ICS Protocol Fuzzing: Coverage Guided Packet Crack and Generation

Zhengxiong Luo
KLISS, BNRist, School of Software
Tsinghua University, China
luozx19@mails.tsinghua.edu.cn

Feilong Zuo
KLISS, BNRist, School of Software
Tsinghua University, China
zuofl19@mails.tsinghua.edu.cn

Yuheng Shen
KLISS, BNRist, School of Software
Tsinghua University, China
syh1308@gmail.com

Xun Jiao
Department of Electrical and Computer Engineering
Villanova University, USA
xun.jiao@villanova.edu

Wanli Chang
Department of Computer Science
University of York, UK
wanli.chang@york.ac.uk

Yu Jiang*
KLISS, BNRist, School of Software
Tsinghua University, China
jiangyu198964@126.com

***Abstract*—Industrial Control System (ICS) protocols play an essential role in building communications among system components. Recently, many severe vulnerabilities, such as Stuxnet and DragonFly, exposed in ICS protocols have affected a wide distribution of devices. Therefore, it is of vital importance to ensure their correctness. However, the vulnerability detection efficiency of traditional techniques such as fuzzing is challenged by the complexity and diversity of the protocols.**

In this paper, we propose to equip the traditional protocol fuzzing with coverage-guided packet crack and generation. We collect the coverage information during testing procedure, save those valuable packets that trigger new path coverage and crack them into pieces, based on which, we can construct higher quality new packets for further testing. For evaluation, we build `Peach*` on top of Peach, which is one of the most widely used protocol fuzzers, and conduct experiments on several ICS protocols such as Modbus and DNP3. Results show that, compared with the original Peach, `Peach*` achieves the same code coverage and bug detection numbers at the speed of 1.2X-25X. It also gains final increase with 8.35%-36.84% more paths within 24 hours, and has exposed 9 previously unknown vulnerabilities.

***Index Terms*—Fuzzing, ICS Protocol, Vulnerability Detection**

I. Introduction

Industrial Control System (ICS) refers to a system combining hardware and software with network connectivity so as to support critical infrastructure. In recent years, we have witnessed a wide adoption of ICS, including energy, transportation, communications, etc. To meet the demand of the developing industry, there is a trend towards higher openness of ICS, with an increasing number of ICS components available on the Internet. However, this openness inevitably makes ICS, primarily due to greater awareness of ICS protocols, easy prey for attackers who aim at compromising and controlling those ICS devices. More concretely, ICS protocol is designed to acquire measurements/status of remote physical devices and control them via packets carrying special commands. The loose protection of these protocols presents ICS to the miscreants low-hanging fruits. Those severe security vulnerabilities revealed in ICS protocols, such as Stuxnet [1], DragonFly [2], and their evolutions, have affected a wide distribution of devices. Hence, guaranteeing the correctness of those protocols is of imminent need.

Many techniques have been proposed to ensure the security of those ICS protocols. Fuzzing, as an automated software testing technique, has emerged as one of the most effective techniques for detecting security vulnerabilities in real-world software. Given the target program with parameters, fuzzers work as follows: generating malformed inputs (as for ICS protocol programs, the protocol packet can be considered as the input), feeding them to the program and looking for abnormal behaviors such as crashes or hangs. Two main approaches are utilized to generate those malformed inputs: data mutation and data generation. Mutation-based fuzzers, such as American Fuzzy Lop (or simply AFL) [3], generate new inputs by randomly mutating existing inputs, while generation-based fuzzers, including Peach [4] and Sulley [5] for protocol, construct inputs by leveraging the knowledge of format specification provided by users. Mutation-based fuzzers are popular due to their ease-of-use and fantastic vulnerability-detecting power. Nevertheless, lacking format specification, mutation-based fuzzers can easily get bogged down because the validity verification code is a significant time sink for them. Those applications that process highly-structured inputs, such as protocol programs, make it a small probability of success for them to discover vulnerabilities deep in the program state space. As opposed to mutation-based fuzzers, generation-based fuzzers are capable of generating valid inputs by utilizing the input model, which specifies the format of the data chunks and integrity constraints. Those generated valid inputs manage to carry the path exploration beyond the parser code so that it is more likely to discover vulnerabilities deep in the program's processing logic.

In practice, those generation-based fuzzers such as Peach have exposed a great deal of vulnerabilities in ICS protocols. Even so, as for fuzzing of complex ICS protocols, there remain two challenges heavily limiting their effectiveness: (i) despite awareness of input structure, due to lack of rational utilizable way, existing fuzzers discard those previously generated valuable inputs which achieve new code coverage; and (ii) the random and pointless generation strategy makes it less likely to produce high-quality inputs that are capable of digging into deep paths of protocol state space.

