Vulnerability Detection of ICS Protocols Via Cross-State Fuzzing

Feilong Zuo, Zhengxiong Luo, Junze Yu, Ting Chen, Zichen Xu, Aiguo Cui, Yu Jiang✉

***Abstract*—Industrial Control System (ICS) employs complex multi-state protocols to realize high-reliability communication and intelligent control over automation equipment. ICS has been widely used in various embedded fields, such as autonomous vehicle systems and power automation systems, etc. However, in recent years, many attacks have been performed on ICS, especially its protocols, like the hijacks over Jeep Uconnect and Tesla Autopilot autonomous systems, also the Stuxnet and DragonFly viruses over national infrastructures. It is important to guarantee the security of ICS protocols.**

In this paper, we present `Charon`, an efficient fuzzing platform for the vulnerability detection of ICS protocol implementations. In `Charon`, we propose an innovative fuzzing strategy that leverages state guidance to maximize cross-state code coverage instead of focusing on isolated states during the fuzzing of ICS protocols. Moreover, we devise a novel feedback collection method that employs program status inferring to avoid the restart of ICS protocol at each iteration, allowing for continuous fuzzing.

We evaluate `Charon` on several popular ICS protocol implementations, including RTPS, IEC61850-MMS, MQTT, etc. Compared with typical fuzzers such as AFL, Polar, AFLNET, Boofuzz, and Peach, it averagely improves branch coverage by 234.2%, 194.4%, 215.9%, 52.58%, and 35.18% respectively. Moreover, it has already confirmed 21 previously unknown vulnerabilities (e.g. stack buffer overflow) among these ICS protocols, most of which are security-critical and corresponding patches from vendors have been released accordingly.

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

I. Introduction

INDUSTRIAL Control System (ICS) refers to the network infrastructure where a huge amount of data and signals are exchanged between hardware endpoints at high speed and reliability. It develops from the origin Computer Control System (CCS), to the following Distributed Control System (DCS), and finally the Fieldbus Control System (FCS) in recent years[1][2]. Modern ICS employs complex multi-layer and multi-state digital communication protocols to realize the intelligent control of automation equipment. It has been widely used in various embedded fields, such as autonomous vehicle systems, power automation systems, robot control systems, etc. However, along with the development and increasing complexity of ICS, more times of attacks have been maliciously performed on ICS components[2], especially the multi-state protocols, which realize the connectivity of the whole ICS. For example, the vulnerabilities in autonomous systems of Jeep Uconnect and Tesla Autopilot once gave the chance for attackers to control the whole vehicle. Also, viruses such as Stuxnet[3] and DragonFly[4] have caused serious damage to national infrastructures. Hence, it is significantly important to guarantee the security of ICS protocols.

Fuzzing is one of the most effective techniques for security vulnerability detection in real-world programs and protocols. Given a system under test (SUT), fuzzers generate a large number of test inputs (i.e. ”seeds”) and inject them into the SUT to uncover errors. Based on how seeds are derived, fuzzers can be roughly divided into two categories: 1) mutation-based and 2) generation-based. Mutation-based fuzzers, such as American Fuzzy Lop (AFL) [5], leverage coverage feedback to find ”interesting” seeds that increase program coverage and use them as a basis for further mutations, such as random bit flips or dictionary replacement. Generation-based fuzzers, like Peach [6], Sulley [7], and Boofuzz [8], generate sequences of syntactically valid structured inputs with the help of manually written model files that specify the state model and the data models of input seeds in each state.

Despite their effectiveness, as proven by the sheer amount of bugs found in libraries and command-line utilities such as libjpeg and libxml, protocols such as SSL, FTP, etc., these state-of-the-art fuzzers encounter two major challenges when deployed on multi-state ICS Protocol implementations:

The first is how to cover the diverse processing logic for handling state transition in ICS Protocols. Many ICS protocols, like RTPS and IEC61850-MMS, use an ad-hoc decentralized publishers-subscribers model and any nodes who wish to participate in the network must go through multiple states and corresponding packets of different structures are sent in order. However, state-of-the-art fuzzers such as AFL, AFLNet[9], and Peach, are only able to focus on exploring code in *isolated* states by feedback-driven mutation or structure-aware generation. They ignore the abundant diverse situations in *state transitions*.

The second is how to improve the speed and throughput

Manuscript received April 07, 2022; revised June 11, 2022; accepted July 05, 2022. This article was presented at the International Conference on Embedded Software (EMSOFT) 2022 and appeared as part of the ESWEEK-TCAD special issue.

This research is sponsored in part by the NSFC Program (No. 62022046, 92167101, U1911401, 62021002, 62192730), the National Key Research and Development Project (No. 2019YFB1706203, No. 2021QY0604) and the MIIT Project (Design of intelligent networked vehicle based on the SOA central control).

Feilong Zuo, Zhengxiong Luo, Junze Yu, and Yu Jiang are with the KLISS, BNRist, School of Software, Tsinghua University, Beijing 100084, China (e-mails: zuofl19@mails.tsinghua.edu.cn, luozx19@mails.tsinghua.edu.cn, yujz21@mails.tsinghua.edu.cn, jiangyu198964@126.com).

