Polar: Function Code Aware Fuzz Testing of ICS Protocol

ZHENGXIONG LUO, FEILONG ZUO, YU JIANG, and JIAN GAO, KLISS, BNRist, School of Software, Tsinghua University, China
XUN JIAO, Department of Computer Science and Engineering, Villanova University, USA
JIAGUANG SUN, KLISS, BNRist, School of Software, Tsinghua University, China

Industrial Control System (ICS) protocols are widely used to build communications among system components. Compared with common internet protocols, ICS protocols have more control over remote devices by carrying a specific field called "function code", which assigns what the receive end should do. Therefore, it is of vital importance to ensure their correctness. However, traditional vulnerability detection techniques such as fuzz testing are challenged by the increasing complexity of these diverse ICS protocols.

In this paper, we present a function code aware fuzzing framework — `Polar`, which automatically extracts semantic information from the ICS protocol and utilizes this information to accelerate security vulnerability detection. Based on static analysis and dynamic taint analysis, `Polar` initiates the values of the function code field and identifies some vulnerable operations. Then, novel semantic aware mutation and selection strategies are designed to optimize the fuzzing procedure. For evaluation, we implement `Polar` on top of two popular fuzzers — AFL and AFLFast, and conduct experiments on several widely used ICS protocols such as Modbus, IEC104, and IEC 61850. Results show that, compared with AFL and AFLFast, `Polar` achieves the same code coverage and bug detection numbers at the speed of 1.5X-12X. It also gains increase with 0%–91% more paths within 24 hours. Furthermore, `Polar` has exposed 10 previously unknown vulnerabilities in those protocols, 6 of which have been assigned unique CVE identifiers in the US National Vulnerability Database.

CCS Concepts: • **Security and privacy → Domain-specific security and privacy architectures**; **Vulnerability scanners**; • **Computer systems organization** → *Embedded and cyber-physical systems*;

Additional Key Words and Phrases: Fuzz testing, industrial control system protocol, function code, vulnerability detection

ACM Reference format:
Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function Code Aware Fuzz Testing of ICS Protocol. *ACM Trans. Embed. Comput. Syst.* 18, 5s, Article 93 (October 2019), 22 pages.
https://doi.org/10.1145/3358227

This article appears as part of the ESWEEK-TECS special issue and was presented at the International Conference on Embedded Software (EMSOFT) 2019.

Authors' addresses: Z. Luo, F. Zuo, Y. Jiang (corresponding author), and J. Gao, KLISS, BNRist, School of Software, Tsinghua University, Beijing, China; emails: {luozx19, zuofl19}@mails.tsinghua.edu.cn, jiangyu198964@126.com, gaojian094@gmail.com; X. Jiao, Department of Computer Science and Engineering, Villanova University, USA; email: xun.jiao@villanova.edu; J. Sun, KLISS, BNRist, School of Software, Tsinghua University, Beijing, China; email: sunjiaguang@126.com.

1539-9087/2019/10-ART93 $15.00
https://doi.org/10.1145/3358227

1 INTRODUCTION

Industrial Control System (ICS) is widely used in industrial production. As a general term, it is used to describe the combination of hardware and software with network connectivity, supporting operation or automation on industrial processes. ICS protocol plays an important role in building communications among components of ICS. Different from common internet protocols, ICS protocols are designed to have more control over remote devices for industrial practice. They carry a specific field called "function code" to assign what the receive end should do, e.g., *start*, *stop*, and *report self status*. As such, it is vital to guarantee the correctness of those protocols. However, a large number of severe security vulnerabilities have been revealed in widespread industrial control system protocols. For instance, the well-known Heartbleed [33] vulnerability (CVE-2014-0160) in the OpenSSL library has affected a wide distribution of devices.

There are many fuzzing tools suitable for ICS protocol vulnerability detection such as American Fuzzy Lop (or simply AFL) [43], Sulley [1], Peach [30], and so on. Those fuzzers have been widely used to detect security vulnerability such as application crash, buffer overflow, memory leaks, and double free. They can be roughly classified into two categories based on how test cases are produced: generation-based and mutation-based. Mutation-based fuzzers, such as AFL, generate new inputs by randomly mutating existing inputs. Those fuzzers are popular due to their ease-of-use and fantastic vulnerability-detecting power. In contrast, generation-based fuzzers, including Peach, require format specification and construct inputs by leveraging the knowledge of this format.

In practice, even as these fuzzers have detected lots of vulnerabilities, there remain two challenges heavily limiting their effectiveness: (i) it is not easy to obtain the format specification of ICS protocol and initialize the corresponding values, which requires significant manual efforts to read the documentation and the source code; and (ii) it is difficult for existing fuzzers to reach deep paths in program state space and achieve high code coverage at fast speed because invalid test input mutation and generation result in meaningless repetitions and dramatically affect the speed of fuzzing.

In this paper, we present `Polar`, a function code aware fuzzing framework which requires no extra format specification of protocol packet and is scalable to discover vulnerabilities faster. Instead of optimizing the input generation process to produce more inputs in some existing fuzzing approaches, we extract protocol features and utilize them to produce fewer but higher quality inputs. Through investigation of different ICS protocols, we found that function code field plays an important role in service realization and we can enhance the efficiency of traditional fuzzers by equipping them with function code information. From the perspective of the source code, the value of function code usually determines subsequent execution track, thus, making fuzzers aware of function code information can help them determine where and how to mutate. Meanwhile, we also found that some security-sensitive points in protocol (e.g., dynamic memory allocation `malloc`, we define them as vulnerable operations) can be obtained to assist fuzzers in generating more inputs so as to exercise those vulnerable operations more often.

We utilize static code analysis to filter some candidates of function code parameters and identify some vulnerable operations. Then byte-level taint analysis is applied to further verify those candidates. It assigns each byte in input with a unique label when input data is accessed, tracks the propagation of these labels and verifies how data flows in each variable associated with function code. After obtaining those semantic information, `Polar` makes the best of them to accelerate fuzzing phase. Based on lightweight instrumentation of the source code, `Polar` is able to obtain some feedback during program execution for a given input $\boldsymbol{I}$, such as whether the function code related statements or vulnerable operations are executed. If so, the further fuzzing process of $\boldsymbol{I}$

will take the feedback into consideration, which we define as guided fuzzing. This has three advantages: (i) it utilizes function code information to help exploring new paths by synchronizing useful mutation information between seed inputs that have different values of function code; (ii) it efficiently reduces the size of the mutation space because the legal values of function code are taken from a fixed small set where enumerating exhaustively would be unnecessary and inefficient; and (iii) the malformed inputs generated by guided fuzzing are more likely to exercise vulnerable operations in the program and expose security vulnerabilities.

For evaluation, we augmented AFL and AFLFast with `Polar` and evaluated their performance on several widely used open-source implementations of ICS protocols – Modbus [22, 37], IEC104 [9, 36] and IEC61850 [13, 34]. Experiment results show that `Polar` can help to achieve high code coverage at a faster speed (an average of 3.6X and 1.5X for AFL and AFLFast respectively) and can gain sustained increases in paths covered (an average of 19.9% and 18.8% increase for AFL and AFLFast respectively) after 24 hours. Meanwhile, `Polar` has already exposed 10 previously unknown vulnerabilities, 6 of which have been assigned with unique CVE identifiers in the US National Vulnerability Database.

To the best of our knowledge, most ICS protocols are function code-oriented. Apart from Modbus, IEC104, and IEC 61850, many other ICS protocols can also be applied with `Polar`, such as IEEE C37.118 [2], Profinet [41], DNP3 [38], ICCP [39], and IEC101 [40]. In conclusion, our paper makes the following contributions:

- We propose a novel lightweight approach that combines static and dynamic code analysis to infer where an ICS protocol program processes function code field of packet without manual effort.
- We propose a novel semantic information aware fuzzing strategy to accelerate fuzzing speed and improve path coverage so as to expose more vulnerabilities.
- We implement `Polar`, evaluate it on several ICS protocols, and have detected many previously unknown vulnerabilities. `Polar`[1] is open source for public use and can be augmented to existing fuzzers for further improvement.

The rest of this paper is organized as follows: necessary background is introduced in Section 2 and a motivating example is given in Section 3. Then we detail the method and implementation of `Polar` in Section 4 and the performance results in Section 5. Last, we introduce some related work in Section 6 and summarize the paper in Section 7.

2 BACKGROUND

2.1 Industrial Control System and Protocol

Industrial control system (ICS) is a general term used to describe the combination of hardware and software with network connectivity so as to support critical infrastructure, such as energy, transportation, and communications. However, the increasing number of ICS components available over the Internet makes ICS easy prey for attackers. In the design of this system structure, ICS protocol plays an important role in building communications among system components. Those ICS protocols such as Modbus, IEC104, and IEC 61850 have been used in a wide range of industrial domains. Their correctness directly affects the safe operation of ICS, thus, vulnerability detecting techniques for ICS protocols are clearly needed.

Unlike the common internet protocols, ICS protocols are designed to acquire measurements and status and to control other devices. In order to do this, the ICS protocol packet usually carries a special field, called the function code field, to specify what is received and what should be responded.