To tackle these problems, we present `Peach*`, an automated fuzzing tool targeted for ICS protocol. The key innovation of `Peach*` is that, instead of speeding up input generation process to produce more inputs as some existing fuzzing approaches, it proposes a novel utilizable way to leverage those previously generated inputs so as to generate more high-quality inputs. Through investigation of diverse ICS protocols, we found that the construction rules of different types of protocol packets may have something in common: some chunks, that belong to different types of ICS protocol packets, may conform to similar/same construction rules. Moreover, those different types of packets usually exercise different program traces in the protocol application. Based on this feature, `Peach*` is designed as follows: (i) Empowered by instrumentation, `Peach*` monitors the

*Yu Jiang is the correspondence author.

program execution path taken by each generated input, and identifies those inputs that contribute to new code coverage. (ii) To learn from the success of those valuable inputs, `Peach*` constructs a corpus by cracking them into pieces based on the information of file format. These pieces can be used as donors to rule out some meaningless repetitions of path exploration. (iii) To this end, `Peach*` applies a novel semantic aware generation strategy. Instead of starting from scratch, it derives new inputs by selecting appropriate pieces from the constructed corpus in preference to instantiation from input model.

We implemented `Peach*` on top of Peach and evaluated its performance on several well-fuzzed and open-source implementations of widely-used ICS protocols – Modbus [6], DNP3 [7] and so on. Experimental results demonstrate that, compared with the original Peach, `Peach*` outperforms in terms of fuzzing speed (1.2X-25X, an average of 5.7X) as well as path covered (8.35%-36.84%, an average of 27.35% increase) within a time limit of 24 hours. Furthermore, along with the coverage improvement, `Peach*` has already exposed 9 previously unknown vulnerabilities in those well-known protocols, most of them are on the attack surface thus security-critical.

II. Generation-Based Fuzzing

Fuzzers treat input file as a vector of input bytes thus the modifications on the seed (the test case generated by fuzzers are also called "seed") file mainly concentrate on bits/bytes such as bit flip and splice, etc. For better effectiveness of protocol fuzzing, generation-based fuzzers work on the file structure that is organized as a tree where individual nodes are called chunks and different chunks conform to its own format specification described in the configuration file (e.g., Peach Pit [4] for Peach). Figure 1 shows a simple data model which contains four attributes: `ID, Size, Data`, and `CRC`. Specifically, `Data` consists of three individual chunks. The `Size` field is a variant that is computed by the *Relation* function `sizeof` and carries the size of `Data` field. The `CRC` field supports the error check mechanism by the *Fixup* function `Crc32Fixup`.

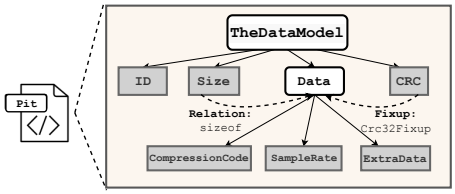


Fig. 1: Simple data model $\mathcal{M}$ used in Peach, illustrated as a tree.

Those generation-based fuzzers construct seeds or packets by leveraging the above tree format. Algorithm 1 provides an overview of the process. In the beginning, the fuzzer is provided with a format specification $\mathcal{G}$, and the detailed data model set can be extracted from it (line 2, one format specification usually contains several data models, used for producing different types of valid inputs). Then the fuzzer works in a continuous loop unless timeout or aborted (lines 3-12). For each iteration, it works as follows: it first selects one data model $\mathcal{M}$ from set (line 4), and then analyzes the chunks required to generate by traversing the format tree specified by $\mathcal{M}$ and collects the individual nodes as implemented in method ANALYZE (line 5). In general, those chunks adhere to specified data types such as `String`, `Number`, etc, and they are generated separately based on their data types in conjunction with given functions (e.g., *Relation*, *Fixup* in Figure 1) with params, and new seed is thus produced by jointing them in the order declared in $\mathcal{M}$ (lines 7-9). The method GENERATE implements data generation based on pre-defined rules. For example, Peach implements generation by those `Mutators` that are designed for different data types. As a general view, `Mutator` generates data in these ways: random generation, mutation on default value and mutation on existing chunks (those from user-provided initial seeds or previously generated seeds). New generated seed is further utilized to run the target application and those seeds that crash or hang the protocol program are recorded for further processing (lines 10-12).