Ting Chen is with the University of Electronic Science and Technology of China, Sichuan 610054, China (e-mail: brokendragon@uestc.edu.cn).

Zichen Xu is with the School of Mathematics and Computer Science, Nanchang University, Jiangxi 330000, China (e-mail: xuz@ncu.edu.cn).

Aiguo Cui is with the Godel Lab, Huawei Technologies Co., Ltd, Shanghai 200001, China (e-mail: ag.cui@huawei.com).

Yu Jiang is the corresponding author. Feilong Zuo and Zhengxiong Luo contributed equally to this research.

of ICS protocol fuzzing when using the feedback mechanism. State-of-the-art fuzzers instrument the target program by inserting the flag of coverage collection or simply restarting it for each iteration in order to collect coverage information [10], [6], [9]. This is difficult for ICS protocols, as they are usually complex and distributed. Therefore, no distinct flags can be provided at the code level after processing packets. What's more, restarting the entire system of the under-test ICS protocol is not feasible considering that it is commonly time-consuming to initialize.

To overcome these challenges, we propose `Charon`, an efficient fuzzing platform for vulnerability detection of ICS protocol implementations. 1) `Charon` utilizes *cross-state guidance* to maximize code coverage in state transitions for ICS protocols during the fuzzing procedure. For a single state $\mathcal{A}$, it constantly evolves the corresponding packet to explore as many conditions as possible in this state. Once a new condition $\mathcal{C}_{\mathcal{A}}$ is found, `Charon` propagates this new prior condition to a successive state $\mathcal{B}$. We take all conditions that have been discovered in state $\mathcal{B}$ as $Set_{\mathcal{B}}$, then for any condition $\mathcal{C}_{\mathcal{B}} \in Set_{\mathcal{B}}$, the state transition from $\mathcal{A}$ to $\mathcal{B}$ through $\mathcal{C}_{\mathcal{A}}$ to $\mathcal{C}_{\mathcal{B}}$ is brand-new. Additionally, more legal conditions of $\mathcal{B}$ can be explored due to this new prior condition in $\mathcal{A}$. The more diverse ways of transition between states, the more codes are covered, which results in more possibilities to find vulnerabilities. 2) Moreover, `Charon` integrates an intelligent *program status inferring module* which is able to infer the moment when a packet is fully consumed by the target ICS protocol and then notify `Charon` to collect the feedback. Therefore, it realizes the requirement of feedback collection while maintaining the continuous running scenario of ICS protocols instead of frequent restarts, greatly improving the overall fuzzing efficiency.

We evaluated `Charon` on several popular ICS protocol implementations, like RTPS, IEC61850-MMS and MQTT, etc. Results show that, within 24 hours, `Charon` gains more code branches than state-of-the-art fuzzers, including the most widely used general fuzzer AFL [5] and four domain-specific fuzzers designed for protocol Polar [10], AFLNET [9], Boofuzz [8], and Peach [6], averagely by 234.2%, 194.4%, 215.9%, 52.58%, and 35.18%, respectively. Furthermore, `Charon` has confirmed 21 previously unknown security-related severe vulnerabilities that may cause serious consequences if maliciously exploited. We reported the details of these 0-day vulnerabilities to the vendors and all corresponding patches have been released.

In summary, our paper makes the following contributions:

- We propose a novel fuzzing strategy for multi-state ICS protocols which realizes both cross-state guiding and continuous fuzzing.
- We implement `Charon`, which includes a cross-state guiding module to maximize code coverage in state transitions, and a program status inferring module to keep the initial continuous running scenario of ICS protocols.
- We apply `Charon` to widely-used ICS protocol implementations, and it outperforms the state-of-the-art fuzzers in terms of both code coverage and bug detection.

II. Background

A. Multiple States in ICS Protocols

ICS protocols are leveraged to dispatch commands and exchange messages between numerous endpoints in ICS. To better manage these participants and handle the huge amount of data along with signals, ICS protocols tend to be designed with complex implementations and multiple intra-system states. These states need to exactly reflect the status of endpoints or connection instances. Moreover, they stipulate what the participants or endpoints should do at specific time points. Here we use the example of the RTPS protocol to detailedly illustrate the multiple states in ICS protocols.

RTPS (Real-Time Publish Subscribe) protocol [11], proposed by Object Management Group (OMG) [12], is the standard interoperability protocol for Data Distribution Service (DDS) [13]. In recent years, RTPS has been adopted as the communication protocol in many automated driving systems (burgeoning and typical application of ICS), for example, Autoware [14], Apollo [15] and Adaptive AUTOSAR [16]. It satisfies autonomous driving's requirement for real-time and high-bandwidth communication. RTPS uses a Publishers-Subscribers (P/S) network model where some nodes are publishers which only publish content while others are subscribers who subscribe to publishers with the topic of interest. RTPS automatically encapsulates and delivers packets from publishers to subscribers when publishers publish content, without specifying destinations.