[1] https://github.com/fouzhe/Polar-Fuzz

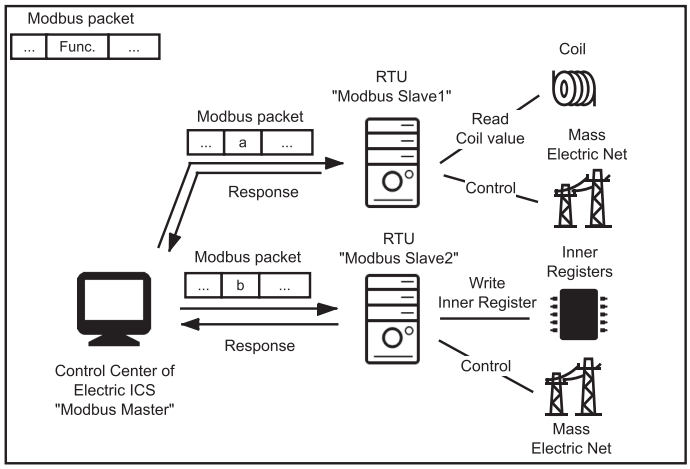


Fig. 1. Electrical ICS running Modbus protocol.

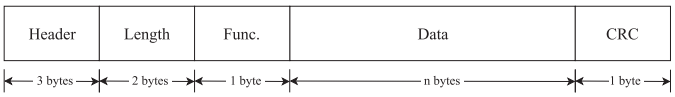


Fig. 2. An example of ICS protocol packet.

Figure 1 shows an example of electrical ICS running protocol Modbus. In this example, a device in the Control Center of the ICS plays the role as "Modbus Master" and two remote terminal units (RTUs) serve as the "Modbus Slave"s. Each RTU has direct control over a mass electric net, which has a great influence on people's daily life. The whole ICS builds communication among devices with protocol Modbus. A function code field is applied in Modbus to assign what the receive end should do. As shown in this example, the master sends a packet with function code value *a* to slave 1, which means the slave should read its coil value. So after receiving this packet, slave 1 reads its own coil, gets the value and responses to the master. Similarly, slave 2 gets a packet with function code value *b*, which orders it to write some values in the packet to its inner registers to change the operation of the electric net. Relaying on protocol Modbus, this ICS performs smoothly and normally.

As an example, Figure 2 shows the simplified format of an ICS protocol packet: The `Header` field declares information about protocol and packet, which usually contains protocol signature, transaction identifier, unit identifier, etc; `Length` field means the number of remaining bytes in this frame; `Func.` field represents the function code field; `Data` field indicates the data content associated with function code; `CRC` field is short for cyclic redundancy check, which is an error check mechanism to ensure the reliability of data. In this format, the `Func.` field is just the special field designed for transmitting orders.

Figure 3 shows a simplified process of packet analysis in ICS protocol. We found that different values of the function code in the packet direct the program to exercise different code paths. In detail, assuming that an ICS protocol packet is received, the program first checks its integrity through such filed as CRC (stage I). If valid, the program further processes the packet based on the `Func.` field. The value indicated by the function code specifies the following trace. (stage II). Then, the program will execute the corresponding trace with predefined libraries (stage III).

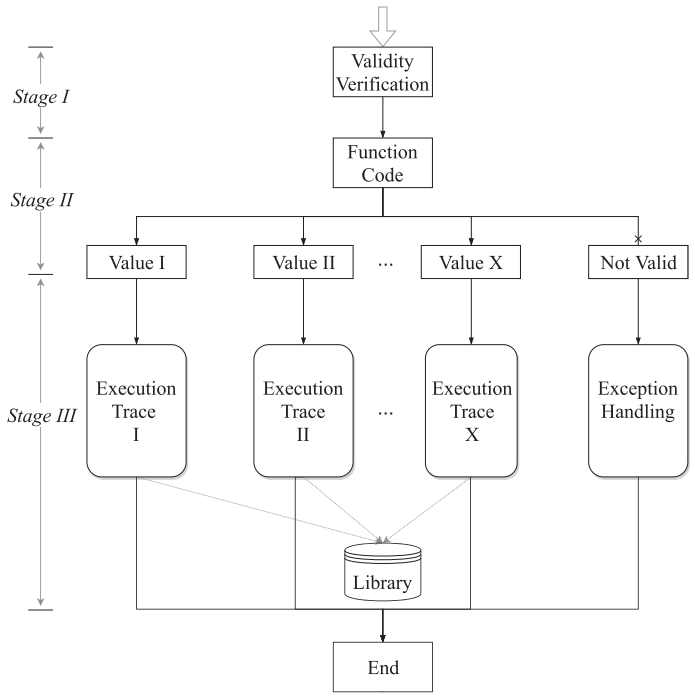


Fig. 3. Overview of packet analysis in ICS protocol.

2.2 Fuzz Testing

As an automated software testing technique, fuzzing has emerged as one of the most effective techniques for detecting bugs and security vulnerabilities in real-world software [7, 10, 12, 16, 20, 26]. It is first developed by Miller et al. [21] in 1990 and has, since then, been widely adopted in practice. A number of serious security vulnerabilities in some important software programs have been exposed by fuzzing [28].

Based on the utilization degree of internal program structure, fuzzers can be classified into whitebox, blackbox and greybox. A whitebox fuzzer [4, 14, 15] utilizes source code analysis to better understand the structure of program, while a blackbox fuzzer [1, 30] only requires access to the program. A greybox fuzzer is an intermediate solution, and it employs some approaches to obtain feedback from under-test-program and leverages those information to guide their fuzzing strategy. Coverage-based greybox fuzzing (CGF) is a typical greybox fuzzing technology that employs lightweight instrumentation to obtain coverage information. Taking AFL for example, it injects instrumentation at branch points of the program under test and gets the coverage information as follows:

```
cur_location = <COMPILE_TIME_RANDOM>;
shared_mem[cur_location ^ prev_location]++;
prev_location = cur_location >> 1;
```

The value of `cur_location` is generated randomly during compile time and is used to specify the basic block. The `shared_mem[]` array is a 64 kB shared memory region used to track coverage. Each byte set at $(A >> 1) \oplus B$ in `shared_mem[]` records hits of transition from basic block A to B. CGF leverages coverage information to guide seed mutation. A general sketch is shown in Algorithm 1.

ALGORITHM 1: Coverage-based Greybox Fuzzing (CGF)

```
   Input: S: initial seeds
   Input: P: program under test
   Output: Crashes: test cases that make program crash
   Output: Hangs: test cases that make program hang
1  Queue ← S
2  Crashes ← ∅
3  Hangs ← ∅
4  while true do
5      seed ← PickSeed(Queue)
6      score ← CalculateScore(P, seed)
7      for 1 ≤ i ≤ score do
8          seed′ ← Mutate(seed)
9          Results ← RunTarget(P, seed′)
10         if Crash(Results) then
11             Crashes ← Crashes ⋃ {seed′}
12         else if Hang(Results) then
13             Hangs ← Hangs ⋃ {seed′}
14         else if isInteresting(Results) then
15             AddToQueue(Queue, seed′)
```

At the beginning, the fuzzer is provided with a set of seed inputs S and they are added to the seed pool *Queue* (lines 1–3). The seeds in *Queue* are chosen in a continuous loop unless timeout or aborted (line 4). In each loop iteration, the fuzzer first chooses a *seed* from *Queue* (line 5) and calculates the performance score of *seed* (line 6) as implemented in *CalculateScore*(). This is also where the power schedule is implemented because *score* is then used to determine the amount of time to spend mutating *seed* (line 7). In the implementation of AFL, *CalculateScore*() makes use of the performance reports of *seed* such as execution time, block transition coverage, and program depth achieved. Then, the fuzzer generates new inputs by randomly mutating *seed* as implemented in *Mutate*() (line 8). The new input *seed′* is then used to run the under-test-program (line 9). If it crashes or hangs the program, then it will be added to the corresponding set (lines 10–13). When *seed′* achieves new program coverage, it will be marked as *interesting seed* and added to *Queue* for further fuzzing (lines 14–15).

3 MOTIVATIONAL EXAMPLE

As a motivating example for the proposed fuzzing framework, Listing 1 shows a simplified sample code snippet that parses packets from the example format shown in Figure 2. In order to point out the problem, we omit the verification code snippet of `Header` and `CRC`. First, the code reads the `Length` and `Func.` fields (lines 9–10), and then takes further measures according to the value of function code (line 12-30). In the sample code of Listing 1, the various operations supported by the function code can be classified into two classes: *Data Access* and *Diagnostics*. The former means some operations involved with the data in the internal register or other storage devices in the programmable logic controller (PLC) devices, including `WRITE_REGISTERS` (line 16), `READ_REGISTERS` (line 23) and `WRITE_AND_READ_REGISTERS` (line 25); the latter usually refers to a series of operations about getting the status or event log of PLC, such as `REPORT_SLAVE_ID` (line 13). In general, *Data Access* is more vulnerable than *Diagnostics* because *Data Access* involves

```
1  #define REPORT_SLAVE_ID                  0x01
2  #define READ_REGISTERS                   0x0F
3  #define WRITE_REGISTERS                  0x16
4  #define WRITE_AND_READ_REGISTERS         0x17
5
6  void decode_packet(char *buf){
7    ...
8    char rsp[MAX_MESSAGE_LENGTH];  /* response message */
9    int length = get_length(buf);
10   char function_code = get_function_code(buf);
11   // further action according to function code
12   switch (function_code) {
13     case REPORT_SLAVE_ID:
14       report(rsp);
15       break;
16     case WRITE_REGISTERS:
17       char* register_t=get_register_address(buf,length);
18       int write_len = get_write_len(buf,length);
19       // Bug: function with heap buffer overflow
20       int status=write(register_t,buf,length,write_len);
21       response_after_write(rsp, status);
22       break;
23     case READ_REGISTERS:
24       ...
25     case WRITE_AND_READ_REGISTERS:
26       ...
27     default:
28       error();
29       break;
30   }
31   ...
32 }
```

Listing 1. An example code snippet of decoding packet

access to the storage device while *Diagnostics* is just a response to query without further action. The sample code has a "deep" heap buffer overflow bug in the function `write` in line 20 because the `write` function uses the `write_len` value from the packet without sanitization, and the corresponding function code is `WRITE_REGISTERS`.

