The Journal of Supercomputing
https://doi.org/10.1007/s11227-020-03245-7

Smart seed selection-based effective black box fuzzing for IIoT protocol

SungJin Kim[1] · Jaeik Cho[2] · Changhoon Lee[3] · Taeshik Shon[4]

Abstract
Connections of cyber-physical system (CPS) components are gradually increasing owing to the introduction of the Industrial Internet of Things (IIoT). IIoT vulnerability analysis has become a major issue because complex skillful cyber-attacks on CPS systems exploit their zero-day vulnerabilities. However, current white box techniques for vulnerability analysis are difficult to use in real heterogeneous environments, where devices supplied by various manufacturers and diverse firmware versions are used. Therefore, we herein propose a novel protocol fuzzing test technique that can be applied in a heterogeneous environment. As seed configuration can significantly influence the test result in a black box test, we update the seed pool using test cases that travel different program paths compared to the seed. The input, output, and Delta times are used to determine if a new program area has been searched in the black box environment. We experimentally verified the effectiveness of the proposed.

Keywords Fuzzing test · CPS · IIoT · Vulnerability analysis

1 Introduction

With the advent of Industrial Internet of Things (IIoT) devices in industrial control systems, the connections and hence the communication among various cyber-physical system (CPS) components have rapidly increased. This has vastly improved the information exchange, as well as the productivity of manufacturing systems.

✉ Taeshik Shon
tsshon@ajou.ac.kr

[1] Department of Computer Engineering, Ajou University, Suwon, Korea

[2] Security Division, IBM MEA, Dubai, United Arab Emirates

[3] Department of Computer Science and Engineering, Seoul National University of Science and Technology, Seoul, Korea

[4] Department of Cyber Security, Ajou University, Suwon, Korea

Published online: 14 March 2020

However, as a result of the increase in the network connections, the malicious attack surface has also expanded and the number of incidents targeting the CPSs has increased. Furthermore, ever since Stuxnet, which allegedly crippled Iran's nuclear program, was uncovered, attacks against industrial control systems have become more sophisticated. For example, the 'Crashoverride' malicious code which caused a power outage in Kiev city of Ukraine in December 2016 generated a dedicated protocol traffic in the target environment and rendered the target systems disabled by sending a flood of fake messages. One interesting feature of this attack was that this malware impersonated normal traffic to avoid network monitoring. All these actions of Crashoverride exploited a one-day vulnerability of its target product [1, 2]. Subsequently, more sophisticated attacks have taken place on CPSs [3]. Therefore, the need for the cyber security of industrial control systems, all over the world, cannot be overemphasized.

Currently, a number of research studies on the topic of coping with sophisticated malware attacks on IIoT are available. Although various techniques, such as intrusion detection and intrusion prevention systems for IIoT, have been proposed, many of them are not applicable to real systems owing to one primary characteristic of IIoT, namely its emphasis on being available [4]. The cost of installation of the IIoT devices is too high to replace or upgrade them, even if vulnerabilities are uncovered. For this reason, most vendors release their IIoT products after conducting fuzzing tests as part of their vulnerability analyses. However, new vulnerabilities continue to be discovered every day.

To address this problem, studies for developing more effective analysis methods to tackle newly discovered vulnerabilities of IIoT are being conducted. Most studies use white box or gray box-based testing to generate optimized test cases. However, these testing tools suffer from a disadvantage in that they need significant amount of time for the pre-analysis of each device or software that is likely to face the vulnerabilities. Most of the CPS security managers do not have access to the source code of the IIoT devices. For this reason, a white box-based testing is not feasible. Furthermore, these devices have various firmware versions from diverse manufactures. The gray box testing is not practically viable either. Thus, a black box-based testing is most suitable for actual IIoT needs.

In this study, we present new test evaluation parameters which can optimize the black box-based fuzzing tests for the IIoT protocol. After verifying our approach through experiments, we propose a technique that not only minimizes the pre-analysis time by improving the seed selection process of the IIoT protocol fuzzing test, but also enhances the efficiency of the black box test by quickly generating effective test cases. The key contributions of our work toward improving the IIoT security are as follows.

- A new protocol fuzzing framework is developed, which updates the seed pool for detecting a new program area in a black box-based test.
- Some parameters are proposed, which act as indexes to check whether a new program path is found.

- The efficiency of the proposed protocol fuzzing framework is verified through 'line coverage' and 'branch coverage' by comparing it with traditional black box technique.

The rest of the paper is organized as follows: Sect. 2 discusses related research and basic knowledge. Section 3 introduces the proposed black box fuzzing framework, and Sect. 4 discusses the details of the proposed technique and its validity. Finally, Sect. 5 presents some conclusions and discusses possible future research directions.