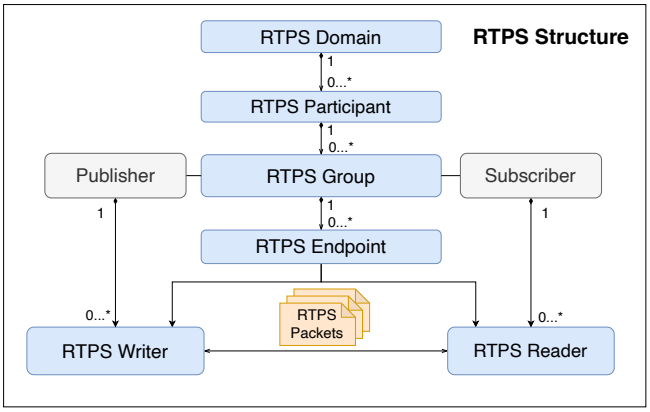


Fig. 1. Entity structure of RTPS protocol.

Entity Structure of RTPS. Figure 1 shows how entities are organized in RTPS. In the structure of RTPS, entities are separated by domains, which provide independent communication spaces. Only entities in the same domain are allowed to communicate with each other. A domain contains a set of participants each of which is a container of endpoints sharing common properties, especially those in the same address space. Endpoints may be grouped according to specific aims, e.g. a few endpoints can be combined into a group working as a publisher or subscriber. Endpoints are the basic communication units in RTPS. There are two kinds of endpoints: RTPS writer and RTPS reader. Writers can send RTPS packets to readers via underlying network protocols such as UDP.

Multiple States of RTPS. RTPS is designed in a decentralized ad-hoc model where publishers and subscribers can automatically discover each other and match up for communication by going through multiple states. Figure 2 presents a simplified example of a publisher and a subscriber successfully establishing communication. The publisher and subscriber are in different participants, so the two participants must first recognize each other in the network, which is called **Participant Discovery**. Then, they exchange information of their inner endpoints, which is called **Endpoint Discovery**. The two standard sub-protocols that specify these two discovery procedures are called *Simple Participant Discovery Protocol* (SPDP) and *Simple Endpoint Discovery Protocol* (SEDP).

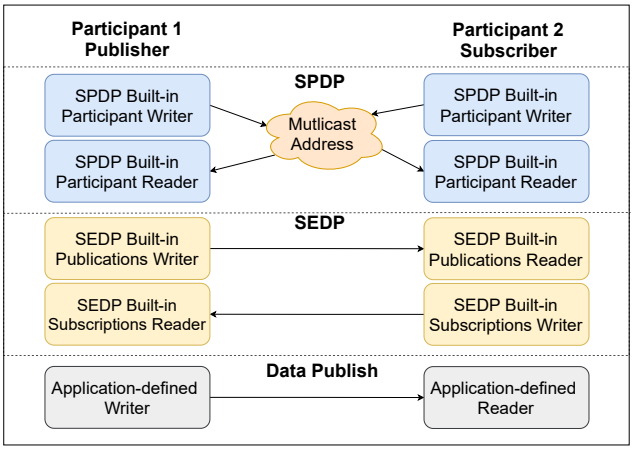


Fig. 2. A simplified example of an RTPS publisher and a subscriber, involving the state of SPDP and SEDP, and Data Publishing.

In participant discovery, each participant uses two built-in endpoints to exchange information, namely the SPDP built-in participant writer and reader. The participant writer periodically sends an SPDP packet, which includes information such as name, unicast address, etc., to pre-configured addresses (e.g. a multicast address in this example) to announce its existence. The participant reader persistently listens to the same addresses to detect the existence of other participants. After the two participants have received SPDP data from each other, they recognize each other and then come into endpoint discovery. The publisher utilizes a built-in publications writer to send an SEDP packet, consisting of topic, data type, address of application-defined endpoint to the subscriber's built-in publications reader. Similarly, the subscriber's built-in subscriptions writer delivers the packet of itself to the publisher's built-in subscriptions reader. If the publisher and subscriber share the same topic and data type, they match up and the communication is established between the application-defined writer in publisher and the application-defined reader in subscriber. Then the publisher can publish specific contents and the corresponding packet is sent through the channel.

A state refers to the situation when a publisher or subscriber has received and consumed an RTPS packet. Therefore, from this simplified example, we conclude that an RTPS publisher/subscriber has at least three ordered states: SPDP state, SEDP state, and Data Publishing/Subscribing state. Besides RTPS, many other ICS protocols are also equipped with similar multiple states for communication management, such as MQTT, IEC61850-MMS, and so on.

B. Generation-based fuzzing

We use a popular generation-based fuzzer, Peach, to demonstrate how generation-based fuzzer works in detail. Peach is widely used to fuzz diverse targets ranging from file-processing programs to network protocols in industrial control systems. To use Peach to fuzz a target system, test engineers are required to create a Pits file which defines the state model and the data models for each fuzzing state, configures the I/O interface, the ways to start and monitor the target system, etc.

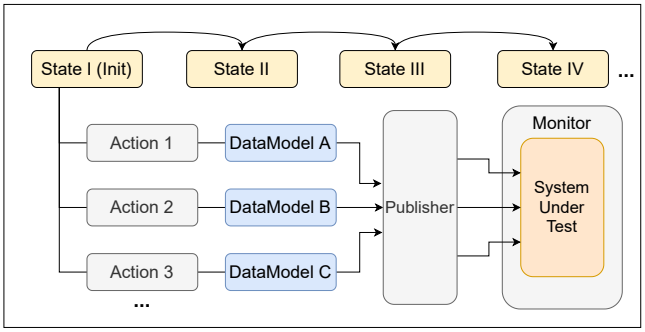


Fig. 3. A simplified example of a Pits model file used in Peach Fuzzer.