Algorithm 1: Generation-Based Fuzzing

Input: $\mathcal{G}$: input model specified by format specification
Input: $\mathcal{P}$: program under test
Output: $C_{\text{✗}}$: seeds that crash or hang the program $\mathcal{P}$

1 $C_{\text{✗}} \leftarrow \varnothing$
2 $\mathcal{S}_{\mathcal{M}} = \text{EXTRACTDATAMODEL}(\mathcal{G})$ `// Data Model Set`
3 **while** *true* **do**
4 $\quad \mathcal{M} \leftarrow \text{CHOOSE}(\mathcal{S}_{\mathcal{M}})$
5 $\quad Chunks \leftarrow \text{ANALYZE}(\mathcal{M})$
6 $\quad seed \leftarrow null$
7 $\quad$ **for** $Chunk \in Chunks$ **do**
8 $\quad\quad component \leftarrow \text{GENERATE}(\mathcal{M}, Chunk)$
9 $\quad\quad seed \leftarrow \text{JOINT}(seed, component)$
10 $\quad Results \leftarrow \text{RUNTARGET}(\mathcal{P}, seed)$
11 $\quad$ **if** $\text{CRASH}(Results)$ *or* $\text{HANG}(Results)$ **then**
12 $\quad\quad C_{\text{✗}} \leftarrow C_{\text{✗}} \bigcup \{seed\}$

III. Motivation

Despite awareness of input structure, some research infers that, due to the unique random generation strategy, generation-based fuzzers such as Peach and Defensics [8] may not perform very well on complex ICS protocols or reveal the bugs hidden in the deep paths [9]. Since the generation of test cases is inherently random, we can consider the generation-based fuzzing as the following model: given the format specification, equivalently, those fuzzers are provided with the universal set $\mathcal{S}_{\mathcal{I}}$ of all legal seeds. In each iteration of input generation, they can be regarded as choosing one seed from $\mathcal{S}_{\mathcal{I}}$ randomly as the program input. Hence, in theory, if given enough time and resources, those fuzzers are able to enumerate all possible situations exhaustively and detect all potential bugs. However, this ideal case is usually unreachable as the set $\mathcal{S}_{\mathcal{I}}$ can be infinite and the budgets are constrained forever. It is our hope to make intelligent design decisions with optimal strategies wherever possible. Through investigation of diverse ICS protocols, we found that it is possible to augment existing generation-based fuzzers with some guided information for further improvement.

ICS protocol is designed for a specific domain – industry control, thus it possesses some specific features compared with other common internet protocols. From the perspective of the field in packet, these protocols employ a special field to identify different packets, called "function code" field (or "opcode" field) that encodes the instruction to be performed by the devices, such as *restart*, *write inner register*, *report self status* and so on. More importantly, after diving into the input model used in ICS protocols, we found that different types of packets would trigger different traces, but they would share similar data chunks which would trigger similar parsing code. Hence, we can replace the traditional data model based random generation with the coverage guided packet crack and generation to improve the fuzzing speed and depth. We use Figure 2 to illustrate our insights in detail.

For a data model $\mathcal{M}$, the organizing mode as a tree can be translated into the linear model $M_{\mathcal{L}}$ similar to Figure 2(a), where the individual nodes of tree take up in line with the order specified in $\mathcal{M}$. As a consequence, Figure 2(a) shows the organization of three types of packets with different opcode values. Each rectangle represents the construction rule of the chunk as defined in $\mathcal{M}$. In light of our investigation, we found that some chunks that belong to different types of packets may conform to similar or same construction rules (chunks with the same color in Figure 2(a) means they conform to similar construction rules). For instance, as shown in Figure 2(a), the chunks generated by `Rule` α_1 can also be parsed by `Rule` α_2 smoothly in most cases, and vice versa. In particular, the rectangle with dashed outline refers to `Fixup` mechanism such as `Crc32Fixup` shown in Figure 1. This feature is also reflected in the Control Flow Graph (CFG) of the packet processing part in protocol program and Figure 2(b) shows the detail. Different types of packets can cause different execution traces (indicated as different colors), but those traces may contain some shared code blocks used to decode corresponding chunks. The code blocks of different traces may not be shared, but they may act similarly because they are used to parse those chunks generated by similar rules. Taking the packets in Modbus [6] for example, those operations such as calculating the mapping address, calculating the data to write, constructing a response message are all required for the packets of `write single register` and `write single coil`. The only difference between them is the place to write, one for register while another for coil.

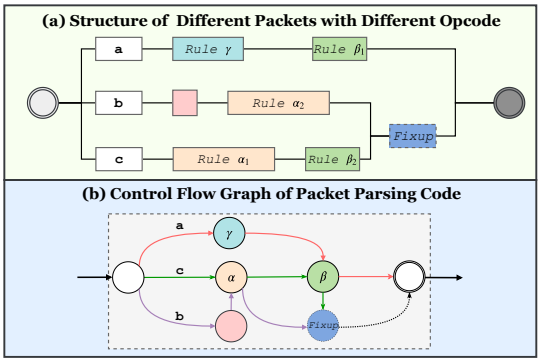


Fig. 2: `Peach*` insights. Crack those packets that trigger new path into pieces, and construct higher quality new packets based on these pieces to trigger more new paths in the control flow graph.

