Evaluation Setting. As mentioned earlier, each path is transformed into a set, with internal elements as edges. To measure the quality of the selected seeds, we propose edge **coverage** as a measurement standard, which refers to the proportion of edges provided by the selected seeds. The higher the coverage, the more representative the seeds we choose; the lower the coverage, the more functions we have missed.

Besides, we compare the effectiveness of the methods by comparing the number of seeds required between different methods to achieve the same coverage. Use the ratio of the quantity required by Method A to the quantity of Method B as the measurement standard when reaching the same coverage. If the ratio exceeds 1, the performance of method A is better than method B. On the contrary, it shows that the B method is better. When the ratio is close to 1, the performance of the two methods is similar.

4.2 Message Similarity and Trace Similarity

We use the following steps to verify whether dissimilar messages indicate different execution paths. ①Select dissimilar messages by Algorithm 1 and use $discrepancy_{lev}$ as the discrepancy. Giving RR sequences in different orders and random thresholds, many message combinations, whose internal messages are dissimilar to each other, can be obtained. Calculate the number of non-repeated edges corresponding to each combination; the higher the number, the less similar the execution path. ②Randomly select messages and count the number of corresponding edges. Random sampling is performed with different selection numbers, respectively. ③Compare the corresponding edge coverage of dissimilar messages and random messages.

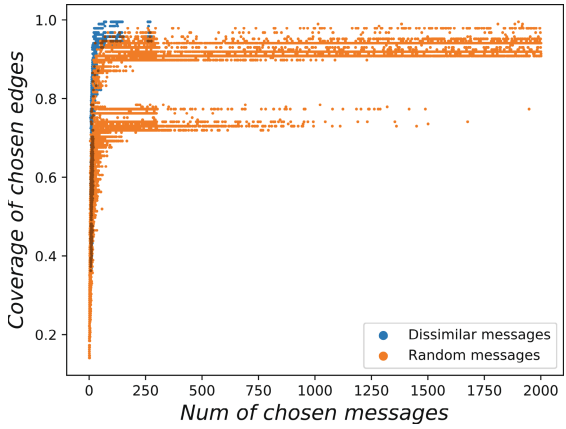

Fig. 3. Coverage of dissimilar messages and random messages in Modbus.

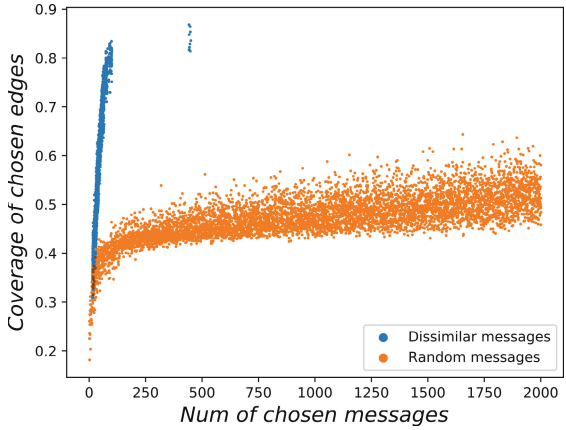

Fig. 4. Coverage of dissimilar messages and random messages in S7comm. (Color figure online)

As shown in Fig. 3 and Fig. 4, dissimilar message combinations correspond to a higher number of edges, meaning divergent execution path combinations are selected. More specifically, because the Modbus protocol is relatively simple, a combination of 250 dissimilar messages is sufficient to correspond to 95% of the execution path. At the same number of messages, random combinations that close to the same execution path just account for a small proportion. When the coverage is the same, the points corresponding to dissimilar packets are clustered on the figure's left edge, which means that fewer packets are needed. In the S7comm protocol, the effect is more significant. When the number of elements of the combination is about 100, the edge coverage of dissimilar packets has reached 80%, while the highest coverage of random packets is less than 50%. Besides, it can be drawn from the figure that it is difficult to obtain high coverage for random messages unless using a vast number of messages.