Figure 3 shows a simplified structure of Pits file and demonstrates some of the relationships between these structure elements. Peach uses `DataModel` as the base template to generate seeds. Test engineers are required to provide a detailed description of each data element in `DataModel`, including name, type, length, default values, etc. Peach utilizes a `StateModel` to manage the process of fuzzing. Each `State` represents one step of fuzzing which includes multiple actions, such as output actions and input actions. Taking commonly-used output action as an example, Peach generates a bitstream according to the generation rule of the action's `DataModel`. Then it is consumed by a configured Peach publisher, for example a UDP or TCP publisher. Finally, the publisher encapsulates the bitstream into a packet and sends it to the under-test program that is controlled by a specific local or remote monitor.

The strategy which Peach uses to fuzz a target system based on the aforementioned Pits file is briefly described in Algorithm 1. First, Peach analyzes the given Pits file to acquire necessary data including the user-defined `StateModel`, `DataModel`, selected publisher, monitor and other parameters (line 2). Then, Peach initiates the fuzzing loop. It performs fuzzing from an initial `State` and moves on to the next until finishing the last one. Multiple actions can be taken in each state. For each commonly-used output action, Peach derives a seed based on the generation rule of the action's inner `DataModel` and then the seed is published to the system under test (SUT) (lines 8-10). If the monitor finds the SUT crashes during the fuzzing loop, it records the whole sequence of seeds in this iteration for crash analysis (lines 14-15).

Algorithm 1: Fuzzing Algorithm of Peach Fuzzer

Input: $\mathcal{P}$: Input Pits file
Input: $\mathcal{S}$: System under test
Output: $C_{\text{✗}}$: Set of seeds sequences crashing $\mathcal{S}$

```
1  C✗ ← ∅
2  PitsAnalyzer(P)
3  while iteration not exceeds limitation do
4      Seed_Seed ← EmptySequence()
5      State ← GetInitState(P)
6      while State! = null do
7          for Action ∈ State do
8              D ← GetInnerDataModel(Action)
9              Seed ← SeedGeneration(D)
10             PublishToSut(Seed)
11             Seq_Seed ← Seq_Seed ⋃ {Seed}
12         State ← GetNextState(State)
13     Results ← ResultAnalyzer(S)
14     if Crash(Results) then
15         C✗ ← C✗ ⋃ {Seq_Seed}
```

III. Challenges in Fuzzing ICS Protocols

Both mutation- and generation-based fuzzers have achieved success in fuzzing libraries, command-line utilities, file processing programs, etc. However, when they attempt to fuzz ICS protocols, we need to overcome two major obstacles.

(1) Maximize code coverage for the state transition processing logic in ICS protocols. Commonly, ICS protocols work with multiple states that specify the inner status of endpoints and connection instances. For example, in the case of RTPS protocol, as described in Section II-A, an RTPS publisher/subscriber has at least three ordered states: SPDP state, SEDP state and Data Publishing/Subscribing state. However, state-of-the-art protocol fuzzers are only able to focus on exploring code coverage in *isolated* states[6][9].

```
<StateModel name="RTPS" initialState="fuzz_SPDP">
 <State name="fuzz_SPDP">
  <Action type="output" publisher="multicast">
   <DataModel ref="SPDP_mutable" />
  </Action> ...
  <Action type="changestate" ref="fuzz_SEDP" ...>
 </State>
 <State name="fuzz_SEDP">
  <Action type="output" publisher="multicast">
   <DataModel ref="SPDP_immutable" />
  </Action> ...
  <Action type="output" publisher="unicast1">
   <DataModel ref="SEDP_mutable" />
  </Action> ...
  <Action type="changestate"ref="fuzz_DATA_SUB"..>
 </State>
 <State name="fuzz_DATA_SUB"> ... </State> ...
</StateModel>
```

Listing 1. Snippet `StateModel` of Fast-RTPS

Take the fuzzing procedure of RTPS protocol implementation as an example. Listing 1 presents a snippet of a `StateModel` in the Pits file of Fast-RTPS when using Peach for vulnerabilities detection. During fuzzing, Peach operates as an RTPS publisher and the system under-test is a subscriber (usually the RTPS subscriber is the under-test program in RTPS). Three fuzzing states are described by this Pits file, corresponding to the three fuzzed states of the subscriber: SPDP, SEDP, and Data Subscribing. In each fuzzing state, several output actions are taken while only one action with mutable data model is used for exploring code coverage and prior actions with immutable ones are "bedding" data models that help under-test subscriber pass prior states. Take `fuzz_SEDP` as an example, `SEDP_mutable` is the coverage-exploring action and `SPDP_immutable` is the bedding action. Some variants of AFL, such as AFLNET [9], splice all RTPS packets together as a binary stream and fix some bits of it to help the program under-test reach the destination state, which is similar to how Peach operates. If the input packet of former states are also configured as mutable, the under-test protocol tend to be simply abled to reach the target state due to the malformed former packets.