Based on this feature, we argue that the specification in the data model can be further exploited for optimization. Since different types of packet represent different commands and these different commands require different parameters, the construction rules of these packets can be regarded as different data models. Assuming that there are n types of packets (usually denoted by the legal values of the opcode field) in some ICS protocol, their data models can be donated as $\mathcal{M}_1, \mathcal{M}_2, ..., \mathcal{M}_n$. If one seed $\mathcal{I}_v$ generated by $\mathcal{M}_i$ is valuable (usually labelled as valuable when a new path is triggered), then, based on the similarity of different chunks, the cracked chunks of $\mathcal{I}_v$ can be utilized to help optimize path exploration when generating inputs using other data models $\mathcal{M}_j (1 \leq j \leq n, j \neq i)$. Based on this, we can implement a more efficient packet generation strategy.

IV. System Design

In this section, we first introduce the workflow of `Peach*`. Then, we present the details of each component.

A. `Peach*` Overview.

To illustrate the workflow and detailed design of `Peach*` better, we first introduce the following two definitions.

Definition 1: `Instantiation Tree`. The `Instantiation Tree` (or `InsTree` for short) of the data model $\mathcal{M}$ has the same structure as the data model tree (e.g. Figure 1 shows a data model tree). The only difference is that, the individual nodes of data model tree are construction rules of corresponding chunks, while the homologous individual nodes of `InsTree` are instantiations of these construction rules, namely realistic data chunks.

Definition 2: `Puzzle`. One `puzzle` refers to a combination of all the individual nodes of any sub-tree of the `InsTree`, and these chunks are organized in order as described in the data model. Suppose that the individual nodes of the tree shown in Figure 1 are all replaced with realistic data chunks, then those individual nodes `ID`, `Size` are both `puzzles`, and the combination of those atomic chunks `CompressionCode`, `SampleRate` and `ExtraData` in order is also one `puzzle`.

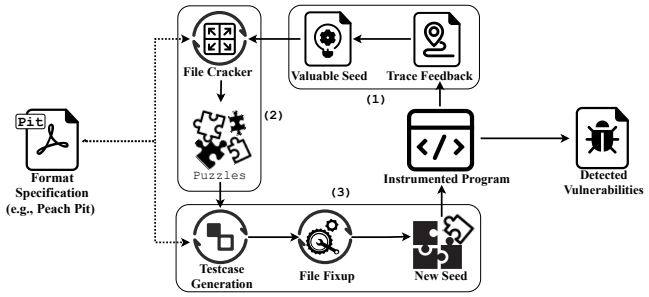


Fig. 3: `Peach*` fuzzer overview, including coverage based valuable packets identification, packet cracking to get useful puzzles, and semantic aware new packets generation with necessary fixup.

Figure 3 describes the system overview of `Peach*`, which works as follows: the fuzzing routine takes the same input as those traditional generation-based fuzzers, a target protocol program and packet format specification, and then runs in a continuous loop. Initially, the `puzzle` corpus is vacant, and `Peach*` generates new seed $\mathcal{I}$ in the *Testcase Generator* module by leveraging the format specification alone with the inherent generation strategies of Peach, and the *File Fixup* module is needless at this stage. Once a new seed is generated, it is then used to run the target program for potential vulnerability detection. Moreover, lightweight instrumentation is inserted into the target program to obtain coverage information, based on which, `Peach*` is able to identify $\mathcal{I}$'s contribution to new code coverage. `Peach*` retains $\mathcal{I}$ if it is valuable, and then cracks $\mathcal{I}$ into `puzzles` by the *File Cracker* module based on the format specification. In this case, the `puzzle` corpus becomes available, and the *Testcase Generator* module employs a new generation strategy (called "semantic aware generation strategy") to take full advantage of this corpus. Meanwhile, the seeds generated by this new strategy may be illegal, and the *File Fixup* module is used to repair it to ensure validity. In the design of `Peach*`, the `puzzle` corpus is available by *File Cracker* only in the case that a valuable seed is detected based on the feedback collected from the target program. Thereafter, in the following iteration of seed generation, the proposed generation strategy will be employed. Otherwise, the generation strategy to apply remains inherent. `Peach*` consists of three main components as shown in Figure 3: (1) collecting coverage information and detecting valuable seed; (2) cracking valuable seed into `puzzles`; (3) applying

semantic aware generation with necessary file repairment. In the following sections, we dive into details of the design of each part.

B. Valuable Seeds Identification

Code coverage information is a feedback which is wildly used in traditional software unit test generation [10] and has been confirmed effective. Hence, as shown in the component (1) in Figure 3, we try to augment the traditional generation-based fuzzers with feedback loop, and use the code coverage as the feedback to evaluate whether a seed is valuable. We use the edge coverage and obtain this information by injecting instrumentation at branch points in the target protocol program as follows:

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

The variable `cur_location`, with random value generated during compilation time, is used to specify the basic block. The `shared_mem[]` array is a shared memory region used to track coverage. $(A >> 1) \oplus B$, a kind of hash for simplification, can be thought of as an edge from basic block A to B, and the byte set at this position in `shared_mem[]` records the times of transition from A to B. Empowered by instrumentation, `Peach*` can track the execution flow exercised by the new generated seed $\mathcal{I}$, and determine whether $\mathcal{I}$ reaches a new program execution state that has not appeared before. If so, the seed $\mathcal{I}$ is considered valuable and would be cracked into `puzzles` to improve the generation of new seeds.