There is an apparent separation between the blue dots in each figure, and a small part of the dots are in the position where the number of packets is higher than most dots. This separation is because the minimal degree of discrepancy discrimination brings a significant increase in the number of messages in the combination.

Table 3. Average coverage in dissimilar messages and random messages

Modbus				S7comm			
Range	Dissimilar	Random	↑ratio	Range	Dissimilar	Random	↑ratio
0–100	0.81	0.69	16.6%	0–100	0.49	0.36	34.7%
100–200	0.96	0.79	20.4%	100–200	0.80	0.42	89.7%
200+	0.96	0.83	15.6%	200+	0.83	0.45	84.6%

As shown in the Table 3, in each range, the average edge coverage of dissimilar messages is more than that of random messages. Furthermore, due to the slow rise of the *Random method*'s effect, it must choose a disproportionate number of messages to achieve the same effect. In summary, this experiment shows that dissimilar messages are effective indicators of different execution paths.

4.3 Comparing with Traditional Method

A seed selection experiment is performed from the data packets obtained during the experiment preparation process in 4.1. Evaluate the performance of seed selection methods compared with *Random method*.

Seed selection methods include *Hamming method* (based on $discrepancy_{ham}$) and *Levenshtein method* (based on $discrepancy_{lev}$). The *Random method* counts the average, maximum, and minimum values of edge coverage of randomly selected message for 100 times each number. The experiment evaluated the effectiveness of the seed selection method on Modbus and S7comm protocols. The edge coverage here refers to the proportion of edges triggered by the selected seed to the edges triggered by all to-be-selected messages.

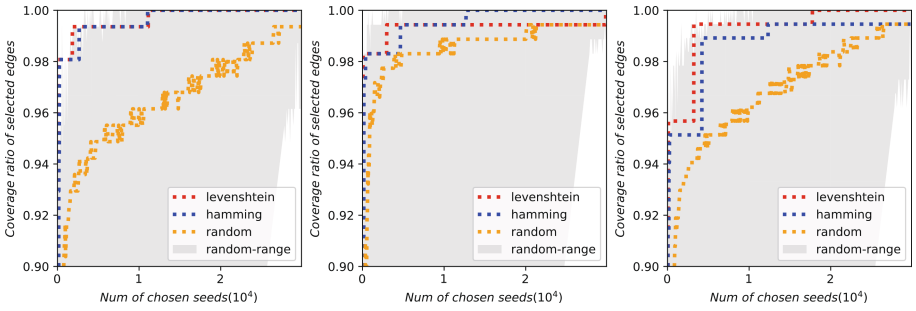

Fig. 5. Edge coverage in Modbus.

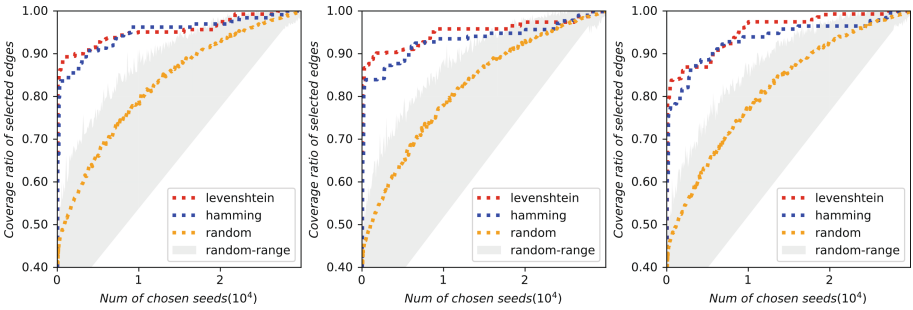

Fig. 6. Edge coverage in S7comm.

Figure 5 and Fig. 6 show that when the number of seeds is the same, the method based on discrepancy comparison can get more edges than the *Random method*, no matter which distance algorithm is based on. Each small picture represents an independent experiment based on different data, showing that the effect is robust.