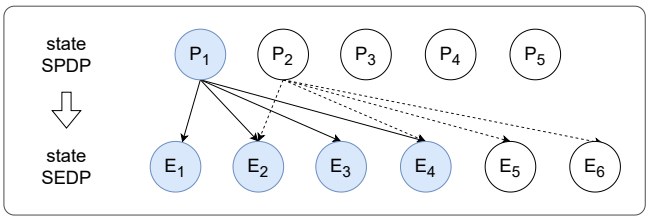


Fig. 4. State transitions which Peach covers (solid arrow).

However, both of them ignore the diverse situations in state *transition*. RTPS is a protocol with multiple states, and in each state there are various conditions about different configurations, values of inner data structures, etc. State transitions from $\mathcal{A}$ to $\mathcal{B}$ are diverse due to the diversity of both $\mathcal{A}$ and $\mathcal{B}$. But in the aforementioned fuzzing strategy, the conditions in previous state $\mathcal{A}$ are limited to one or a few and are not flexible. Figure 4 briefly shows this limitation. Take an example, the conditions of state SPDP range from P_1 to P_5 while those of SEDP range from E_1 to E_6. As for Peach, it takes the bedding immutable data model with the condition of P_1 to drive RTPS to the SPDP state and enter the target SEDP state. With the fixed P_1, the available conditions of SEDP may limited to E_1 to E_4. Therefore, Peach is only able to cover the transitions from (P_1, E_1) to (P_1, E_4) (as show in solid arrow in this figure), omitting other diverse transitions such as (P_2, E_2), (P_4, E_4), etc,. Moreover, the logics of condition E_5 and E_6 in SEDP are also unreachable. State-of-the-art fuzzers are challenged that how to direct fuzzing to cover the abundant cross-state code coverage in state transitions.

We use the example of Listing 2 to clearly illustrate the difference between common branch coverage-guided fuzzing and the cross-state fuzzing strategy we proposed. This example presents a function used to parse an SEDP packet and then construct the EDPEndpoint in the SEDP state of RTPS protocol. The first code fragment from line 3 to line 5 is an if-judgement constraint and its variable is 'm_simpleEDP.usePubWriterANDSubReader', a variable that has no relation with the former SPDP state and only relies on the configuration specified in SEDP packet. Branch coverage-

guided fuzzing is suitable for covering the logics in this fragment by conducting mutations on SEDP packet. However, the second and third code fragments at lines 7-14 and lines 16-18 are totally different. The second fragment is a switch-judgement and the third one is an if-judgement, but they all take variables of properties in the former SPDP state. These fragments are clearly related to state transitions. As a result, for common fuzzers, the value of these variables tend to be limited, thus these state transition-related code logics are not covered enough during fuzzing.

```
1  void createEDPEndpoint(CDRMessage *EDPMessage,
       PDPParticipant mp_PDP, /*...*/){
2    //... Construct m_simpleEDP from EDPMessage ...
3    if(m_simpleEDP.usePubWriterANDSubReader==true){
4      // ...
5    }else{/*...*/}
6    // ...
7    switch(mp_PDP->getRTPSParticipantAttributes().
       throughputController){
8      case PDPControl::MaxBytesPerPeriod: //...
9      case PDPControl::NormalBytesPerPeriod: //...
10       if(m_simpleEDP.readAttr.kind == KEY){/**/}
11       else {/*...*/}
12     case PDPControl::BytesPerPeriodNotControl: //
13     // ...
14   }
15   // ...
16   if(mp_PDP->getRTPSParticipantAttributes().
       allocation.initial > 1){
17     // set attributes of m_simpleEDP
18   }else{/*...*/}
19   // ...
20 }
```

Listing 2. Code snippet from RTPS Protocol

(2) Maximize the throughput for execution of ICS protocols. During practice, we find that state-of-the-art fuzzers such as AFLNET [9] are not suitable for the continuous running scenario of ICS protocols, and the restarting of target protocols in each iteration is time consuming.

In order to use feedback for optimization and guidance, state-of-the-art fuzzers commonly collect code coverage of the under-test program. Existing fuzzers collect such information through mainly two methods. The first method is to instrument the program to insert a flag for coverage collection, such as AFL's persistent mode and Libfuzzer. However, this is not suitable for ICS protocols because they usually run as background multi-processes thus no distinct flags are provided at code level after they finish processing packets. The other method is to collect coverage after the program under-test has exited, such as AFL. This is more versatile as the fuzzer can kill the processes of the program, collect its coverage and then restart the program for the next iteration. However, restarting the target ICS protocol is sub-optimal considering the complexity of bringing up such a system, thus periodic restarts significantly affect fuzzing efficiency.

IV. System Design

We present the framework of Charon in Figure 5. Charon leverages the module of Peach Pits file introduced in Section II-B, to describe state model and data models. To realize the cross-state continuous fuzzing of ICS protocols, we design two innovate modules. One is the cross-state guiding module which maximizes coverage in state transitions, and the other is the program status inferring module that keeps the continuous scenario of the target ICS protocol.