Traditional mutation-based fuzzers, like AFL, do not adequately expose such heap buffer overflow vulnerability in Listing 1. Since mutation-based fuzzers are unaware of the packet format, random mutation operations on the `Func.` field will cause most of the generated packets to be rejected in line 27. Furthermore, with equal treatment for each seed, the lack of awareness of vulnerable operations and pertinence of more vulnerable traces would result in a low probability of triggering the error in line 20. Traditional fuzzing methods would require significant effort applying blind and meaningless repeated modifications to explore the whole space.

In contrast, `Polar` leverages static and dynamic program analysis to detect function code (line 12) information and vulnerable operation (line 20) information, and then makes use of those information during fuzzing. In detail, function code information (`function_code`) can be used to accelerate the fuzzing process by exploiting the information to synchronize useful mutation information between seeds and avoid invalid mutation on this field. Furthermore, vulnerable operation information (`status = write()`) can be applied to help fuzzer select more seeds to exercise those operations more frequently, which makes triggering vulnerabilities faster and more likely.

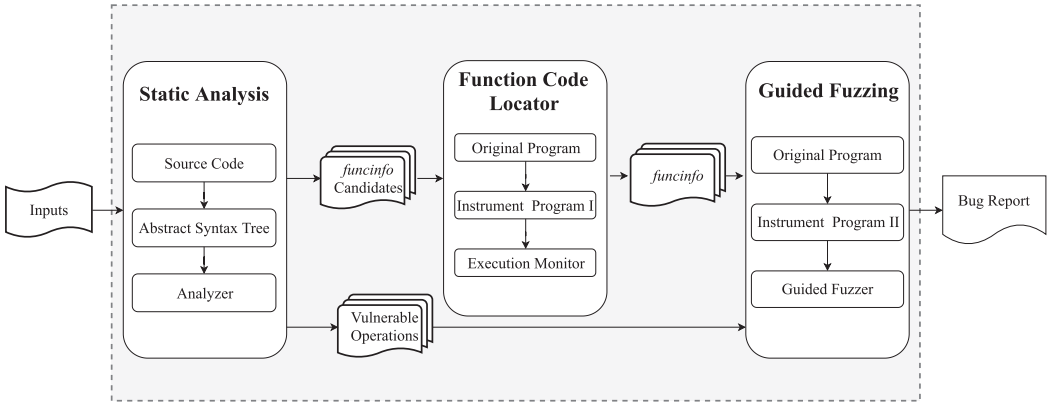


Fig. 4. Polar System Overview. It mainly consists of three components: Static Analysis for detecting function code candidates and vulnerable operations; Function Code Locator for function code verification; Guided Fuzzing for novel fuzzing strategy.

4 SYSTEM DESIGN

In this section, we first introduce the overview of Polar using the motivation example above, and then present details of the design.

4.1 Polar Overview

To achieve the proposed fuzzing framework, one of the most critical steps in the process is to acquire pertinent information such as which byte offsets in the packet belong to the function code field. Therefore, Polar requires byte-level taint tracing. But taint tracing is relatively expensive for: (i) tracking each variable individually; and (ii) taint tracing in the fuzzing stage. Our key insight is that taint tracing is unnecessary for the execution of the program during fuzzing. We can conduct pre-processing before the fuzzing phase to obtain function code information. Furthermore, static analysis can be applied for reducing the number of variables to be tainted. At its highest level, Polar contains three components: *static analysis, function code locator*, and *guided fuzzing*, as shown in Figure 4. We use the program in Listing 1 to illustrate Polar 's basic workflow.

Static Analysis Module. Given an ICS protocol program ***P***, Polar first uses a lightweight static analysis to obtain: (i)*funcinfo* candidates; by scanning the source code of ICS protocol, static analyzer extracts specified code structures that function code processing statement may conform to and records their information in the file *funcinfo* candidates; (ii) *Vulnerable Operations*; simultaneously, static analyzer will also collect information about security-sensitive operations such as dynamic memory allocation functions (e.g., malloc, realloc) and a set of functions implementing operations on strings (e.g. memcpy, strcpy, strcat).

In the example of Listing 1, the *funcinfo* candidates will include:

```
Source File: decoder.c
   Position: line 12
   Variable: function_code
     Values: [0x01, 0x0F, 0x16, 0x17]
```

which means the variable function_code in line 12 of decoder.c may record the value of the function code and its possible values are 0x01, 0x0F, 0x16, and 0x17. Similarly, if there is a memcpy

function call in function `write` in line 20 and a `malloc` function call in line 31, the *Vulnerable Operations* report will include entries as follows:

```
decoder.c: 20: memcpy
decoder.c: 31: malloc
```

which means there are vulnerable operations in line 20 and 31 of `decoder.c`, and their related functions are `memcpy` and `malloc` respectively.

Function Code Locator Module. The first step may identify many *funcinfo* candidates with potential false positive, which need to be further verified. In this phase, `Polar` applies taint analysis to monitor the flow of data in protocol program ***P*** for a given input ***I***. More specifically, the execution monitor records which input bytes of ***I*** determine the value of candidate *funcinfo* variable. In this example, the byte that influences the value of candidate *funcinfo* variable `function_code` is the sixth byte (see Figure 2). Through that taint information collected during the preliminary run-time execution, the *funcinfo* can be further confirmed. The final output version of *funcinfo* is as below:

```
    Source File: decoder.c
       Position: line 12
  Starting Byte: 6
    Ending Byte: 6
       Variable: function_code
         Values: [0x01, 0x0F, 0x16, 0x17]
```

Guided Fuzzing Module. In the third phase, the traditional fuzzer is improved based on the identified *funcinfo* and *Vulnerable Operations*. `Polar`'s main optimization in the fuzzing strategy is designed for ICS protocols based on our investigation of their features. `Polar` incorporates lightweight synchronization mechanism to share useful path information, avoids invalid repetition on different values of the function code field and tries to exercise vulnerable traces more often. For example, the test input seeds that cover line 12 and 31 would be selected and mutated with more possibilities. The mutation information of interesting seeds with the function code of value 0x0F would be partially synchronized to seeds with the function code of value 0x17 by `Polar`'s fuzzing procedure.

4.2 Static Analysis

The static analysis module is used to detect suspicious function code statements and vulnerable operations in the source code. Since the function code field in the ICS protocol is usually the key point designed for a variety of control demands, the function code processing statement is usually a multi-branch statement that determines the following execution directions. As shown in line 12 in Listing 1, different functions are represented as different values of the `Func.` field in Figure 2. For instance, the set of legal values that the variable `function_code` in Listing 1 can have is {0x01, 0x0F, 0x16, 0x17}, designed for different demands respectively.

Algorithm 2 shows the detail of this module. We first translate the source code into abstract syntax tree (AST) for better understanding and analysis (line 2). Compared to the source program, AST provides a more structured and precise format for analyzing the code construction and logic component during the static analysis period. In the tree format, each node of the tree represents one syntax structure in the source program and the children of the node correspond to the units forming this syntax structure.

Then, we use the Depth First Search (DFS) algorithm to traverse the abstract syntax tree and locate the potential source code related to the function code field (lines 3–13). As mentioned before, a

ALGORITHM 2: Extract *funcinfo* candidates

```
   Input: P: program under test
   Output: f: funcinfo candidates
1  Algorithm
2      AST ← CompileAst(P)
3      root ← GetRoot(AST)
4      DFS(root)
5  Procedure DFS(tree_node)
6      if tree_node equals null then
7          return
8      else
9          if IsMultipleBranch(tree_node) then
10             f ← f ⋃ ExtractFuncinfo(tree_node)
11         Children ← GetChildren(tree_node)
12         for child ∈ Children do
13             DFS(child)
```

function code judging statement usually tends to be multiple branch statement. From the perspective of source code, it may be switch-case statement or if-then statements. Correspondingly, the function *IsMultipleBranch*() can be designed to filter multi-branch subtree based on their feature. We can dive into the details for those two structures: (i) Switch-case statement. In the format of AST, the whole information of a switch-case structure is recorded in the subtree with the root of the "SwitchStmt" tree node. As a result, to locate a switch-case structured multiple branch statement, we argue that when a "SwtichStm" tree node is found during DFS, the subtree with root of this node can be marked as what we want. (ii) In addition, when facing protocols with structures like if-then statements, we can use similar ways to extract *funcinfo* candidates. Based on the feature of function code in ICS protocol, we list two screening conditions for *IsMultipleBranch*(): (1) the depth of corresponding if-then AST should be greater than a threshold; and (2) the condition params of all "if" must be the same one, which should be recorded as *funcinfo* candidate variable and the corresponding values of conditions will be marked as legal values of this variable. Once obtaining a multi-branch subtree, we can extract its information such as variable name, position of this multi-branch, legal value set, and so on (line 10). Meanwhile, when traversing AST, we can collect the position information of vulnerable operations in the source code.