Table 4. Compare with traditional method in Modbus

	Levenshtein	Hamming	Random
Coverage-0.75	**48**(41,58)	67(27,145)	64(38,94)
Coverage-0.80	**49**(42,59)	107(29,147)	143(81,215)
Coverage-0.85	**49**(42,59)	111(40,147)	314(209,390)
Coverage-0.90	**49**(42,59)	113(44,150)	648(336,838)
Coverage-0.95	**61**(42,96)	201(158,227)	3722(870,5720)

Table 5. Compare with traditional method in S7comm

	Levenshtein	Hamming	Random
Coverage-0.75	**66**(62,70)	197(193,200)	8676(8242,9055)
Coverage-0.80	**146**(64,306)	773(202,1848)	10781(10259,11187)
Coverage-0.85	**797**(188,1940)	2273(1364,2733)	13619(12982,14142)
Coverage-0.90	**3193**(1760,5658)	5188(3548,6043)	17433(17100,17655)
Coverage-0.95	**9039**(8371,9899)	14272(9029,20051)	22371(22238,22485)

The experiment also evaluates the number of seeds required by different methods when the same coverage is reached. Among them, the smaller the number, the better the seed selection performance. As Shown in Table 4 and Table 5, in both Modbus and S7comm protocol experiments, the order of the effects of methods is *Levenshtein method* > *Hamming method* ≫ *Random method*. Specifically, when achieving the same effect, the *Levenshtein method* provides about 60 seeds, which is only 0.76% to 1.64% of the traditional method.

4.4 Comparing with Guiding Method

In this part, we evaluate the effect by comparing the *Levenshtein method* (without protocol information) and the *Guiding method* (using the protocol function code information). The *Guiding method* emulates the situation of obtaining the protocol function code correctly by manual or automatic protocol reverse engineering. Owing to this method is guided by function code information, it is called the *Guiding method*. The *Guiding method* classifies the messages according to the function code specified by Wireshark and then selects messages from different functions each time. It should be noted that this method is only designed for comparison with our method and cannot be directly applied to real situations. As proprietary protocols cannot be parsed in Wireshark, obtaining information about function codes in the real world requires much extra work.

In the Modbus experiment, both *Levenshtein method* and *Guiding method* are significantly better than the *Random method*, and the *Guiding method* is slightly better than *Levenshtein method*. This small gap shows that our method adaptively learns the function code information of Modbus. As shown in Fig. 7, the effect of the *Guiding method* is very significant, indicating that for the Modbus protocol, function code information can distinguish functions well. Under the same function code, there are relatively few different traces. The average number of trace types for each function code is 3.4, so that the information provided by the function code is sufficient for efficient seed selection. The function-code-based *Guiding method* can quickly achieve complete coverage of the original edges.

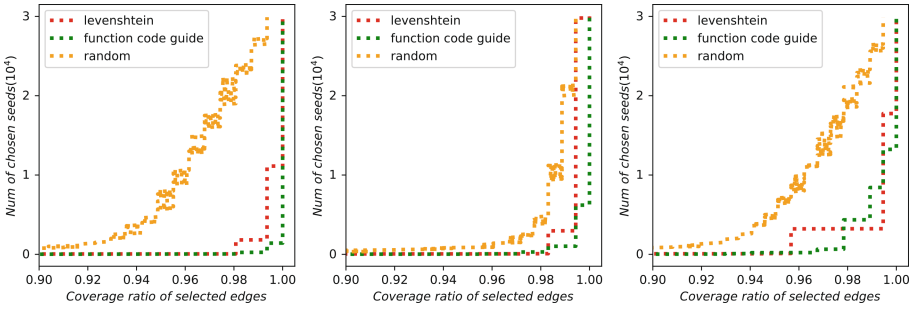

Fig. 7. Comparison of *Levenshtein method* and *Guiding method* in Modbus.