2 Background

2.1 Vulnerability analysis techniques

The fuzzing test, which is one of the well-known vulnerability analysis techniques, is a random test method which causes an abnormal state of the software by injecting an abnormal input values into the software system. The purpose of this test is to find vulnerabilities in protocol implementation and has been the focus of numerous studies conducted in recent times [5–10]. In addition, studies focused on symbolic execution for efficient test case generation have been also carried out [11–15]. Although these studies can derive optimal test cases by analyzing the target software, it is difficult to practically perform white box or gray box testing in real environments, where each device or software may have originated from a different manufacturer. Moreover, the firmware version for each device or software could be different as well. Therefore, it is exceedingly difficult for the administrators who actually manage the industrial control systems to use these methods to check the security of their systems. In this context, Shapiro's work [16] and 'netzob' [17] are the best choices because they can be performed with minimal knowledge. As the current study does not need to analyze the specifications of the protocol, it is expected to be effective during its operation in an IoT environment. However, since the accuracy of automatic protocol reverse engineering is not adequate enough to distinguish between the fields of the protocol, additional studies are needed.

Currently, test cases are usually generated based on the target protocol standard. Numerous research studies were conducted for the IIoT protocol vulnerability analyses, based on their protocol specifications. Peng used a technique to verify the boundary values prior to randomly generate the input values in the 'Modbus' protocol vulnerability analysis [18]. SungJin Kim analyzed the vulnerability of the manufacturing message specification (MMS) protocol with nested structures [19]. His technique has the advantage of being able to generate test cases, based on the target protocol specification, to quickly conduct vulnerability analysis of multiple devices. On the flip side, this technique may generate inefficient test cases as it does not consider the operation of the target system.

Recent fuzzing studies have used test case evaluation indexes that consider the behavior of programs, for example, their code coverage, to derive efficient test cases. However, most of the current evaluation indexes are applicable only to white box and gray box testing. Therefore, a new evaluation index is needed for the black box

test. The evaluation indexes for the vulnerability analysis tests that various studies currently refer to are as follows.

2.2 Evaluation indexes of vulnerability analysis

The effectiveness of a test for vulnerability analysis can be assessed by vulnerability analysis evaluation indexes. These indexes include (1) method coverage, which indicates how many of the functions of the target software have been performed, (2) line coverage, which indicates how many source code areas have been executed, and (3) branch coverage, which shows the degree of execution of a branch statement [20]. Recent studies have emphasized on achieving high code coverage with fewer test cases. Particularly, some indexes continuously check the code coverage during the testing and use that information to create new test cases. A brief explanation of these indexes is provided below, along with their advantages and disadvantages.

2.2.1 Method coverage (function coverage)

The method coverage index indicates how many functions (methods) of the target software have been tested. To measure this index, a binary analysis of the source code of the target software is needed. Hence, this index is not suitable for a black box-based test. Another issue with this index is that, if a method consists of several lines of source code, it may jump to the conclusion that the method has been executed even if only a few lines of the source code in that method are executed. 'Skyfire fuzzer' analyzed the results using both the line coverage and function coverage to validate the effectiveness of this index and confirmed that line coverage has a better bug detection probability than the method coverage [13].

2.2.2 Code coverage—line coverage

Line coverage (L-Cov) is the most common vulnerability analysis index that evaluates how many lines a test case has passed. Because it is practically difficult to know where vulnerability exists in the target software source code, line coverage is the most intuitive evaluation index, which many studies have used to demonstrate the performance of their proposed techniques [21, 22]. However, line coverage has a disadvantage in that it does not consider any vulnerability that exists only in a particular branch.

The mutation-based tests, such as 'Seelix,' use line coverage for applying a technique that creates the next input value by considering the increased coverage of the target [23]. Code coverage-based seed selection method has been used to improve the overall test efficiency in [12]; however, such methods still inherit the problems of line coverage.

2.2.3 Code Coverage—branch coverage

Branch coverage (B-Cov), similar to line coverage, is the most commonly used evaluation index in literature [24, 25]. As this index evaluates how many branching statements

are executed among all of the branching statements of a target program, it may indicate how many scenarios have been checked during the execution of the program. In particular, it is considered a better index than line coverage because it can determine how many different software error handling statements are passed through. However, both line coverage and branch coverage are available only in white box testing.