C. Packet Cracker

Algorithm 2: File Cracker Algorithm

Input: $\mathcal{G}$: input model specified by format specification
Input: $\mathcal{I}_v$: valuable seed detected
Output: $Corpus$: set of `puzzles` after crack

1 **Algorithm**
2 $Corpus \leftarrow \varnothing$
3 $\mathcal{S}_{\mathcal{M}} = \text{ExtractDataModel}(\mathcal{G})$
4 **for** $\mathcal{M} \in \mathcal{S}_{\mathcal{M}}$ **do**
5 $InsTree \leftarrow \text{Parse}(\mathcal{M}, \mathcal{I}_v)$
6 **if** $\text{Legal}(InsTree)$ **then**
7 $Root \leftarrow \text{GetRoot}(InsTree)$
8 `DFS` $(Root)$
9 **Procedure** `DFS` $(TreeNode)$
10 $SubTreePuzzle \leftarrow \varnothing$
11 $Children \leftarrow \text{GetChildren}(TreeNode)$
12 **if** *$Children$ is empty* **then**
13 $SubTreePuzzle \leftarrow \text{GetContent}(TreeNode)$
14 **else**
15 **for** $Child \in Children$ **do**
16 $SubTreePuzzle \leftarrow$ $\text{Joint}(SubTreePuzzle,$ `DFS` $(Child))$
17 $Corpus \leftarrow Corpus \bigcup SubTreePuzzle$
18 **return** $SubTreePuzzle$

To make the best of the detected valuable seed $\mathcal{I}_v$, we design the *File Cracker* module to split it into `puzzles`. Algorithm 2 provides an overview of this module. Given the format specification $\mathcal{G}$, we first extract the detailed data model set (line 3) and try to use these models to crack $\mathcal{I}_v$ one by one (line 4). For the selected model $\mathcal{M}$, the method Parse implements the parse of $\mathcal{I}_v$ and obtains `Instantiation Tree` $InsTree$ (line 5). If $InsTree$ is legal (line 6), we use Depth First Search (DFS) algorithm to traverse $InsTree$ and collect the `puzzle` corpus from each sub-tree (lines 9-18). For the sub-tree with root $TreeNode$, if it is an individual node, the `puzzle` represented by this sub-tree is the chunk content of itself (lines 12-13); otherwise, if it is an internal node, then the `puzzle` is the combination of the `puzzles` of its child nodes (lines 14-16). For instance, assuming that the tree shown in Figure 1 is `Instantiation Tree`, then the `puzzle` of sub-tree with root `ID` is itself and the `puzzle` of sub-tree with root `Data` is the combination of `CompressionCode`, `SampleRate` and `ExtraData`. To ensure the order of chunks in `puzzle`, the order to traverse (line 15) should adhere to the format specification.

After obtaining the `puzzle` corpus of the detected valuable seed $\mathcal{I}_v$, the new proposed generation strategy – semantic aware generation strategy, will be applied in the following iteration of test case generation based on this high-quality corpus.

D. Semantic Aware Generation and File Fixup

As mentioned in the motivation section, though different types of packets exercise different program paths, there are some chunks in them conforming to same/similar construction rules and parsing code block. Motivated by this, we argue that, a valuable seed $\mathcal{I}_v$ with one value of the opcode can be used to optimize seed generation for other values of the opcode. More specifically, the `puzzles` produced by cracking $\mathcal{I}_v$ can be donated to the data model that used to generate packets with other values of the opcode. Algorithm 3 provides our semantic aware generation strategy.

Algorithm 3: Semantic Aware Generation Algorithm

Input: $\mathcal{M}_{\mathcal{L}}$: linear data model
Input: $Corpus$: `puzzle` corpus
Output: $Seeds$: set of seeds generated

1 **Algorithm**
2 $Seeds \leftarrow \varnothing$
3 $Size \leftarrow \text{GetSize}(\mathcal{M}_{\mathcal{L}})$
4 `Construct` $(1, Size, \varnothing)$
5 **Procedure** `Construct` $(CurPos, Size, CurSeed)$
6 **if** $\text{Equal}(CurPos, Size + 1)$ **then**
7 $Seeds \leftarrow Seeds \bigcup CurSeed$
8 **else**
9 $Rule \leftarrow \text{GetConstructionRule}(\mathcal{M}_{\mathcal{L}}, CurPos)$
10 $Candidates \leftarrow \text{GetDonor}(Rule, Corpus)$
11 **if** *$Candidates$ is not empty* **then**
12 **for** $Candidate \in Candidates$ **do**
13 `Construct` $(CurPos + 1, Size,$ $\text{Joint}(CurSeed, candidate))$
14 **else**
15 `Construct` $(CurPos + 1, Size,$ $\text{Joint}(CurSeed, \text{Generate}(Rule)))$
16 **return**