In the S7comm experiment, as shown in Fig. 8, the *Levenshtein method* and *Guiding method* have similar effects when coverage less than 0.7, which significantly better than the *Random method*. But when the coverage is higher, the *Guiding method*'s effect begins to deteriorate until it is close to the *Random method*. However, our method is still significantly better than those methods. This is because the S7comm protocol is more complicated so that the function code provides insufficient information. There are relatively more different traces under the same function code, and the average trace type number of each function code is

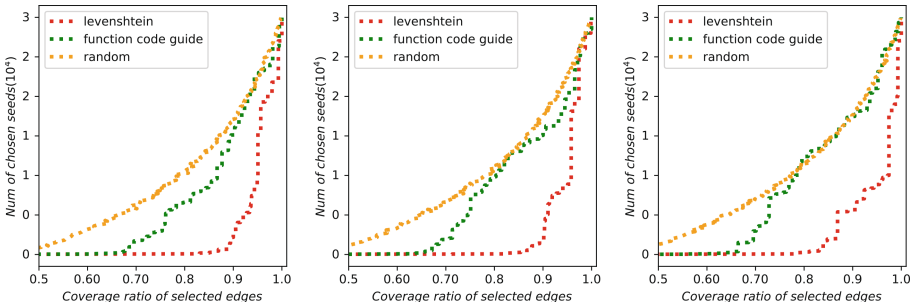

Fig. 8. Comparison of *Levenshtein method* and *Guiding method* in S7comm.

30.8. This leads to the fact that *only using function code information is not enough to select seeds with high coverage*. The experiment also shows that our method can learn more complex protocol information than function codes. Therefore, it is also effective for more complex protocols and has universality for ICS protocols.

5 Discussion

In the previous sections, we propose the seed selection method DSS and evaluated it. In this part, we will introduce how DSS is applied to security analysis. It has been verified that DSS can provide high-quality seeds for mutation-based fuzz testing. Besides, the core algorithm of DSS can also be applied to other proprietary protocol security analysis because of its ability to extract messages with different meanings. We will introduce three application scenarios, including the main application: mutation-based fuzzing, and two other scenarios: test case reduction and protocol reverse engineering.

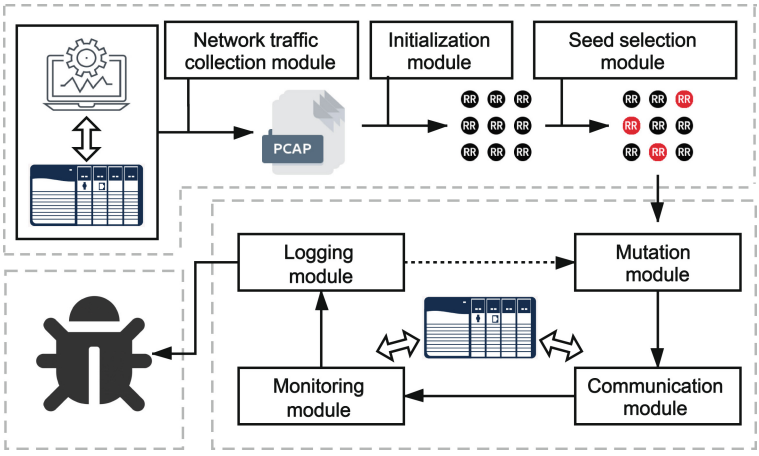

Fig. 9. Framework of fuzzing prototype using seed selection method

5.1 Mutation-Based Fuzzing

To show how our method can be applied to fuzzing, we briefly describe the workflow, as shown in Fig. 9, there are two main steps in fuzzing: selecting seeds and testing a seed.

In selecting seeds stage, there are the following three processes. ①Record the IP, port, and message information of each interaction in real work scenarios in the Network traffic collection module. ②Combine each request message and its response message to generate RRs in the Initialization module. ③Using method elaborated in Sect. 3.3 to select high-quality seeds in the Seed selection module. The detailed steps have been elaborated in Sect. 3.

In testing a seed stage, there are the following four processes. ①Mutate the seed to generate test cases in Mutation module. ②Establish a connection with the industrial device in the Communication module so that the fuzzer can interact with the device. ③Monitor the ICS device's status to determine whether it is working correctly in the Monitoring module. ④Record messages and events during the fuzzing in the Logging module. Since this fuzzer is based on a small number of seeds with high coverage, it will explore more program execution paths in a shorter time, and then efficiently discover vulnerabilities.