'Kameleon Fuzz,' published in 2014, is a black box fuzzing tool that automatically generates test cases using genetic algorithms (GA) for finding the XSS vulnerability [26]. It uses 'new page' and 'macro-state' as part of the GA fitness function parameter to develop test cases to increase application coverage. Thus, this tool increases the efficiency of test cases by using parameters related to the program path. These parameters can be used in a black box test, where only limited information can be used. However, information on these parameters (i.e., new page and macro-state) is difficult to collect in black box testing of the IIoT devices, owing to the highly optimized embedded facilities in such devices. Thus, the IIoT requires a comprehensive black box testing technique that can overcome the aforementioned limitation. In this study, to achieve this objective, we analyzed the factors that have positive correlation with code coverage and can work in a complete black box environment.

As mentioned above, the recent vulnerability analysis studies focused on improving the efficiency by generating test cases using binary dynamic analysis techniques. In some cases, the efficiency of the test case was maximized by utilizing the indicator used as the performance measure. However, most of these techniques are available only for white box or gray box testing. Even if there are some studies that have used similar approaches in a black box testing environment, such tests cannot be considered as completely black box-based. Therefore, in this study, we propose a new smart seed selection-based fuzzing test framework that addresses the problems of black box-based testing by finding the test evaluation indexes that can be used in black boxes and adding seeds based on them.

3 Smart seed selection-based fuzzing

Before discussing the proposed black box testing technique, let us consider the probability of finding vulnerability in each source code area. Is it possible to classify the source code areas into those that have high probability of finding vulnerabilities and those that do not. Furthermore, the probability of finding vulnerability in a source code area, where a widely used simple method is implemented, is lower than that in areas where less familiar methods are implemented. If access to the source code is granted, it is possible to distinguish the areas that have a high probability of finding vulnerabilities. However, in a black box test, this is not the case.

A black box-based vulnerability analysis mainly generates test cases by applying an 'attack grammar' to seed and inject test cases into the target system as an input value. If the composition of a seed pool is not ideal, it may result in only some area of the program being tested. Irrespective of how good the attack grammar is, if the initial seed configuration is wrong, it searches only specific areas of the program. Thus, the probability of finding vulnerabilities is extremely low. To solve this

problem, we propose the following fuzzing test framework, shown in Fig. 1. Each of the steps in the framework is described below.

- Step 1. Create seed pool

In the generation of seed-based test cases, seed pool is the set of normal and abnormal inputs that need to be prepared before the attack grammar is applied. If the composition of the seed pool is not ideal, the test cases may pass through only a limited area of the software, and this can have a bearing on the test result. In this study, we propose a method that updates the seed pool even during the test.

- Step 2. Select a new seed from the seed pool

In the seed pool, a seed that will be used to create the test case is selected. It should be ensured that the selected seed can generate more test cases according to the attack grammar. If all the attack grammar has been applied to the seed, then another seed should be selected.

- Step 3. Generate a test case and execute

In this step, the fuzzer creates a test case by applying the attack grammar to the selected seed. For example, the attack grammar changes some field of the seed to a

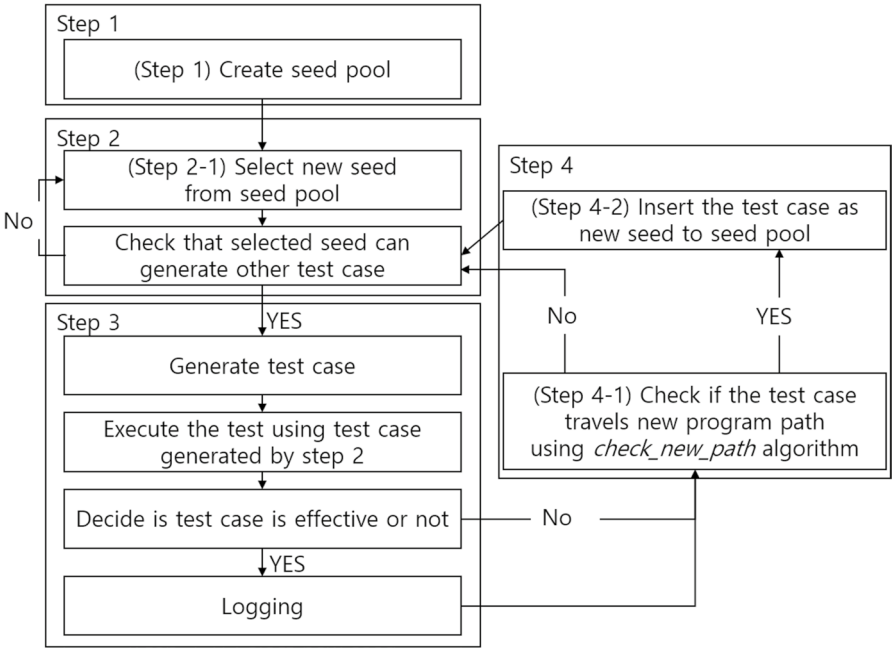

Fig. 1 Schematic of the proposed fuzzing framework

random value. Subsequently, the fuzzer enters this test case into the device under test (DUT) and monitors the target's reaction to the test case. The monitored result is saved as a log. If it is not practical to monitor memory leaks or software crashes on the target system, the fuzzer only observes the output of the target system, such as the response time and output. The detailed process information, e.g., memory leak and software crash, provides a strong clue of a vulnerability; however, it is hard to capture the same in industrial control systems.