4.3 Function Code Locator

After obtaining the information of potential function code related statements, `Polar` works as follows to reduce false positives and verify their authenticity.

Dynamic Taint Tracing. In order to monitor the data flow of under-test-program for given input, `Polar` uses dynamic taint analysis (DTA). DTA can keep track of tainted input and determine which memory locations and registers are dependent on it. Furthermore, based on different granularity, DTA can be extended to trace the derivation of the taint values to individual offsets in the input. `Polar` implements a byte-level taint tracing and tracks the label propagation during program execution based on LLVM DataFlowSanitizer (DFSan) [8].

To implement DTA for ICS protocol analysis, there are two problems that need to be solved: (i) How to identify the taint source to assign taint labels? (ii) How to extract the label information of the target variables? For the first problem, the taint source is the sampled ICS protocol packet,

and the data read from the packet should be marked for further taint label assignment. To do this, `Polar` makes use of the `ABI List` in DFSan to intercept relevant function calls and replace them with the corresponding `custom wrapper`. For example, for a packet transmitted via socket, `Polar` intercepts function calls such as `socket, recv, read`, and `close`. When a socket is created, `Polar` records the file descriptor `fd`. Each time the program reads from this socket, `Polar` records the offset and assigns different labels for each byte in the buffer. The value of offset can be updated in accordance with the return value of `recv`. Furthermore, `Polar` stops tracking `fd` when the socket is closed. For the second problem, `Polar` instruments the original source code with calls to the taint source library, and the position for instrumentation is the code point where the *funcinfo* candidate variable is used, such as line 12 in the Listing 1. The taint library is used to extract the label information for those target variables and pass it to the execution monitor.

Function Code Verification. In this step, the execution monitor is used to verify *funcinfo* candidates and delete false positives. Algorithm 3 provides the overview of the process.

ALGORITHM 3: Verify Function Code Information

Input: $\mathcal{M}$: set of *funcinfo* candidates
Input: *seeds*: initial inputs for the program
Input: P: program under test
Output: $\mathcal{M}'$: subset of *funcinfo* candidates after verification(true *funcinfo*)
Output: $\mathcal{O}$: set of offset information for each *funcinfo* in $\mathcal{M}$

```
1  for e ∈ M do
2      O_e ← ∅
3  M' ← M
4  for seed ∈ seeds do
5      taint_information ← RunTarget(P, seed)
6      for e ∈ M do
7          I ← GetOffsets(taint_information, e)
8          if O_e is empty then
9              O_e ← I
10         else if not Equal(O_e, O_e ∩ I) then
11             M' ← M' − e
```

For Algorithm 3 in detail, $\mathcal{M}$ represents the set of *funcinfo* candidates. Then, for each iteration, the execution monitor fetches an input *seed* from the initial test inputs, runs the target program with *seed*, and collects the *taint_information* simultaneously during the execution (lines 4–5). The initial test inputs are packets randomly sampled on network in real industrial production environment. Then, for each candidate e in $\mathcal{M}$, the execution monitor extracts the offsets of bytes in the *seed* that taint the target variable of e from the *taint_information* through the `Polar`'s taint source library (lines 6–7). Let the offsets of e be $\mathcal{O}_e$. The execution monitor records $\mathcal{O}_e$ after each run and monitors whether it is always the same for each *seed*. More specifically, if $\mathcal{O}_e$ is always the same, the execution monitor retains e. Otherwise, it discards e from $\mathcal{M}'$ (lines 8–11). This is based on the observation that, for a given function code *func.* (there maybe multiple function codes in some ICS protocols), the byte offsets of *func.* in the protocol packets are fixed.

After running the program with all the initial *seeds*, we assume that the remaining *funcinfo* candidates in $\mathcal{M}'$ are true function code information, and record their information along with byte offsets in the file *funcinfo*. A piece of *funcinfo* consists of the source file, variable position,

starting offset of byte, ending offset of byte and legal values as shown in the Section 4.1. Those information is used for guided fuzzing.

4.4 Guided Fuzzing

The guided fuzzing module of Polar incorporates that information into the traditional coverage-based greybox fuzzing (CGF) shown in Algorithm 1. In simple terms, the original CGF generates malformed inputs by mutating existing inputs, feeds them to the under-test-program and records the inputs when the program crashes or hangs. In Polar, we devise a novel fuzzing strategy for ICS protocol fuzzing to increase the efficiency based on the *funcinfo* and *Vulnerable Operations* obtained in the above steps. Our modifications are mainly reflected in three aspects: (i) First, we modify the way of calculating a seed's performance score, so that vulnerable operations are exercised more frequently; (ii) Second, we modify the way mutations are imposed on seeds, avoiding blind modification on some key areas; (iii) Third, based on the feature of ICS protocol, we design a lightweight synchronization mechanism between seeds with different values of the function code to help explore new paths faster. The first two aspects are addressed in the function *CalculateScore*() and *Mutate*() of Algorithm 1. The third aspect is addressed in Algorithm 4.

Guided Seed Prioritization and Mutation. Given the original program, we first instrument the program based on the *funcinfo* and *Vulnerable Operations*. Unlike the instrumentation of CFG presented in Section 2.2, we use a more precise instrumentation to record whether the function code statements or vulnerable operations are executed. A sketch of the code that is inserted at each target point in the program is shown in Listing 2 (lines 2,7):

```
1 // function code statement
2 do{shared_mem[Func_ID]++; } while (0)
3 switch(function_code) {...}
4 ...
5 // vulnerable operation
6 malloc(...);
7 do{shared_mem[Vul_Op_Index]++; } while (0)
```

Listing 2. Polar 's Instrumentation

The variable Func_ID identifies the current function code statement. It is assigned by the execution monitor in the function code locator module, and the variable Vul_Op_Index is assigned a constant value. Hence, the shared memory of existing fuzzers such as AFL could be expanded to collect those information easily. The execution trace of the under-test-program on a given input is collected as a set of pairs of the form (*Func_ID, hit_counts*) and a pair of the form (*Vul_Op_Index, hit_counts*), where *hit_counts* means the hit times of the corresponding statement for a single execution. Due to the small number of *funcinfo* and *Vulnerable Operations*, the overhead of our instrumentation can be roughly ignored.

Secondly, Polar implements a power schedule that not only considers those indexes mentioned in Section 2.2, but also brings vulnerable operations into the analytical scope. Let $\mathcal{E}(I)$ denotes energy of seed $\boldsymbol{I}$, and the original energy of $\boldsymbol{I}$ calculated by base fuzzer is $\mathcal{E}_{ini}(I)$. The times of vulnerable operations $Count_I$ is recorded during execution. Polar computes $\mathcal{E}(I)$ as Formula (1):

$$\mathcal{E}(I) = min\left(\frac{\mathcal{E}_{ini}(I)}{\beta}h(Count_I), M\right) \tag{1}$$

$h(x)$ is an increasing function and $\mathcal{E}(I)$ increases as $Count_I$ increases. β balances the relation between the base fuzzer energy assignment value and Polar's energy assignment value. The constant

M provides an upper bound on the number of mutations per fuzzing iteration. Hence, the higher the value of $Count_I$ is, the more energy will be assigned to $\boldsymbol{I}$ for further mutation. Thus, the chance to exercise those vulnerable operations is much higher.

Thirdly, `Polar` optimizes the mutation strategy used by traditional CGF that takes each bit/byte into consideration. For a given seed $\boldsymbol{I}$, let $\mathcal{P}_I$ represents the set of *Func_ID* hit by $\boldsymbol{I}$ during execution, that is to say, for each *Func_ID* $\int \in \mathcal{P}_I$, the value of *hit_counts* of $\int$ is non-zero. If $\boldsymbol{I}$ achieves new program coverage and is marked as *interesting seed* (Algorithm 1, line 14), $\mathcal{P}_I$ will be taken into consideration during $\boldsymbol{I}$'s mutation. In particular, if a byte of $\boldsymbol{I}$ belongs to $[start\_byte_{\int}, end\_byte_{\int}]$ for some $\int \in \mathcal{P}_I$ in the *funcinfo*, it is then protected against being mutated. This approach can effectively reduce the cardinality of mutation space. We have found empirically that more than 90% of *seeds* in the *Queue* (Algorithm 1, line 5) hit one or more function code branches. Therefore, it makes sense to protect the `Func.` field because blind mutation will cause too many invalid seeds, and feeding them to the target program is time-consuming (especially for large programs that run slowly) and meaningless. In addition, due to random mutation of CGF, many seeds produced are likely to be rejected during `Validity Verification` (Figure 3, stage I). Assuming that seed $\boldsymbol{I}_0$ can pass the verification, blindly mutating the `Func.` field of $\boldsymbol{I}_0$ will increase the probability of being rejected during `Function Code Verification` (Figure 3, stage II), making it more difficult for fuzzers to reach deep places in a program.