5.2 Reduction of Test Cases

Our method can also be used in another security testing process: reduce test cases, shown in Fig. 10. ①Fuzz an ICS device that using a proprietary protocol and record all the messages in this process as test cases. ②Transform these request and response messages to RRs and use the method proposed in Sect. 3.3 to select test cases with different meanings. ③Use these chosen test cases to test other devices using the same proprietary protocol.

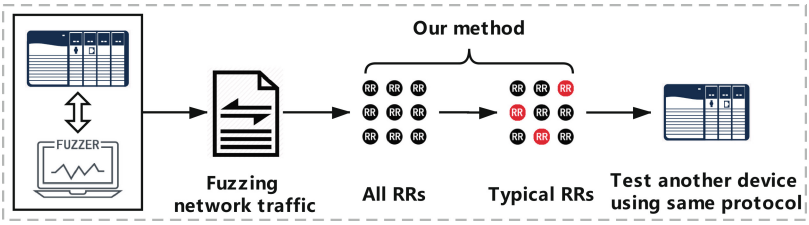

Fig. 10. Framework of reducing test cases

Devices using the same proprietary protocol generally have similar execution processes and similar causes of vulnerabilities. Therefore, if some previously used messages can efficiently test a device's code area, these messages can also efficiently trigger another device's logic, which uses the same protocol. We use Algorithm 1 to select representative messages for testing, reducing many repeated test cases to improve efficiency.

5.3 Protocol Reverse Engineering

There is a step in protocol reverse engineering called message type identification [10], which aims at dividing the message into different categories for next step, analyzing each category's specific protocol format. Our method also has the ability to classify messages, as shown in Fig. 11. The following will introduce how it assists in protocol reverse engineering. ①Convert the messages to RRs. ②Use Algorithm 1 to process all RRs to get the typical-set and the similar-list of each typical RR. ③The classification is completed by treating the packets in the same similar-list as the same type.

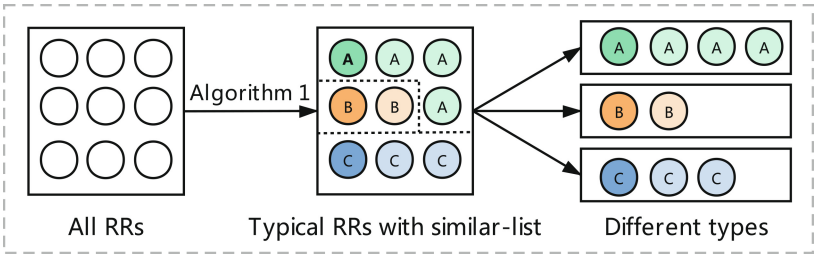

Fig. 11. Application in message type identification for protocol reverse engineering

6 Conclusion

We introduce DSS, a discrepancy-aware seeds selection method for ICS protocol fuzzing. DSS compares ICS messages to determine whether they trigger the same execution path, thereby selecting a high-quality seed set containing a small number of seeds but obtaining high edge coverage. The DSS is suitable for black-box industrial devices and proprietary protocol scenarios, where many methods cannot be applied.

When achieving the same trace coverage, the seeds selected by the *Levenshtein method* is significantly less than the traditional *Random method*, and the proportion is only 0.7% in the optimal situation. Fuzzing based on the high-quality seeds selected by the *Levenshtein method* can test core code traces in a limited time.

Acknowledgement. This paper is supported by the science and technology project of State Grid Corporation of China: "Research on 5G Electric Power security protection system and key technology verification" (Grant No. 5700-202058379A-0-0-00).