- Step 4. Check how the test case travels through the software

Finally, step 4 is verified if the test case has examined a new area or not. If the test case has passed through a new area, it is added to the seed pool as a new seed. The difference between the proposed framework and the other existing black box-based vulnerability analysis tools is that here, the seed is added during the experiment to expand the program areas to be examined. For this iteration, checking if the test case has passed a new program area is crucial. In general, white box-based tests can use indexes, such as line coverage and branch coverage. However, as we have seen in Sect. 2, it is not possible to use those indexes in black box fuzzing. Hence, in this study, we use a new method to determine whether the test case searches a new program area or not, based on the input value, output value, and the consumed time (Delta time), as an alternative to line and branch coverages. The program area search inference method is performed as shown below in Fig. 2.

4 Evaluation

As mentioned earlier, code coverage cannot be used as an index in a black box-based fuzzing test. Hence, in this study, we have implemented another indicator that replicates the functionality of code coverage index.

The new indicators used in this research are the input, output, and Delta time. These can fill in the role that code coverage plays in a white box test. In this section, we discuss the measurement of these indexes and experimentally prove their effectiveness. Additionally, we have also conducted a comparative analysis of our approach versus the Modbus protocol, which is mainly used in IIoT devices [27, 28], using a randomly selected test case. For the convenience of analyzing the results, a Modbus server and a Modbus client using Ubuntu 18.04 were built using a virtual machine. In the virtual machine environment, both the server and the client were set to have 1 GB of memory in a single processor in consideration of the low power and low-performance the IIoT environment.

4.1 Input/output

Owing to the characteristics of the protocol fuzzing, the output has a strong relationship with the input in most cases. When the vulnerability analysis is performed for the server, the test device (TD) sends a request message; the DUT processes the request

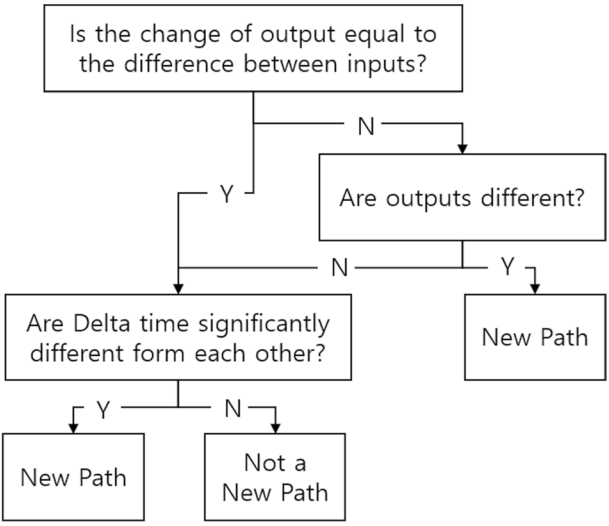

Fig. 2 Basic idea of checking new program path

message, and transmits the response message to the TD. On the contrary, when the DUT is a client, the response to the normal response message is not transmitted by the TD. However, most of the protocols are designed to send an error message when the client receives an abnormal response. Therefore, the input value and the output value are closely related to the vulnerability analysis of the target system.

The change in the output value corresponding to a change in the input value can be largely classified into three types: In the first type, some of the input fields are used directly in a particular field of the output. In the second, the output is the same even if the input changes. In these two cases, it is not possible to know whether other program areas are executed or not. However, it can be deduced that the output is generated in the same source code area. In contrast to the above two types, in the third, the changes in the inputs and those in the output are significantly different. In this case, it can be clearly determined that the output is generated in a new program area. To distinguish these three cases, we calculate the dissimilarity using the following method.

$$\begin{aligned}&\text{Dissimilarity(Seed, Test case)}\\&= \text{Dissimilarity}\left(\left(\text{in}_s, \text{out}_s\right), \left(\text{in}_{tc}, \text{out}_{tc}\right)\right)\\&= \frac{\sum \left|\text{out}_{s,i} - \text{out}_{t,i}\right|}{\sum \left|\text{in}_{s,i} - \text{in}_{t,i}\right|}\end{aligned}$$

The dissimilarity is the ratio of the difference between the input values of the seed and the test case to the difference between the output values of the seed and the test case. These differences are calculated in terms of byte units because most of the IIoT protocols distinguish fields in terms of bytes. If the fields are of different lengths, zero padding is applied to the short length field and the calculation is performed.