Guided Seed Synchronization. Last, we add a novel synchronization strategy called `replace` to traditional CGF, aimed to synchronize useful mutation information between seeds with different values of the function code. As mentioned before, different function code values will cause different execution traces. However, for different values, we also observe that there are many similar operations between some traces and they tend to include the same code snippet or call the same functions in some library as shown in Figure 3. Taking the function code in Modbus [37] for example, the function code value specified to `write single coil` has almost the same operations as the function code value specified to `write single register`. They all need to calculate the mapping address, calculate the data to write and construct a response message. The only difference between them is the place to write. In addition, the function code specified to `write single coil` performs a subset of the operations carried out with the function code

ALGORITHM 4: Synchronization Mechanism between different values of the Function Code

Input: $\boldsymbol{I}$: seed to mutate
Input: $\mathcal{P}_I$: set of *Func_ID* hit by $\boldsymbol{I}$ during execution
Input: *funcinfo*: function code information obtained
Output: $\mathcal{F}$: test cases generated by mutating $\boldsymbol{I}$ in `replace` phase

1 $\mathcal{F} \leftarrow \varnothing$
2 **if** $\mathcal{P}_I$ *is not empty* **then**
3 **for** $\int \in \mathcal{P}_I$ **do**
4 $\mathcal{X}_{\int} \leftarrow$ GetSpecific(*funcinfo*, $\int$)
5 $C \leftarrow$ GetCandidates($\mathcal{X}_{\int}$)
6 $start\_byte, end\_byte \leftarrow$ GetRange($\mathcal{X}_{\int}$)
7 **for** $c \in C$ **do**
8 $\boldsymbol{I}' \leftarrow \boldsymbol{I}$
9 **for** $start\_byte \le i \le end\_byte$ **do**
10 $\boldsymbol{I}' \leftarrow$ Replace($\boldsymbol{I}', c, i$)
11 $\mathcal{F} \leftarrow \mathcal{F} \bigcup \{\boldsymbol{I}'\}$

`write multiple coil`. Hence, an input with one value of the function code that achieves new program coverage can be used to guide the mutation of inputs with other values of the function code. Based on those features, we implement the following synchronization strategy.

Algorithm 4 describes this strategy. Given a seed $\boldsymbol{I}$ to mutate, when set $\mathcal{P}_I$ of $\boldsymbol{I}$ is not empty, the mutation strategy `replace` will be applied to generate new inputs (lines 3–11). For each *Func_ID* $\int \in \mathcal{P}_I$, the detailed information will be extracted from *funcinfo* (lines 4–6) and the bytes of $\boldsymbol{I}$ in range $[start\_byte_{\int}, end\_byte_{\int}]$ will be replaced with legal candidate values (lines 7–11).

This synchronization strategy is lightweight and efficient for fuzz testing of ICS protocol. Due to the similarity between some traces for different values of the function code, the test cases generated in this phase satisfy many primitive constraints and have a high likelihood of exploring new paths, making trigger potential vulnerabilities more likely. Through the experiment, we found that this synchronization mechanism can sustainably and efficiently provide *interesting seeds* during the fuzzing of ICS protocol.

4.5 System Implementation

`Polar` consists of three modules: the static analyzer, the function code locator and the guided fuzzer. The static analyzer exploits Clang compiler to obtain the AST information from the ICS protocol source program during the period of compiling. In the function code locator, to support fine-grained byte-level taint tracing described in Section 4.3, we implement taint tracing for `Polar` based on LLVM DataFlowSanitizer (DFSan) [8]. To identify taint sources, we make use of the `ABI List` in DFsan to intercept relevant function calls. Furthermore, we implement a taint library to extract label information for the execution monitor.

The guided fuzzer module is based on AFL 2.52b/AFLFast [3] (called `Polar`-AFL and `Polar`-AFLFast respectively). We extend the shared memory segment to store hits of function code branches and times of vulnerable operations. To support source code instrumentation, we extend afl-clang-fast [29] and use it as the compiler of our tool chain. Moreover, we add a new mutation strategy named `replace` and modify the existing mutation strategy in AFL/AFLFast, making use of *funcinfo*. Meanwhile, the power schedule takes *Vulnerable Operations* into consideration. Implementation details are available through the Github page reported at Footnote 1.

5 EVALUATION

In order to measure the effectiveness of `Polar`, this section presents an evaluation of the program. First, we evaluated the accuracy of `Polar`'s function code identification (for the module *Static Analysis* and *Function Code Locator*) in Section 5.2. Then we evaluated the efficiency of our novel fuzzing strategy (for the module *Guided Fuzzing*) in Section 5.3. We implemented our framework on AFL and AFLFast (called `Polar`-AFL and `Polar`-AFLFast separately) and compared their results to demonstrate the acceleration in fuzzing and improvement in coverage. Last, in Section 5.4, we list the previously unknown vulnerabilities detected by `Polar`.

5.1 Experiment Setup

We evaluated the performance of `Polar` on several open-source implementations of some widely used ICS protocols. We chose libmodbus [22, 37], IEC104 [9, 36] and libiec61850 [13, 34]. Those ICS protocols are both typical and widely used in industrial practice. Table 1 shows the description of those protocols.

We tested our framework on the top of two popular fuzzing tools, AFL and AFLFast. AFL is a state-of-art coverage-base greybox fuzzer that has exposed vulnerabilities in a widespread of programs including OpenSSL, PHP, tcpdump [35]. AFLFast is an enhancement of AFL. It extends AFL by collecting path frequency and utilizing it to prioritize seeds exercising low-frequency paths.

Table 1. Description of Selected ICS Protocols

Protocol	Description
Modbus	Modbus protocol has turned into the de facto standard for building communications between industrial devices since it was presented in 1979. It is a serial network protocol based on TCP/IP.
IEC104	IEC104 is an international standard widely used in electric power, urban rail transit and other industries. It ensures the power supply systems operating safely and reliably.
IEC61850	IEC61850 protocol is one of the most important protocols in the field of electric power system automation. It realizes the engineering operation standardization of intelligent substation.

Table 2. Function Code Identification Results

Project	$\|\mathcal{M}\|$	$\|funcinfo\|$	Set of Legal Values (hexadecimal) for Each *funcinfo* Piece	True?
libmodbus	11	1	[01,02,03,04,05,06,07,0F,10,11,16,17]	✓
IEC104	12	2	[07,13,43,0B,23,83,64]	✓
			[83,64,67,30,32,80,81]	✓
libiec61850	174	1	[02,80,A1,82,A4,A5,A6,AB,AC,AD]	✓

It, thus, achieves high code coverage and exposes vulnerabilities faster than AFL [3]. To analyze the code coverage achieved by different fuzzing tools, we chose the commonly used path coverage as the main metric. A new path in AFL means a new program execution state. Therefore, the more paths a fuzzing tool explores, the higher probability it will detect a vulnerability.

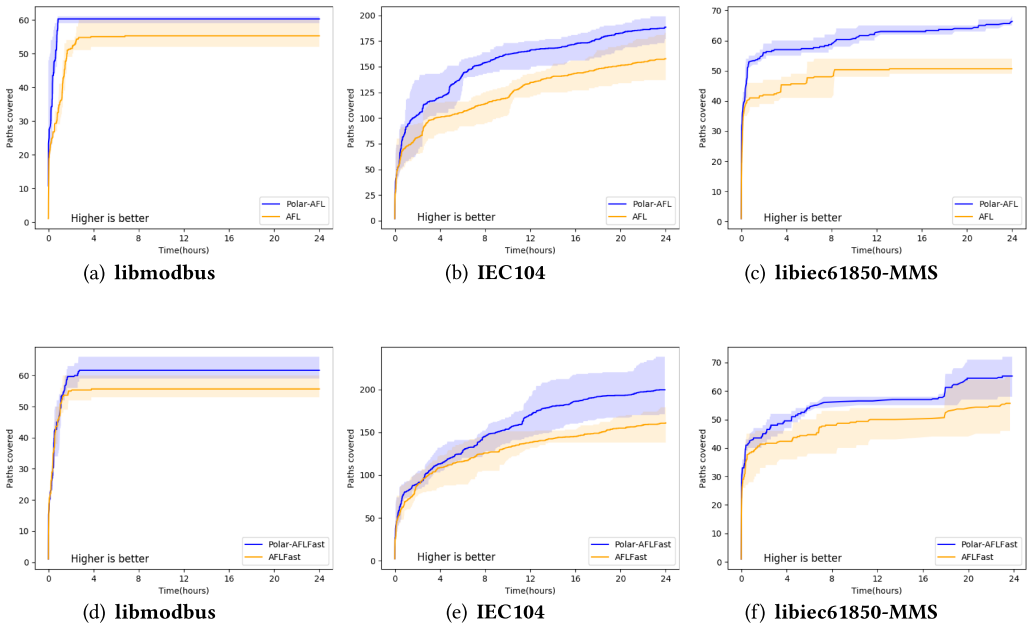
Our experiments were conducted on a 64-bit machine with 80 cores (Intel(R) Xeon(R) Gold 6148 CPU @ 2.40GHz), 128GB of main memory, and Ubuntu 16.04.6 LTS as the host OS. We ran each fuzzing tool on each program for 24 hours (on a single CPU core).