For Algorithm 3 in detail, given the linear data model $\mathcal{M}_{\mathcal{L}}$ and constructed `puzzle` corpus, we first get the number of chunks required to generate (line 3) and then construct seeds in a recursive way as implemented in the procedure `Construct`: we construct each data chunk in order until all of their values are assigned (line 6). For each chunk to generate, the construct rule is extracted from

$\mathcal{M}_{\mathcal{L}}$ (line 9) and the subset of $Corpus$ containing `puzzles` that conform to this rule is marked as $Candidates$ (line 10). Those `puzzles` in $Candidates$ are used to initialize this field one by one when $Candidates$ is not empty (lines 11-13). Otherwise, we use the inherent $Rule$ to provide the content of this field (lines 14-15). Suppose there are only two fields, namely a and b, in some ICS protocol packet, and the size of the set $Candidates$ of field a is p while the size of $Candidates$ of b is q (if the set $Candidates$ is empty, then the size is considered as 1 because the inherent construction rule will be applied). Then $p \times q$ new seeds will be generated by our generation strategy.

After obtaining new seeds, we need to apply necessary file fixup for them. Protocol packets usually employ integrity constrains, such as size-of, length-of and checksums, to ensure data integrity. The integrity of the seeds generated by our strategy may be compromised, therefore, we design the *File Fixup* module to re-establish their integrity. Actually, we can use the *Fixup* and *Relation* mechanism of Peach directly for file repair.

V. Evaluation

We have implemented a prototype of `Peach*` based on the widely used generation-based fuzzer Peach 3.0.202 [4]. In *Valuable Seeds Identification* part, we implemented `Peach*-clang`, which is a wrapper of clang and enables inserting an additional LLVM pass to under-test-program to collect coverage information. The *Packet Cracker* part added a memory to store all the chunks of the seed being generated and utilized this stored chunk corpus to build `puzzle` corpus. For the *Semantic Aware Generation and File Fixup* part, we added a new generation strategy for Peach as stated in Section IV-D, and the *File Fixup* module is based on the fixup mechanism of Peach.

We evaluate `Peach*` experimentally to answer the following two research questions:

1) *Is `Peach*` more efficient than Peach, when augmented with the coverage guided packet crack and generation?*
2) *Is `Peach*` effective in exposing previously unknown vulnerabilities in real-world ICS protocol applications?*

A. Experiment setup

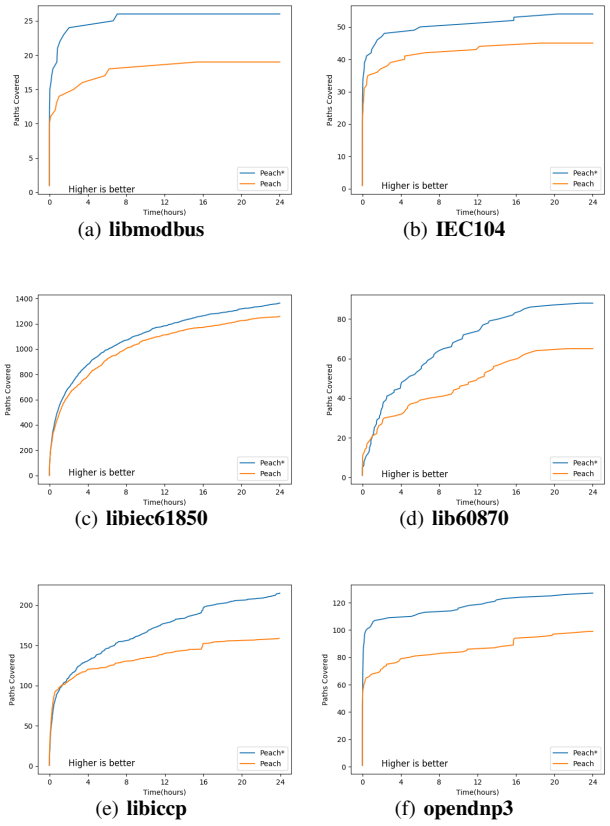
We evaluated the performance of `Peach*` on several open-source ICS protocol projects, including libmodbus [6], IEC104 [11], libiccp [12], opendnp3 [7], libiec61850 [13], and lib60870 [14]. Those ICS protocols with different code scale are widely applied in various industrial scenarios at present.