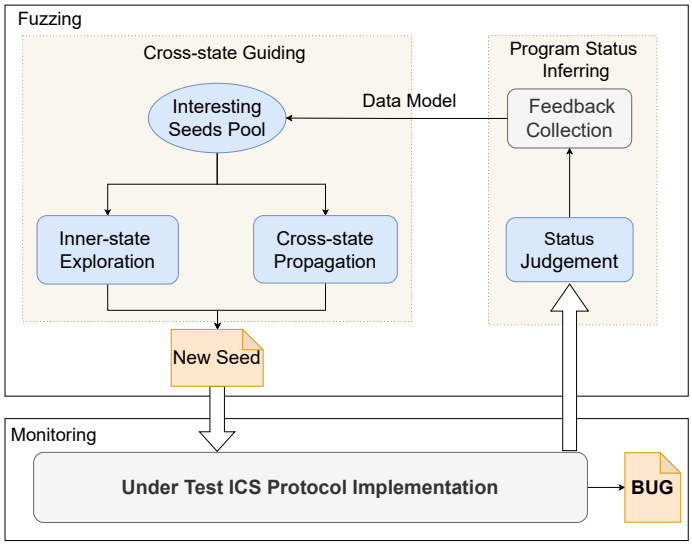


Fig. 5. Overview of Charon. The cross-state guiding module helps to maximize code coverage in state transitions, and the program status inferring module keeps the continuous running scenario of the target ICS protocol.

At the very beginning, Charon uses the generation rule of Pits file to generate a seed $\mathcal{S}$ according to the DataModel and outputs $\mathcal{S}$ to the endpoint, which is previously injected with lightweight instrumentation to acquire coverage information. Then Charon utilizes the program status inferring module to judge whether the subscriber has finished processing $\mathcal{S}$. If the module judges that $\mathcal{S}$ has been fully consumed, Charon then collects code coverage information of $\mathcal{S}$. Next, Charon compares the newly retrieved coverage to its previous coverage set (initially empty) to conclude whether $\mathcal{S}$ achieves new coverage. If so, $\mathcal{S}$ it is added to the interesting seeds pool.

Charon adopts an innovative cross-state guiding module that uses state guidance to maximize cross-state code coverage during state transitions. For each state, the inner-state exploration submodule in Charon constantly evolves the corresponding protocol packet to explore as many conditions as possible. In Charon, a new condition in a state corresponds to a new interesting packet generated or mutated from the data model of this state, as Charon uses packet-level feedback and one packet belongs to exactly one state. Once a new condition in a prior state is found by a produced protocol packet, the cross-state propagation submodule propagates this new prior condition to its following state. New sequences of packets will be generated to attempt to cover the code in state transition between these two states. Moreover, this new condition in the prior state will help to explore more available new legal conditions in its following state.

The newly derived ICS protocol packet of the cross-state guiding module is then injected to the under-test ICS protocol endpoint. Afterwards, Charon returns to program status inferring module and continues its fuzzing loop. As the program status inferring module is able to infer the time point at which

the endpoint has finished processing the packet, the target ICS protocol does not need to repeatedly restart to collect feedback. Therefore, during the whole fuzzing process, it only restarts when it crashes due to a potential bug, which allows for continuous running thus improves the fuzzing efficiency.

A. Cross-State Guiding

Inner-state Exploration Submodule. `Charon` utilizes the inner-state exploration submodule to explore as many conditions as possible in each state of the under-test ICS protocol endpoint. Those interesting seeds that uncover new conditions are saved in the interesting seeds pool. During inner-state exploration, we use *branch* coverage to detect any new conditions in each state and inject lightweight instrumentation at branch points of the under-test ICS protocol to obtain it. Therefore, to exploring state $\mathcal{A}$, we can use interesting ICS protocol packets which correspond to state $\mathcal{A}$ to speed up the exploration process. This is based on the evolutionary algorithms adopted by many mutation-based fuzzers. However, existing mutation-based fuzzers conduct bit-level mutations which are likely to destroy the integrity of ICS protocol packets, while `Charon` is aware of their detailed structure so we perform structure-aware mutations on elements.

```
1  void foo(int x) {
2      __instrumentation();                    // A
3    if (x) {
4          __instrumentation();                // B
5      do_stuff(x);
6    } else {
7          __instrumentation();                // C
8      do_other_stuff(x);
9    }
10 }
```

Fig. 6. Example code snippet abstracted from instrumented program.

To identify interesting conditions in each fuzzing state, as Figure 6 shows, we invoke instrumentation function `__instrumentation()` (lines 2, 4, 7) at the branch point of each basic block. For example, the symbol $\mathcal{B}$ represents the basic block from line 3 to 6, where the instrumentation is injected at line 4. When the instrumentation is executed, a branch ID is calculated according to both the previous block and the current block. If a generated valid protocol packet covers any new branches or visiting some branches with notable changes of times during the parse process, we takes it as interesting packets in this state.

```
<DataModel name="Package_SPDP">
 <Block name="head">
  <String name="Magic_Number" size="32" .../>
  <Block name="Version">
   <Number name="major" size="8" value="02" .../>
   <Number name="minor" size="8" value="02" .../>
  </Block> ...
 </Block>  ...
</DataModel>
```