The dissimilarity is 0 if and only if the input and the output values are independent of each other. It is 1, if the output value changes in proportion to the input value. On the other hand, if the input value and the output value are quite different from each other, the dissimilarity is proportional to their difference.

Table 1 validates the effectiveness of the proposed dissimilarity. It is a part of an experiment that involves changing the address field and the value field based, on the normal *write single register* function.

Based on the results obtained by checking the operation of the open source libmodbus, Test1 and Test2 have the same function call as the seed, while Test3 has been confirmed to operate the error handling function when accessing an illegal address. Similarly, each test has been confirmed to have dissimilarity values of 1 or greater than 1. In the normal case, 1 is the output, and in the case of an abnormal address access, the output is greater than 1.

Figure 3 shows the function call graph that yielded the experimental results shown in Table 1. In the case of Test1 and Test2, the message is transmitted directly from the function *_modbus_tcp_repare_response_tid*. In the case of Test3, it is confirmed that a message is transmitted after calling the functions for error handling. Thus, it is experimentally feasible to determine whether different code areas have been executed through the difference between the input and the output.

4.2 Delta time

After checking the dissimilarity using the input and output, the Delta time-based program path classification is performed for test cases that are determined to have not passed the new program. It is clear that if the time to search the program area is different, it may be inferred that the search has happened in another area. However, Delta time may be affected by noise occurring due to OS scheduling, network jitter, etc.

Table 1 Dissimilarity of Modbus write function test

Input	Func	Address	Value	Dissimilarity
Seed				
In	0 × 06	0 × 53	0 × 00a3	–
Out	0 × 06	0 × 53	0 × 00a3	
Test1				
In	0 × 06	0 × 6b	0 × 00a3	1
Out	0 × 06	0 × 6b	0 × 00a3	
Test2				
In	0 × 06	0 × 53	0 × 1124	1
Out	0 × 06	0 × 53	0 × 1124	
Test3				
In	0 × 06	0 × a3	0 × 00a3	174
Out	0 × 86	0 × 02	–	

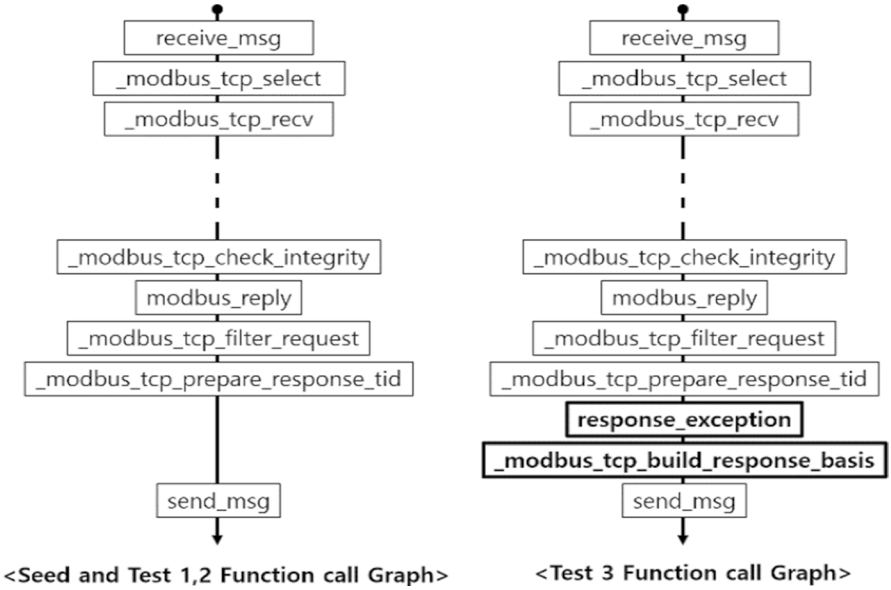

Fig. 3 Function call graph of libmodbus write function test

To solve this problem, we use the law of large numbers. The theorem states that the average of a randomly selected sample converges to the mean of the whole population as the sample size increases. The error is assumed to follow Gaussian normal distribution [29]. Here, as the number of iteration tests increases, the average value of the actual Delta time can be obtained as follows.

$$\text{Let}, X_i = \text{Delta Time}_i + e_i, e_i \sim \text{gaussian normal distribution}$$
$$E(X) = E(\text{Delta Time}) + E(e) = E(\text{Delta Time})$$
$$\text{Therefore}, E(\text{Delta Time}) \approx \text{AvgDelta}_{\text{Real}}, \text{if } n \text{ is large enough}$$

Delta time hardly changes because the same operation is performed by the same function call. Therefore, X also follows the Gaussian normal distribution because the changes in Delta time are limited. Therefore, Avg (X) is close to the mean Delta time, if n is large enough. Twenty-five iterations are not enough to the law of large numbers; however, the sample mean is very close to the population mean of X because the test environment is very static. Therefore, in this study, we measure Delta time through 25 iterations. The experimental results of the actual operations are shown in Fig. 4.