5.2 Accuracy on Function Code Identification

Since accurately locating the function code processing statements is critical in our proposed framework, we ran `Polar` on the above three ICS protocols to evaluate its accuracy.

The results of `Polar` 's function code identification are summarized in Table 2. There are two steps for `Polar` to identify function code: a screening process and a verification process. In the first step, `Polar` uses static analysis to filter some *funcinfo* candidates. It scans the whole program and extracts multi-branch information. As Table 2 shows, the column $|\mathcal{M}|$ represents the number of *funcinfo* candidates detected by module *Static Analysis*. For the project libiec61850, $|\mathcal{M}|$ is large because it is complex and applies many switch statements to process other data. In the second step, to refine the set $\mathcal{M}$, `Polar` instruments the program with calls to `Polar` taint library at the detected candidate points, runs the program with malformed packets and monitors how data flows in each point. The third column $|funcinfo|$ indicates the number of pieces in the final *funcinfo*. Additionally, the fourth column lists the legal values detected by `Polar` corresponding to each *funcinfo* piece. After manual inspection with the ground-truth, the fifth column shows whether `Polar` successfully detected true function code processing statements, where "✓" means success and "✗" means failure.

As an example shown in Table 2, in libmodbus, `Polar` detected 11 suspected function code statements after static analysis. After the `Polar` taint analysis stage, only one was left as true *funcinfo*



(a) **libmodbus** (b) **IEC104** (c) **libiec61850-MMS**

(d) **libmodbus** (e) **IEC104** (f) **libiec61850-MMS**

Fig. 5. Number of paths covered by different fuzzing techniques averaged over 25 runs with different seeds.

and the legal value set is [01,02,03,04,05,06,07,0F,10,11,16,17] for the hexadecimal. This *funcinfo* was further verified to be true after we referred to the source code [22] and the documentation [37].

In conclusion, the results show that `Polar` can successfully detect the function code statement pieces and extract the precise legal values corresponding to each `Func.` field. These pieces of information will be made use of in the following guided fuzzing stage.

5.3 Acceleration on Fuzzing

Results Overview. After obtaining the *funcinfo* of each ICS protocol, we further evaluated our novel fuzzing strategy based on those semantic information. We ran each fuzzing tool for 24 hours (on a single core) on each selected project based on 5 different sets of starting seeds (including the empty seed). What's more, we repeated each 24-hour experiment 5 times to establish statistical significance of results [18]. Each project is accompanied with Google AddressSanitizer (ASan)[25]. Figure 5 plots, for each project and technique, the average number of paths covered over all 25 runs at each time slot (dark central line) and 90% confidence intervals in paths covered at each time slot (shaded region around line) over the 25 runs for each 24-hour experiment.

From Figure 5, we can see that with `Polar`, AFL and AFLFast are able to achieve higher path coverage at a faster speed. Meanwhile, in our 24-hour experiments, we find that most fuzzers eventually became convergent, what we describe as the fuzzer reaching a "saturation" state. On average, `Polar`-AFL reaches the "saturation" state 3.6X faster than AFL and 1.5X faster than AFLFast. For a dedicated number of covered path, for example, 55 paths, `Polar`-AFL is about 4x, 1.1x, and 9x faster than AFL on libmodbus, IEC104 and libiec61850 respectively. `Polar`-AFLFast is about 5x, 1.2x, 12x faster than AFLFast on libmodbus, IEC104 and libiec61850 respectively. Furthermore, for libmodbus, `Polar`-AFL covered 9.1% more paths than AFL. `Polar`-AFLFast covered 12.7% more paths than AFLFast. For IEC104, `Polar`-AFL covered 19.5% more paths than AFL. `Polar`-AFLFast covered 24.1% more paths than AFLFast. For libiec61850, `Polar`-AFL achieves 31.0% more paths than AFL and `Polar`-AFLFast achieves 19.6% more paths than AFLFast.

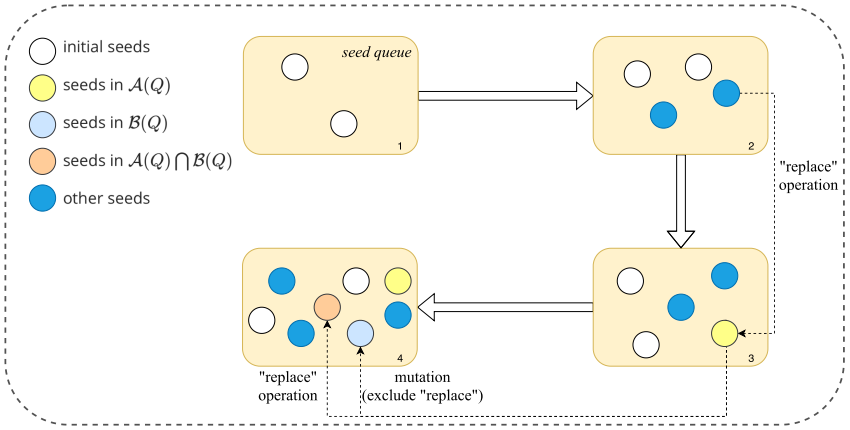


Fig. 6. Illustration for $\mathcal{A}(Q)$ and $\mathcal{B}(Q)$.

To further illustrate the effectiveness of our fuzzing strategy and experiment result impartially, we introduce two definitions as below.

Definition 1. $\mathcal{A}(Q)$. For a seed queue Q (Algorithm 1, line 1), $\mathcal{A}(Q)$ represents the set of interesting seeds generated by mutation operation `replace` in Q.

Definition 2. $\mathcal{B}(Q)$. For a seed queue Q, $\mathcal{B}(Q)$ represents the set of interesting seeds that are generated by some seed $\mathcal{S} \in \mathcal{A}(Q)$ through one or more mutation operations (including `replace`).

To better understand those two definitions, Figure 6 gives an illustration. It shows four stages of seed queue Q. The little circles in Q represent seeds and different colors of them represent their different classes as listed in the top left corner. For stage i, it may go through several mutation iterations to move to stage $i + 1$.

Definition 3. $\mathcal{N}(Q)$. For a seed queue Q in `Polar`, $\mathcal{N}(Q)$ means the impact factor about how our novel fuzzing strategy contributes to providing interesting seeds during the fuzzing process.

Intuitively, we can calculate $\mathcal{N}(Q)$ by Formula (2).

$$\mathcal{N}(Q) = \frac{|\mathcal{A}(Q)| + |\mathcal{B}(Q)| - |\mathcal{A}(Q) \bigcap \mathcal{B}(Q)|}{|Q|} \tag{2}$$

Definition 4. $\mathcal{W}(Q)$. For a seed queue Q in `Polar`, Q^- means the seed queue discards those seeds that appear in the seed queue of the base fuzzer (AFL/AFLFast). $\mathcal{W}(Q)$ means the impact factor similar to $\mathcal{N}(Q)$, and it only takes seeds in Q^- into account.

Analogously, $\mathcal{W}(Q)$ is computed by Formula (3).

$$\mathcal{W}(Q) = \frac{|\mathcal{A}(Q^-)| + |\mathcal{B}(Q^-)| - |\mathcal{A}(Q^-) \bigcap \mathcal{B}(Q^-)|}{|Q|} \tag{3}$$

Then, within the limited time budget, $\mathcal{N}(Q)$ reflects the improvement of fuzzing speed. $\mathcal{W}(Q)$ describes the increment of path covered.

Once an *interesting seed* (Algorithm 1, line 14) is generated, `Polar` will use guided seed prioritization and mutation, and apply lightweight synchronization strategy (Algorithm 4) to synchronize useful seed information, leading to the earlier appearance of more interesting seeds. Therefore, compared to the base fuzzer, `Polar` can achieve higher speed. In addition, the approach of protecting `Func.` field during mutation can also speed up the process of discovering new paths. Taking

Table 3. $\mathcal{N}(Q)$ for Each Project(Polar-AFL/Polar-AFLFast)

Project	$\lvert Q\rvert$	$\lvert\mathcal{A}(Q)\rvert$	$\lvert\mathcal{B}(Q)\rvert$	$\lvert\mathcal{A}(Q)\bigcap\mathcal{B}(Q)\rvert$	$\mathcal{N}(Q)$
libmodbus	60/66	11/18	37/17	0/7	80.0%/42.4%
IEC104	225/236	14/15	195/197	4/5	91.1%/87.7%
libiec61850-MMS	65/72	7/11	6/9	4/8	13.8%/27.8%

Table 4. $\mathcal{W}(Q)$ for Each Project(Polar-AFL/Polar-AFLFast)

Project	$\lvert Q\rvert$	$\lvert Q^-\rvert$	$\lvert\mathcal{A}(Q^-)\rvert$	$\lvert\mathcal{B}(Q^-)\rvert$	$\lvert\mathcal{A}(Q^-)\bigcap\mathcal{B}(Q^-)\rvert$	$\mathcal{W}(Q)$
libmodbus	60/66	14/31	0/6	0/8	0/6	0.0%/12.1%
IEC104	224/236	208/226	5/11	189/173	3/3	84.9%/77.1%
libiec61850-MMS	65/72	62/69	7/11	6/9	4/8	13.8%/27.8%

those interesting seeds that cannot be generated by the base fuzzer ($|Q^-|$) within the limited time budget into count, $\mathcal{W}(Q)$ objectively describes the ability of Polar to discover new paths.