References

1. Amini, P., Portnoy, A.: Sulley fuzzing framework (2010)
2. Case, D.U.: Analysis of the cyber attack on the Ukrainian power grid. Electricity Information Sharing and Analysis Center (E-ISAC) 388 (2016)

3. Chen, D.D., Woo, M., Brumley, D., Egele, M.: Towards automated dynamic analysis for linux-based embedded firmware. In: NDSS, vol. 16, pp. 1–16 (2016)
4. Eddington, M.: Peach fuzzing platform. Peach Fuzzer **34** (2011)
5. Gascon, H., Wressnegger, C., Yamaguchi, F., Arp, D., Rieck, K.: Pulsar: stateful black-box fuzzing of proprietary network protocols. In: Thuraisingham, B., Wang, X.F., Yegneswaran, V. (eds.) SecureComm 2015. LNICST, vol. 164, pp. 330–347. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28865-9_18
6. Hamming, R.W.: Error detecting and error correcting codes. Bell Syst. Techn. J. **29**(2), 147–160 (1950)
7. Heffner, C.: Binwalk: firmware analysis tool (2010). https://code.google.com/p/binwalk/. Visited 03 Mar 2013
8. Hu, Z., Shi, J., Huang, Y., Xiong, J., Bu, X.: GANfuzz: a GAN-based industrial network protocol fuzzing framework. In: Proceedings of the 15th ACM International Conference on Computing Frontiers, pp. 138–145 (2018)
9. Kim, S., Cho, J., Lee, C., Shon, T.: Smart seed selection-based effective black box fuzzing for IIoT protocol. J. Supercomput. **76**, 1–15 (2020)
10. Kleber, S., Maile, L., Kargl, F.: Survey of protocol reverse engineering algorithms: decomposition of tools for static traffic analysis. IEEE Commun. Surv. Tutorials **21**(1), 526–561 (2019). https://doi.org/10.1109/COMST.2018.2867544
11. Langner, R.: Stuxnet: dissecting a cyberwarfare weapon. IEEE Secur. Priv. **9**(3), 49–51 (2011)
12. Levenshtein, V.I.: Binary codes capable of correcting deletions, insertions, and reversals. Soviet Physics Doklady **10**, 707–710 (1966)
13. Luo, Z., Zuo, F., Jiang, Y., Gao, J., Jiao, X., Sun, J.: Polar: function code aware fuzz testing of ICS protocol. ACM Trans. Embed. Comput. Syst. (TECS) **18**(5s), 1–22 (2019)
14. Luo, Z., Zuo, F., Shen, Y., Jiao, X., Chang, W., Jiang, Y.: ICS protocol fuzzing: coverage guided packet crack and generation. In: 2020 57th ACM/IEEE Design Automation Conference (DAC), pp. 1–6. IEEE (2020)
15. Maier, D., Seidel, L., Park, S.: BaseSAFE: baseband sanitized fuzzing through emulation. In: Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks, pp. 122–132 (2020)
16. Rebert, A., et al.: Optimizing seed selection for fuzzing. In: 23rd USENIX Security Symposium (USENIX Security 14), pp. 861–875 (2014)
17. Ruge, J., Classen, J., Gringoli, F., Hollick, M.: Frankenstein: advanced wireless fuzzing to exploit new Bluetooth escalation targets. In: 29th USENIX Security Symposium (USENIX Security 20), pp. 19–36 (2020)
18. Slowik, J.: Evolution of ICS attacks and the prospects for future disruptive events. Threat Intelligence Centre Dragos Inc. (2019)
19. Vaz, R., et al.: Venezuela's power grid disabled by cyber attack. Green Left Weekly (1213) 15 (2019)
20. Zalewski, M.: American fuzzy lop (2014)
21. Zhao, H., Li, Z., Wei, H., Shi, J., Huang, Y.: SeqFuzzer: an industrial protocol fuzzing framework from a deep learning perspective. In: 2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST), pp. 59–67. IEEE (2019)
22. Zheng, Y., Davanian, A., Yin, H., Song, C., Zhu, H., Sun, L.: FIRM-AFL: high-throughput greybox fuzzing of IoT firmware via augmented process emulation. In: 28th USENIX Security Symposium (USENIX Security 19), pp. 1099–1114 (2019)