In the experiment, the acquisition of Delta time is based on the CPU clock of the TD and the data are generalized by dividing the clock into a large number of intervals for usability. It is confirmed that the data fluctuate largely according to the noise in the experiments. Noise can be caused by various factors depending on the network environment and the host environment. In this experiment, since

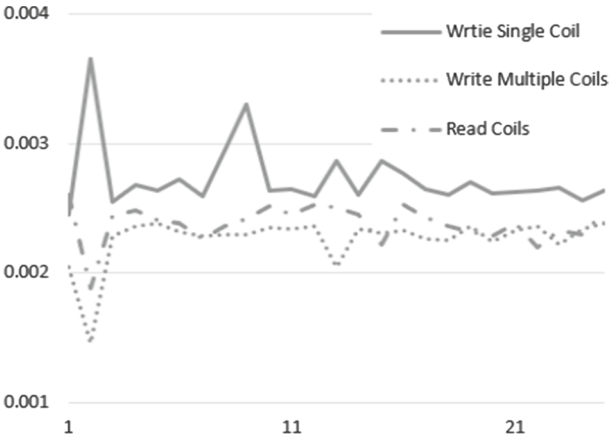

Fig. 4 Delta time of normal operations

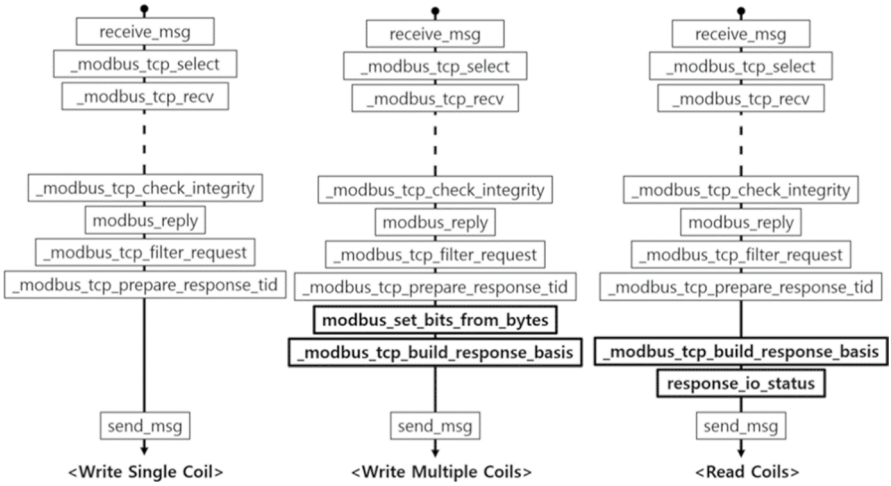

Fig. 5 Function call graph of read and write coil operations

the environment has been constructed using a virtual machine, there is a possibility that the noise is caused by various processes of the host operating the virtual machine. However, despite this high noise, the averages of the experiments show large enough differences to be able to distinguish the experimental results, which are 0.002725 for write single coil, 0.002263 for write multiple coil, and 0.002378 for read coils. Figure 5 shows the function call graph obtained by analyzing the test results in Fig. 4. The function call graph, Fig. 5, shows that the coil read and write operations call different functions. These function calls are reflected in the Delta time. Therefore, the Delta time is an effective indicator to determine if the

test has passed through a new program path or not. The algorithm for checking if a new program path is passed through, based on dissimilarity and Delta time, is as shown below.

Algorithm: The algorithm for checking the traversal of a new program path

```
Func: Check_new_path(s_i, s_o, s_dt, tc_i, tc_o, tc_dt)
//s: seed, i:input, o:output, dt:Delta time, tc:test case and N is a large enough number
1   rep, avgDT ← 0
2   tempDT ← Zero(N)
3   Dissim ← Dissimilarity(s_i, s_o, tc_i, tc_o)
4   if (Dissim is zero or one)
5       while (rep < N) do
6               inject_to_DUT(tc, tempDT[rep])
7               rep ++
8       end while
9       avgDT ← average(rep)
10      if (S_dt – avgDT  >  threshold)
11              return new_path
12      else
13              return travelled_path
14      end if
15 end if
16 return new_path
```

In summary, the dissimilarity check determines whether a new program area has been passed through, using the input and output, while the inspection is performed using the Delta time. Delta time-based decision is performed only when it is difficult to determine the above using the dissimilarity. For the Delta time determination, the number of iterations should be large enough for the law of large numbers to be applicable. If the change in the output is not related to the change in the input or the Delta time is significantly different, it is inferred that the test case has passed through a new path in the code. Furthermore, with the proposed technique, it is possible to perform the test even if the initial seed pool configuration is erroneous. As the test execution time passes, the possibility of searching the un-scanned program area increases.