Detail Analysis. To evaluate Polar using the above two indicators, we selected 1 run from 25 runs for each project and calculated $\mathcal{N}(Q)$ and $\mathcal{W}(Q)$ for them. Table 3 and 4 present the details of $\mathcal{N}(Q)$ and $\mathcal{W}(Q)$ results for Polar-AFL and Polar-AFLFast.

Statistics on libmodbus. Modbus is a stable and relatively simple ICS protocol compared with others both in format and size of the code base. Therefore, as illustrated in Figure 5, most fuzzers tend to reach the "saturation" state in an early phase. As shown in Table 2, the legal values of *funcinfo* are close to each other, which means it is easy for AFL/ AFLFast to achieve the extra-legal values through simple mutation operations such as bit flips, byte flips, and arithmetics. Consequently, the base fuzzer can generate almost the same interesting seeds as Polar did, which contributes to the small value of $|Q^-|$ and $\mathcal{W}(Q)$ in Table 4. Nevertheless, through our semantic aware mutation, Polar is able to generate those seeds earlier, which optimizes the fuzzing process as reflected at $\mathcal{N}(Q)$ in Table 3.

Statistics on libiec61850-MMS. Compared to Modbus, libiec61850-MMS owns more complex packet structures and larger variation of legal *funcinfo* values. The seeds generated by Polar have few similarities with the base fuzzer ($|Q^-|$ is basically equal to $|Q|$). Furthermore, as shown in Figure 5, Polar achieves more paths covered on account of our synchronization mechanism ($\mathcal{W}(Q)$), and explores paths at a higher speed ($\mathcal{N}(Q)$).

Statistics on IEC104. The results of experiments conducted on IEC104 show Polar's excellence not only in number of paths covered but also in the speed boost (as shown in Figure 5). The synchronization mechanism sustainedly and effectively provides interesting seeds for fuzzing course, shown in high value of $\mathcal{N}(Q)$ and $\mathcal{W}(Q)$.

In conclusion, the proposed fuzzing strategy is valuable and effective in practice. It makes a significant contribution by accelerating fuzzing and exploring more paths within a limited time budget. As the results on the three protocols show, Polar can accelerate fuzzing 3.6X and 1.5X for AFL and AFLFast on average, ranging from 1.5X to 12X faster for each path instance. Polar gains final increases in covered path with 19.9% and 18.8% for AFL and AFLFast on average, ranging from 0% to 91% for each time slot.

5.4 Previously Unknown Vulnerabilities

In real practice, Polar's two implementations, Polar-AFL and Polar-AFLFast have already detected several previously unknown security vulnerabilities in widely used implementations of libiec61850 and IEC104, 6 of which have been assigned with unique CVE identifiers. Table 5

Table 5. Vulnerabilities Exposed by Polar

Project	Type	Advisory	Total
libiec61850	heap buffer overflow	CVE-2018-18834 , CVE-2018-19185	6
	NULL pointer dereference	CVE-2018-18937, CVE-2018-19122	
	SEGV	CVE-2018-19093, CVE-2018-19121	
IEC104	stack buffer overflow	Bug-2019-0312	4
	SEGV	Bug-2019-0207, Bug-2019-0307	
	denial of service	Bug-2019-0402	

summarizes those vulnerabilities that have been confirmed and repaired. The column "Type" indicates the cause of vulnerabilities, mainly including null pointer dereference, SEGV, heap-based buffer overflow, stack-based buffer overflow and denial of service. The column "Advisory" shows vulnerability identifier information. "CVE-xxx" means it is assigned a CVE identifier, while "bug-xxx-x" means the vulnerability have been confirmed.

The bugs exposed by Polar can dramatically affect the service of devices running those ICS protocols. Taking the denial of service vulnerability(Bug-2019-0402) of IEC104 in Table 5 as an example, if this bug is made use of for destructive purposes, the server device will immediately shut down, causing the whole system to crash.

We present the denial of service vulnerability in IEC104 in detail. We analyzed this vulnerability with the GNU Project Debugger (gdb) as shown in Listing 4. It is caused by tending to call a unimplemented function (function SaveFirmware) in line 1066 (Listing 3) when processing a packet, which then leads to application crash (segmentation fault). In our experiment, Polar-AFL exposed this vulnerability 18x faster than AFL and Polar-AFLFast exposed it 4x faster than AFLFast.

In conclusion, Polar achieves faster fuzzing speed and higher path coverage, and is more effective in real vulnerability detection.

```
1065 for(i=0; i<3; i++){
1066   ret = IEC10X->SaveFirmware
           (DataLen,DataPtr,FirmwareType, Iec10x_Update_SeekAddr);
1067   if(ret == RET_SUCESS)
1068   break;
1069 }
```

Listing 3. Code Snippet of IEC104

```
Thread 2 "iec104_monitor" received signal SIGSEGV,
Segmentation fault.
(gdb) bt
#0  0x0000000000000000 in ?? ()
#1  0x00000000004087a6 in Iec104_Deal_FirmUpdate
  (asdu=0x7ffff37fe896, Len=20 '\024') at ..//IEC10X/Iec104.c:1066
#2  0x000000000040923f in Iec104_Deal_I
  (Iec104Data=0x7ffff37fe890, len=20) at ..//IEC10X/Iec104.c:1208
#3  0x000000000040972c in Iex104_Receive
  (buf=0x7ffff37fe890 "h\022~\344\202\060\200\006\016", len=41)
  at ..//IEC10X/Iec104.c:1305
#4  0x000000000040b662 in Iec104_main (arg=0x7fffffffe100)
  at main.c:139
#5  0x00007ffff6a306ba in start_thread (arg=0x7ffff37ff700)
  at pthread_create.c:333
#6  0x00007ffff676641d in clone ()
  at ../sysdeps/unix/sysv/linux/x86_64/clone.S:109
```

Listing 4. A denial of service vulnerability in IEC104

Table 6. Comparison between Polar and TaintScope

Comparison	Polar	TaintScope
Goal of Static Analysis	Filter *funcinfo* candidates and vulnerable operations	Filter checksum-based integrity check candidates and vulnerable operations
Goal of Taint Analysis	Verify *funcinfo*	Verify checksum-based integrity checks and identify hot bytes for vulnerable operations
Fuzzing Strategy	Add two strategies for function code aware and one strategy for vulnerable operations	Add one strategy for vulnerable operations (hot bytes)
Logic of Under-test-program	Unchanged	Changed
Crash Verification	No need	Required
Goal of Instrumentation	Obtain the coverage information of function code statements and vulnerable operations	Change the control flow of under-test-program to bypass integrity checks

6 RELATED WORK

Protocol Fuzzing. There are some fuzzers that are highly optimized for protocol testing such as Sulley [1], Defencis, and Peach [30]. However, most of them require format specification of protocol under testing, which causes significant manpower expense. Also, those generation-based fuzzers do not easily detect deep bugs in the source code. In contrast, given an ICS protocol to test, Polar requires no extra format specification of the protocol packet, decreasing the amount of manual work required to test implementations of ICS protocols. It extracts some protocol information automatically and utilizes it during fuzzing, giving it a wide applicability.

Grammar based fuzzing. Several fuzzing techniques have been proposed based on grammar. CSmith [42] is a fuzzer designed for C programming language and generates C programs based on randomly selected production rule in the grammar. LangFuzz [17] leverages ANTLR grammars to parse previously regression test input to code fragments and save them for recombination during seed generation.

Symbolic Execution based fuzzing. This technique has been widely applied to optimize fuzzing tools such as KLEE [4], Driller [27], SAFL [31], and CUTE [24]. Those tools apply symbolic execution to maximize code coverage by collecting constraints along a program path and generating inputs that satisfy unexplored path constraints. They are useful for generating valid packets for protocols with simple format. However, the scalability of this technique cannot be guaranteed because it can result in path explosion problem and requires strict execution of environmental support. Hence, the application of symbolic execution remains a challenge for large programs such as ICS protocols running in an industrial production environment [5].