Listing 3. Example snippet `DataModel` of RTPS protocol

Algorithm 2: Inner-state exploration algorithm

Input: $Pool$: Interesting seeds pool
Input: $State$: State which is current under fuzzing
Output: Set_{seeds}: Set of generated input seeds

1 $Set_{seeds} \leftarrow$ EMPTY()
2 $\mathcal{D} \leftarrow$ GETINNERDATAMODEL($State$)
3 $Type \leftarrow$ GETDATAMODELTYPE($\mathcal{D}$)
4 $List_{IS} \leftarrow$ PICKINTERESTINGSEEDS($Type, Pool$)
5 **for** $seed \in List_{IS}$ **do**
6 $Elements \leftarrow$ SELECTELEMENTS($seed, \mathcal{D}$)
7 **for** $e \in Elements$ **do**
8 $seed \leftarrow$ RUNMUTATION($e, seed$)
9 $Set_{seeds} \leftarrow$ ADD($Set_{seeds}, seed$)
10 **return** Set_{seeds}

We demonstrate how inner-state exploration submodule works in Algorithm 2. We denote $State$ as the current under-test state, the inner-state exploration submodule initially refers to the Pits model to find the mutable `DataModel` $\mathcal{D}$ of this state (line 3). Then it selects a list of interesting seeds corresponding to type $\mathcal{D}$ from the seeds pool (lines 3-4). Afterwards, for each interesting seed in the list, several fields of the seed are randomly chosen to be mutated with its concrete element type in mind, which ensures input integrity (lines 6-8). Listing 3 shows an example snippet `DataModel` of RTPS protocol, each element in the data model has a specific type which determines the mutation strategies. The model also provides other information about elements, such as the size, default value, etc,. After a series of mutations, the exploration submodule produces a list of seeds with higher quality.

Cross-state Propagation Submodule. `Charon` leverages the cross-state propagation submodule to maximize *cross-state code coverage*. Once a new condition $\mathcal{C}_\mathcal{A}$ in state $\mathcal{A}$ is found by a seed $\mathcal{S}_A$, the cross-state propagation submodule propagates this new prior condition to the following state $\mathcal{B}$. Take all discovered valid conditions of $\mathcal{B}$ as $Set_\mathcal{B}$, then for any condition $\mathcal{C}_\mathcal{B} \in Set_\mathcal{B}$, the state transition from $\mathcal{A}$ to $\mathcal{B}$ in the way $(\mathcal{C}_\mathcal{A}, \mathcal{C}_\mathcal{B})$ is considered new. `Charon` then triggers the transition by combining $(\mathcal{S}_\mathcal{A}, \mathcal{S}_\mathcal{B})$, where $\mathcal{S}_\mathcal{B}$ is the corresponding seed for uncovering $\mathcal{C}_\mathcal{B}$. Different conditions in each state represent different configurations and inner data structure values. As a result, we are more likely to execute more code branches when utilizing different transitions of states. In addition, more legal conditions of state $\mathcal{B}$ could be explored due to this new prior condition in state $\mathcal{A}$.

We present an overview of the cross-state propagation submodule in Algorithm 3. Two types of seed sequences are generated due to the cross-state propagation: (i) Seq_α, generated to trigger the transition from $(\mathcal{C}_\mathcal{A}, \mathcal{C}_\mathcal{B})$ by combining $(\mathcal{S}_\mathcal{A}, \mathcal{S}_\mathcal{B})$; (ii) Seq_β, generated to explore more legal conditions of $\mathcal{B}$. When seed $\mathcal{S}_\mathcal{A}$ covers any new conditions in state $\mathcal{A}$, the cross-state propagation submodule first refers to the state model to find the following state $\mathcal{B}$ and extracts the output sequence of state $\mathcal{B}$ as $Seq_\mathcal{B}$ (lines 1-2). To generate Seq_α, this submodule acquires an interesting seed $S_\mathcal{B}$ which

Algorithm 3: Cross-state propagation algorithm

Input: $Pool$: Interesting seeds pool
Input: $\mathcal{A}$: Current state under fuzzing
Input: $S_{\mathcal{A}}$: Seed covering new condition in state $\mathcal{A}$
Output: Seq_α: Sequences of generated packets (in type α)
Output: Seq_β: Sequences of generated packets (in type β)

1 $\mathcal{B} \leftarrow$ GETNEXTSTATE($\mathcal{A}$)
2 $Seq_\mathcal{B} \leftarrow$ EXTRACTDATAMODELSEQ($\mathcal{B}$)
3 $\mathcal{D}_\mathcal{A}, \mathcal{D}_\mathcal{B} \leftarrow$ GETINNERDATAMODEL($\mathcal{A}, \mathcal{B}$)
4 $S_\mathcal{B} \leftarrow$ GETINTERESTINGSEED($\mathcal{D}_\mathcal{B}, Pool$)
5 $Seq_\alpha, Seq_\beta \leftarrow$ EMPTY()
6 **for** $\mathcal{D} \in Seq_\mathcal{B}$ **do**
7 **if** $\mathcal{D} = \mathcal{D}_\mathcal{A}$ **then**
8 $Seq_\alpha, Seq_\beta \leftarrow$ APPEND($Seq_\alpha, Seq_\beta, S_\mathcal{A}$)
9 **else**
10 **if** $\mathcal{D} = \mathcal{D}_\mathcal{B}$ **then**
11 $Seq_\alpha \leftarrow$ APPEND($Seq_\alpha, S_\mathcal{B}$)
12 $Seeds \leftarrow$ INNERSTATEEXPLORE($Pool, \mathcal{D}$)
13 $seed \leftarrow$ CHOOSEONE($Seeds$)
14 $Seq_\beta \leftarrow$ APPEND($Seq_\beta, seed$)
15 **else**
16 $seed \leftarrow$ GENERATEBYDEFAULT($\mathcal{D}$)
17 $Seq_\alpha, Seq_\beta \leftarrow$ APPEND($Seq_\alpha, Seq_\beta, seed$)
18 **return** Seq_α, Seq_β