4.3 General result—code coverage

To demonstrate the effectiveness of the proposed method, we compared the same with a traditional black box-based test. The test target was a ‘Modbus’ slave (Server), and the test was conducted in the environment mentioned earlier. The seed pool consisted of normal traffic of write single coil which is one of the simplest Modbus functions. The attack grammar consisted of three types, namely injecting random value into (1) a function code field, (2) an address field, and (3) a value field. Since Modbus is a simple request–response protocol and not connection oriented, the

Table 2 Comparison of proposed and traditional techniques

Test	L-Cov (%)	B-Cov (%)	Additional seed	Effective additional seed
Proposed	50.0	62.5	5	4
Traditional	46.43	50.0	–	–

communication sequence is not considered. The test results are compared using the code coverage and branch coverage indexes by 'gcov' [30].

The traditional black box test used same attack grammar; however, it does not update seed pool. The proposed method can be applied to many fuzzing tools without alteration of the main framework. Therefore, we focused on verifying the effectiveness of proposed idea. The test results are shown in Table 2.

Based on the basic black box test, the line coverage was 46.43% and B-cov was 50.0%. On the other hand, when the proposed test method and the same seed pool were used, the line coverage and B-Cov were improved to 50.0% and 62.5%, respectively. Thus, compared to the existing black box techniques, the branch coverage and the line coverage were 12.5% and 3.57% higher with the proposed method. Among the new seeds that have been added, 80% have actually passed new program areas. Therefore, the effectiveness of adding new seeds was also confirmed.

5 Conclusion

In this study, we have proposed a new fuzzing framework and a black box test based on the input, output, and Delta time. Since the proposed framework has no direct dependence on the other measures discussed in this paper, it is possible to perform a more efficient vulnerability analysis by applying the same with those measures, if they are available in the future in black box-based tests. In this study, for example, we have used a simple Euclidean distance comparison for input and output dissimilarity. However, we can obtain better results by using the binary distance according to the protocol field [31].

In Sect. 4, the effectiveness of proposed idea was demonstrated. With regard to the time required for testing, the proposed algorithm is not yet optimized. The Delta time-based inferencing logic consumes significant amount of time, and most of the test cases traverse through this logic. To improve the testing time, hypothesis test-based algorithm may be created in future. As the Delta time follows the normal distribution, calculations can be carried out with 95–99% confidence interval for each seed. In future, a test case can be determined as a new seed if and only if the Delta time of the test case is not in the confidence interval of the seed.

The effectiveness of the proposed indicators was verified by Modbus protocol which is widely used in the IIoT environments. It was confirmed that the dissimilarity and the Delta time worked well in our experiments, which have shown that the proposed scheme achieved higher code coverage with the same seed when compared to the existing black box tests.

Acknowledgements This research was supported, in part, by the Basic Science Research Program (Grant No. 2018R1D1A1B07043349) and, in part, by the Energy Cloud R&D Program (Grant No. 2019M3F2A1073386), both through the National Research Foundation of Korea (NRF), funded by the Ministry of Science, ICT and Future Planning.