We used the path coverage achieved and the number of detected unique bugs as metrics. The first metric is commonly used to measure the effectiveness of fuzzers while the second metric indicates the ability to detect vulnerabilities. For comparison, we added path coverage framework on both `Peach*` and Peach to obtain the coverage ratio while fuzzing. We used the existing pit file of Peach, which specifies the input format and is requisition for Peach execution. In real practice, the input model does not have to be elaborate, explaining those key information such as coarse-grained data chunk information is enough.

B. Fuzzing Performance

We ran each fuzzing tool on each selected project with a 24-hour time budget and repeated each 24-hour experiment 10 times to establish statistical significance of results. Figure 4 plots the average number of paths covered for each project and fuzzing tool.

Figure 4 demonstrates that, on all the selected projects, `Peach*` achieves the upper bound in path coverage with a rapid increase, showing a sizeable lead on these projects. These projects are representative because they are widely used and also own diverse complexity and code scale, which is also reflected in the paths covered after 24 hours as shown in Figure 4 – `Peach*` as well as Peach can achieve thousands of paths on project libiec61850, while hundreds of paths for opendnp3 and dozens of paths for IEC104. Take the result of libiec61850 as an example, because of the largest scale of project code, the paths covered of libiec61850 is increasing all the time during our 24-hour experiment. Still, `Peach*` achieves an average of 8.35% more paths covered than Peach after 24 hours, showing its wide applicability and scalability.



(a) **libmodbus** (b) **IEC104**

(c) **libiec61850** (d) **lib60870**

(e) **libiccp** (f) **opendnp3**

Fig. 4: Average number of paths covered by `Peach*` and Peach within 24 hours for 10 repetitions on each ICS protocol program.

We can also infer from the figure that, both `Peach*` and Peach are effective at the beginning of execution, showing a rapid increase in path coverage. However, after a while, Peach tends to get bogged and reach a state where path coverage becomes hard to grow ever since, while the coverage-guided packet crack and generation strategy of `Peach*` can help alleviate this situation by sustainedly providing high-quality seeds. Overall, compared with the original Peach, `Peach*` achieves the same code coverage at the speed of 1.2X-25X. Within 24 hours, it also gains sustained increase with 27.35% more paths on average (up to 36.84% for single project). This is a dramatic improvement in performance because, as suggested by some prior works, after a certain amount of coverage is achieved, even small increases in code coverage can yield more vulnerability detection ability [15].

The experiment results confirm that, the coverage guided packet crack and generation strategy is valuable and makes a significant

contribution on accelerating fuzzing and exploring more paths within a limited time period. This improvement owes to Peach*'s capability that identifies valuable seeds and applies file crack along with semantic aware generation to learn from the success of these valuable seeds, making more high-quality seeds appear at an early stage.

C. Previously Unknown Vulnerabilities

Along with the coverage improvement, Peach* has detected 9 previously unknown security vulnerabilities in those extensively used open-source implementations. Table I summarizes those confirmed vulnerabilities detected by Peach* and corresponding patches from vendors are released or in progress. The column "Vulnerability Type" indicates the cause of vulnerabilities, including SEGV, heap buffer overflow and heap use after free.

TABLE I: Vulnerabilities Exposed by Peach*

Project	Vulnerability Type	Number	Status
lib60870	SEGV	3	Confirmed
libmodbus	Heap Use after Free SEGV	1 1	Confirmed
libiec_iccp_mod	SEGV Heap Buffer Overflow	3 1	Confirmed

These bugs exposed by Peach* cause potential hazards to the devices that running those ICS protocols. For example, Listing 1 illustrates a segmentation violation (SEGV) vulnerability found in the lib60870. We used GNU Project Debugger (gdb) to analyze this vulnerability, as shown in Listing 2. The bug occurs in the function CS101_ASDU_getCOT, which tries to calculate the return value using part of the received packet's ASDU field without verification. Hence, when the particular field is malformed or missing, the program will access an illegal memory location, thereby resulting in a bad address operation and leading to a segmentation fault. If this bug is exploited for malicious proposes, the servers that running this protocol may encounter a crash, causing an immediate shutdown to the whole system. This vulnerability has been confirmed and fixed by the vendor.

```
306 CS101_ASDU_getCOT(CS101_ASDU self)
307 {
308     return (CS101_CauseOfTransmission) (self->asdu[2] &
      0x3f);
309 }
```

Listing 1: Code snippet of lib60870

```
Thread 5 "simple_server" received signal SIGSEGV,
Segmentation fault.
SUMMARY: AddressSanitizer: SEGV /root/temp/iec/lib60870/
     lib60870-C/src/iec60870/cs101/cs101_asdu.c:308:47 in
     CS101_ASDU_getCOT
```

Listing 2: A segmentation violation vulnerability in lib60870

VI. Related Work

Grammar-based fuzzers generate inputs by leveraging the given context-free grammar [16]. These fuzzers are also capable of producing valid inputs within the grammar model. For example, Skyfire [17] leverages the knowledge in existing samples and given grammar model to generate well-distributed inputs. CSmith [18], a fuzzer designed for C programming language, generates C programs based on randomly selected production rules in the grammar. However, due to the limitation of context-free grammar, these fuzzers have trouble in encoding those integrity constraints such as size-of, checksums, all of which are wildly used in ICS protocol programs.