Taint Analysis based fuzzing. Taint analysis based fuzzers exploit program data-flow features by analyzing how the program processes an input during execution. There are also many fuzzing tools that incorporate this technique. Considering how the data-flow features are used, those fuzzers can be roughly divided into two categories: (i) Some fuzzers aim to trigger security-sensitive codes or unexplored branches. For instance, Buzzfuzz [11] is directed for those components of input that affect the values in "attack point". Angora [6] utilizes taint analysis to analyze which byte offsets in the input affect the predicate of unexplored branches, and then utilize those information for targeting the unexplored branches. VUzzer [23] applies taint analysis to extract data-flow features, which allows it to determine where and how to mutate. (ii) There are also some fuzzers that apply this approach with the objective of detecting some critical points in the program at a pre-processing stage like Polar. For example, TaintScope [32] uses taint analysis

to locate checksum-based integrity checks. After obtaining their information, TaintScope injects instrumentation to the target program at the corresponding points to help bypass those integrity checks. Therefore, the control flow of program under fuzzing stage differs from the original one. Then if an input C leads to crash, TaintScope needs to repair checksum fields in C by combined concrete and symbolic execution and then feed it to the original program to verify its reproducibility. `Polar` uses the similar techniques of combined static analysis and taint analysis to locate function code. However, the main difference between `Polar` and TaintScope reflects in fuzzing stage. TaintScope makes modification to the target program while `Polar` leverages *funcinfo* to optimize fuzzing strategy. What's more, TaintScope needs additional verification to reduce false positives when crashes are discovered. Actually, `Polar` can be applied to TaintScope to make further improvements in vulnerability detection. The detail comparison between `Polar` and TaintScope are listed in Table 6.

7 CONCLUSION

In this paper, we present `Polar`, a function code aware fuzzing framework for ICS protocol vulnerability detection. Based on static code analysis and dynamic taint analysis techniques, `Polar` can locate the function code processing statement and some security-sensitive points automatically, and then utilizes those information to guide the fuzzing process. We augmented AFL and AFLFast with `Polar` and evaluated them on three widely used implementations of libmodbus, IEC104 and libiec61850. `Polar`-AFL and `Polar`-AFLFast achieve higher path coverage at a faster speed and have exposed 10 previously unknown bugs, 6 of which have been assigned with unique CVE identifiers. `Polar` is fully automatic and can also be applied to many other fuzzers, such as FairFuzz[19], for further improvement, especially when the format of protocol packet is unavailable. Currently, our present implementation of `Polar` relies on the source code of ICS protocols. Theoretically, it can be also extended to test binary forms of ICS protocols, and we leave it as our future work.

REFERENCES

[1] Pedram Amini and Aaron Portnoy. 2012. Sulley. (2012). https://github.com/OpenRCE/sulleyAccessed August 22nd, 2017.

[2] IEEE Standards Association. Accessed June 3rd, 2019. IEEE C37.118. *Website*. https://standards.ieee.org/standard/C37_118_1-2011.html.

[3] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. 2016. Coverage-based greybox fuzzing as Markov chain. In *ACM Conference on Computer and Communications Security*.

[4] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. 2008. KLEE: Unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*.

[5] Cristian Cadar and Koushik Sen. 2013. Symbolic execution for software testing: Three decades later. *Commun. ACM* 56 (2013), 82–90.

[6] Peng Chen and Hao Chen. 2018. Angora: Efficient fuzzing by principled search. *2018 IEEE Symposium on Security and Privacy (SP)* (2018), 711–725.

[7] Yuanliang Chen, Yu Jiang, Fuchen Ma, Jie Liang, Mingzhe Wang, Chijin Zhou, Zhuo Su, and Xun Jiao. 2018. EnFuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. *arXiv preprint arXiv:1807.00182* (2018).

[8] Clang. Accessed April 5th, 2019. LLVM dataFlowSanitizer. *Website*. https://clang.llvm.org/docs/DataFlowSanitizer.html.

[9] dj chen. Accessed April 5th, 2019. IEC104. *Website*. https://github.com/airpig2011/IEC104.

[10] Ying Fu, Meng Ren, Fuchen Ma, Heyuan Shi, Xin Yang, Yu Jiang, Huizhong Li, and Xiang Shi. 2019. EVMFuzzer: Detect EVM vulnerabilities via fuzz testing. (2019).

[11] Vijay Ganesh, Tim Leek, and Martin C. Rinard. 2009. Taint-based directed whitebox fuzzing. *2009 IEEE 31st International Conference on Software Engineering* (2009), 474–484.

[12] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, Heyuan Shi, and Jiaguang Sun. 2018. Vulseeker-pro: Enhanced semantic learning based binary vulnerability seeker with emulation. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 803–808.

[13] MZ Automation GmbH. Accessed April 5th, 2019. libiec61850. *Website*. https://github.com/mz-automation/libiec61850.

[14] Patrice Godefroid, Adam Kiezun, and Michael Y. Levin. 2008. Grammar-based whitebox fuzzing. In *PLDI*.
[15] Patrice Godefroid, Michael Y. Levin, and David A. Molnar. 2008. Automated whitebox fuzz testing. In *NDSS*.
[16] Jianmin Guo, Yu Jiang, Yue Zhao, Quan Chen, and Jiaguang Sun. 2018. Dlfuzz: Differential fuzzing testing of deep learning systems. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 739–743.
[17] Christian Holler, Kim Herzig, and Andreas Zeller. 2012. Fuzzing with code fragments. In *USENIX Security Symposium*.
[18] George Klees, Andrew Ruef, Benji Cooper, Shiyi Wei, and Michael Hicks. 2018. Evaluating fuzz testing. In *ACM Conference on Computer and Communications Security*.
[19] Caroline Lemieux and Koushik Sen. 2018. FairFuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage. In *ASE*.
[20] Jie Liang, Yu Jiang, Yuanliang Chen, Mingzhe Wang, Chijin Zhou, and Jiaguang Sun. 2018. Pafl: Extend fuzzing optimizations of single mode to industrial parallel mode. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 809–814.
[21] Barton P. Miller, Lars Fredriksen, and Bryan So. 1990. An empirical study of the reliability of UNIX utilities. *Commun. ACM* 33 (1990), 32–44.
[22] StÃ©phane Raimbault. Accessed April 5th, 2019. libmodbus. *Website*. https://github.com/stephane/libmodbus.
[23] Sanjay Rawat, Vivek Jain, Ashish Jith Sreejith Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware evolutionary fuzzing. In *NDSS*.
[24] Koushik Sen, Darko Marinov, and Gul A. Agha. 2005. CUTE: A concolic unit testing engine for C.
[25] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. 2012. AddressSanitizer: A fast address sanity checker. In *USENIX Annual Technical Conference*.
[26] Heyuan Shi, Runzhe Wang, Ying Fu, Mingzhe Wang, Xiaohai Shi, Xun Jiao, Houbing Song, Yu Jiang, and Jiaguang Sun. 2019. Industry practice of coverage-guided enterprise linux kernel fuzzing. (2019).
[27] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Krügel, and Giovanni Vigna. 2016. Driller: Augmenting fuzzing through selective symbolic execution. In *NDSS*.
[28] Michael J. Sutton, Adam Greene, and P. Amini. 2007. Fuzzing: Brute force vulnerability discovery.
[29] Tool. Accessed April 5th, 2019. AFL-Clang-Fast. *Website*. https://github.com/mirrorer/afl/blob/master/llvm_mode/README.llvm.
[30] Tool. Accessed April 5th, 2019. Peach Fuzzing Platform. *Website*. https://www.peach.tech.
[31] Mingzhe Wang, Jie Liang, Yuanliang Chen, Yu Jiang, Xun Jiao, Hao Liu, Xibin Zhao, and Jia-Guang Sun. 2018. SAFL: Increasing and accelerating testing coverage with symbolic execution and guided fuzzing. *2018 IEEE/ACM 40th International Conference on Software Engineering: Companion (ICSE-Companion)* (2018), 61–64.
[32] Tielei Wang, Tao Wei, Guofei Gu, and Wei Zou. 2010. TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection. *2010 IEEE Symposium on Security and Privacy* (2010), 497–512.
[33] Website. 2017. Heartbleed - A vulnerability in OpenSSL. (2017). http://heartbleed.com/Accessed: 2017-05-13.
[34] Website. Accessed April 5th, 2019. IEC 61850. *Website*. http://libiec61850.com/libiec61850/.
[35] Website. Accessed April 5th, 2019. vulnerabilites detected by American Fuzzy Lop. *Website*. http://lcamtuf.coredump.cx/afl/.
[36] Wikipedia. Accessed April 5th, 2019. IEC104. *Website*. https://en.wikipedia.org/w/index.php?title=IEC104&redirect=no.
[37] Wikipedia. Accessed April 5th, 2019. Modbus. *Website*. https://en.wikipedia.org/wiki/Modbus.
[38] Wikipedia. Accessed June 3rd, 2019. DNP3. *Website*. https://en.wikipedia.org/wiki/DNP3.
[39] Wikipedia. Accessed June 3rd, 2019. ICCP. *Website*. https://en.wikipedia.org/w/index.php?title=Inter-Control_Center_Communications_Protocol&redirect=no.
[40] Wikipedia. Accessed June 3rd, 2019. IEC101. *Website*. https://en.wikipedia.org/wiki/IEC_60870-5.
[41] Wikipedia. Accessed June 3rd, 2019. Profinet. *Website*. https://en.wikipedia.org/wiki/PROFINET.
[42] Xuejun Yang, Yang Chen, Eric Eide, and John Regehr. 2011. Finding and understanding bugs in C compilers. In *PLDI*.
[43] Michal Zalewski. 2015. American fuzzy lop. (2015).

Received April 2019; revised June 2019; accepted July 2019