References

1. Anton Cherepanov, WIN32/INDUSTROYER A new threat for industrial Control Systems, ESET, 2017.06
2. Dragos INC, Crashoverride Analsysis of the Threat to Electric Grid Operations, 2017.06
3. Dragos INC, Trisis malware analysis of safety system targeted malware, 2017.12
4. Kaspersky Lab ICS Cert, Threat Landscape for Industrial Automation Systems in the second half of 2016, Kaspersky Lab (2016)
5. Tahbildar H, Bichitra K (2011) Automated software test data generation: direction of research. Int J Comput Sci Eng Surv 2(1):99–120. https://doi.org/10.5121/ijcses.2011.2108
6. Peng H, Shoshitaishvili Y, Payer M (2018) T-Fuzz: fuzzing by program transformation. In: 2018 IEEE Symposium on Security and Privacy (SP). IEEE, San Francisco, CA, USA, pp 697–710. https://doi.org/10.1109/SP.2018.00056
7. Saheed YK, Babatunde AO (2014) Genetic algorithm technique in program path coverage for improving software testing. Afr J Comp ICT 7(5):151–158
8. American fuzzy lop. http://lcamtuf.coredump.cx/afl/. Accessed 13 Mar 2020
9. libfuzzer. https://llvm.org/docs/LibFuzzer.html. Accessed 13 Mar 2020
10. Tsankov P, Dashti MT, Basin D (2013) Semi-valid input coverage for fuzz testing. In: Proceedings of the 2013 International Symposium on Software Testing and Analysis. ACM. pp 56-66. https://doi.org/10.1145/2483760.2483787
11. Cha SK, Woo M, Brumley D (2015) Program-adaptive mutational fuzzing. In: Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, CA, USA, pp 725–741. https://doi.org/10.1109/SP.2015.50
12. Böhme M, Pham V-T, Roychoudhury A (2017) Coverage-based greybox fuzzing as markov chain. IEEE Trans Softw Eng 45(5):489–506. https://doi.org/10.1109/TSE.2017.2785841
13. Wang J et al (2017) Skyfire: data-driven seed generation for fuzzing. In: 2017 IEEE Symposium on Security and Privacy (SP). IEEE, San Jose, CA, USA, pp 579–594. https://doi.org/10.1109/SP.2017.23
14. Yao F et al (2017) Statsym: vulnerable path discovery through statistics-guided symbolic execution. In: 2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN). IEEE, Denver, CO, USA, pp 109–120. https://doi.org/10.1109/DSN.2017.57
15. Godefroid P, Levin MY, Molnar D (2012) SAGE: whitebox fuzzing for security testing. Queue 10(1):1–8. https://doi.org/10.1145/2090147.2094081
16. Shapiro R, Bratus S, Rogers E, Smith S (2011) Identifying vulnerabilities in SCADA systems via fuzz-testing. In: International Conference on Critical Infrastructure Protection, pp 57–72. https://doi.org/10.1007/978-3-642-24864-1_5
17. Netzob. https://github.com/netzob/netzob. Accessed 13 Mar 2020
18. Peng S, Cui B, Jia R, Liang S, Zhang Y (2013) A novel vulnerability detection method for ZigBee MAC layer. Int J Grid Util Comput 4(2–3):134–143. https://doi.org/10.1504/IJGUC.2013.056249
19. Kim SJ, Shon T (2018) Field classification-based novel fuzzing case generation for ICS protocols. J Supercomput 74:4434–4450. https://doi.org/10.1007/s11227-017-1980-3
20. Klees G et al (2018) Evaluating fuzz testing. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. ACM. Toronto, Canada, pp 2123–2138. https://doi.org/10.1145/3243734.3243804
21. Kargén U, Shahmehri N (2015) Turning programs against each other: high coverage fuzz-testing using binary-code mutation and dynamic slicing. In: Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering (ESEC/FSE 2015). pp 782–792
22. Chen P, Chen H (2018) Angora: efficient fuzzing by principled search. arXiv preprint arXiv:1803.01307

23. Li Y et al (2017) Steelix: program-state based binary fuzzing. In: Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering. ACM, pp 627–637. https://doi.org/10.1145/3106237.3106295
24. Henderson A et al (2017) VDF: targeted evolutionary fuzz testing of virtual devices. In: International Symposium on Research in Attacks, Intrusions, and Defenses. Springer, Cham, pp 3–25. https://doi.org/10.1007/978-3-319-66332-6_1
25. Stephens N et al (2016) Driller: augmenting fuzzing through selective symbolic execution. Proc. Symp. Netw. Distrib. Syst. Secur. pp 1–16
26. Duchene F et al (2012) XSS Vulnerability detection using model inference assisted evolutionary fuzzing. In: 2012 IEEE Fifth International Conference on Software Testing, Verification and Validation (ICST). IEEE, pp 815–817
27. libmodbus. http://libmodbus.org/. Accessed 13 Mar 2020
28. Qassim Q et al (2017) A survey of SCADA testbed implementation approaches. Indian J Sci Technol 10(26):1–8. https://doi.org/10.17485/ijst/2017/v10i26/116775
29. Sematech NIST (2013) Nist/sematech e-handbook of statistical methods. NIST SEMATECH. https://www.itl.nist.gov/div898/handbook/. Accessed 13 Mar 2020
30. Gov. https://gcc.gnu.org/onlinedocs/gcc/Gcov.html#Gcov. Accessed 13 Mar 2020
31. Choi Seung-Seok, Cha Sung-Hyuk, Tappert Charles C (2010) A survey of binary similarity and distance measures. J Syst Cybern Inf 8(1):43–48