Symbolic execution has been widely utilized to optimize fuzzing tools such as KLEE [19] and MoWF [20]. To maximize code coverage, these tools utilize symbolic execution by collecting constraints along unexplored program path and generating inputs that satisfy those constraints. For example, MoWF [20] leverages existing data chunks and exploits the file format as a constraint during path exploration in symbolic execution. However, the application of symbolic execution is challenged by the path explosion problem, especially for those large programs such as ICS protocols running in an industrial production environment [21].

Recently, mutation-based fuzzers [3], [22] have also been widely adopted in practice for traditional software testing. Being unaware of file format, those fuzzers remains limited in generating valid inputs and covering large regions of code, especially for protocol programs that process highly-structured packets. To deal with this, several recent research, such as Polar [9], focuses on the critical information in protocol. Polar augments mutation-based fuzzers with the function code information as well as security-sensitive points detected in protocol program to optimize fuzzing. However, it is challenging to obtain drastic effectiveness due to limited awareness of the packet format, and our work focuses on optimizing generation-based fuzzers with guided packet crack and generation.

VII. Conclusion

In this paper, we present Peach*, a coverage-guided generation-based fuzzing tool for ICS protocol vulnerability detection. Based on coverage information collected during fuzzing, Peach* is able to identify those valuable seeds that achieve new code coverage, then it constructs a corpus based on the cracked packet pieces and applies semantic aware generation strategy to optimize the input generation process. We implemented Peach* on top of Peach and evaluated it on several widely used ICS protocols such as Modbus and DNP3. Compared with the baseline Peach, Peach* manages to achieve higher path coverage at a faster speed and has exposed 9 previously unknown bugs. Peach* is fully automatic and has also been applied to many other ICS protocols such as s7comm for vulnerability detection. Our future work mainly focuses on two aspects: the first is to extend the instrumentation module by utilizing those binary instrumentation tools like PIN [23], and the second is to customize our work into other generation- or mutation-based fuzzers.

References

[1] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, 2011.
[2] N. Nelson, "The impact of dragonfly malware on industrial control systems," *SANS Institute*, 2016.
[3] M. Zalewski, "American fuzzy lop," 2015.
[4] Tool, "Peach fuzzing platform." Website, https://www.peach.tech.
[5] P. Amini and A. Portnoy., "Sulley," 2012, accessed Nov 19th, 2017. [Online]. Available: https://github.com/OpenRCE/sulley
[6] S. Raimbault, "libmodbus," https://github.com/stephane/libmodbus.
[7] A. Crain, "opendnp3," Website, https://github.com/dnp3/opendnp3.
[8] synopsys, "Defensics fuzz testing." Website, https://www.synopsys.com/software-integrity/security-testing/fuzz-testing.html.
[9] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ics protocol," in *TECS*, 2019.
[10] G. Fraser and A. Arcuri, "A large-scale evaluation of automated unit test generation using evosuite," *TOSEM*, 2014.
[11] dj chen, "Iec104," Website, https://github.com/airpig2011/IEC104.
[12] F. Covatti, "libiccp," https://github.com/fcovatti/libiec_iccp_mod.
[13] M. A. GmbH, "libiec61850," Website, Accessed Nov 19th, 2019, https://github.com/mz-automation/libiec61850.

[14] M. Automation, "lib60870," Website, Accessed Nov 19th, 2019, https://github.com/mz-automation/lib60870.
[15] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," *Commun. ACM*, 2014.
[16] P. Godefroid *et al.*, "Grammar-based whitebox fuzzing," in *PLDI*, 2008.
[17] J. Wang, B. Chen, L. Wei, and Y. Liu, "Skyfire: Data-driven seed generation for fuzzing," *2017 IEEE Security & Privacy*, 2017.
[18] X. Yang, Y. Chen, E. Eide, and J. Regehr, "Finding and understanding bugs in c compilers," in *PLDI*, 2011.
[19] C. Cadar *et al.*, "Klee: Unassisted and automatic generation of high-coverage tests for complex systems programs," in *OSDI*, 2008.
[20] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," *2016 ASE*, 2016.
[21] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Commun. ACM*, vol. 56, pp. 82–90, 2013.
[22] Y. Chen, Y. Jiang *et al.*, "Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers," in {*USENIX*} *Security*, 2019.
[23] C.-K. Luk *et al.*, "Pin: building customized program analysis tools with dynamic instrumentation," in *PLDI*, 2005.