has covered valid condition in state $\mathcal{B}$ (line 4) from the pool. Then we loop over traversing the sequence of data models. If the loop reaches the mutable data model in $\mathcal{A}$ while immutable in $\mathcal{B}$, both Seq_α and Seq_β take $\mathcal{S}_\mathcal{A}$ as the output seed (lines 7-8). Then if it reaches the mutable data model in state $\mathcal{B}$, $S_\mathcal{B}$ is adopted as the seed to Seq_α while a new seed generated in the inner-state exploration submodule is adopted to explore more conditions in state $\mathcal{B}$ (lines 10-14). Other data models in $Seq_\mathcal{B}$ are generated as the seeds with default value for both Seq_α and Seq_β.

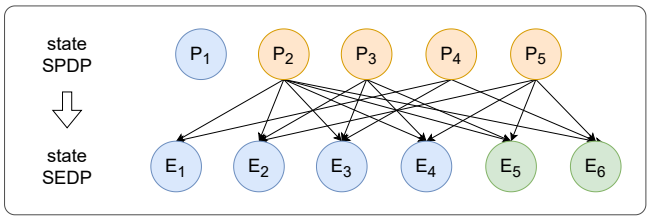


Fig. 7. State transitions Charon additionally covers.

Taking the example of RTPS protocol, we use Figure 7 to present the influence of cross-state guiding strategy in Charon. Compared with Figure 4 in Section III, Charon is able to cover the diverse state transitions between SPDP and SEDP such as (P_2, E_1), (P_2, E_4), (P_4, E_2), etc, with the help of Seq_α. Moreover, more legal conditions of state SEDP, like E_5 and E_6, are covered due to Seq_β.

B. Program Status Inferring

To establish effective continuous fuzzing and cross-state guiding, it is critical to evaluate the improvement of code coverage achieved by each generated ICS protocol packet in each state. Thereafter, the fuzzer should know when the protocol finishes processing each packet $\mathcal{S}$ for each execution so that the coverage feedback of $\mathcal{S}$ can be retrieved for the interesting seed measurement.

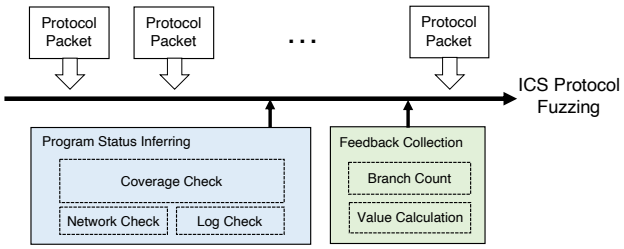


Fig. 8. How program status inferring module works during fuzzing.

Figure 8 clearly presents the role that program status inferring module plays during the whole ICS protocol fuzzing procedure. Between sending each generated protocol packet, Charon invokes this module to monitor the packet parsing process. Three checking strategies are used for interring, where the main one is runtime coverage check. And the other two assistant strategies are respectively network check and log check. When all the three checking strategies reach the conclusion that the packet has been fully consumed, Charon moves on to feedback collection of this packet and then the generation of next ICS protocol packet.

Algorithm 4: Program status inferring algorithm

Input: $StateModel$: State model of the target protocol
1 **for** $state \in StateModel$ **do**
2 $\mathcal{S} \leftarrow$ CROSSSTATEGUIDING($state, StateModel$)
3 FEEDTOICSPROTOCOL($\mathcal{S}$)
4 $Cov_\alpha \leftarrow$ EMPTY()
5 **while** $True$ **do**
6 $Cov_\beta \leftarrow$ COLLECTCOVERAGE()
7 **if** $Cov_\beta = Cov_\alpha$ && NOPACSEND() && NOLOGADD() **then**
8 $break$
9 **else**
10 $Cov_\alpha \leftarrow Cov_\beta$
11 WAITINTERVAL()
12 PERFORMFEEDBACK($\mathcal{S}, state, Cov_\alpha$)
13 MOVETONEXTSTATE($state, StateModel$)

We follow the procedure of Algorithm 4 to conduct status inferring of ICS protocols. After feeding packet $\mathcal{S}$ to under-test ICS protocol endpoint (line 6), we reuse the coverage track component, as the content in shared memory is able to extract the program execution flow exercised by $\mathcal{S}$. By conducting consecutive coverage collections with a specified *interval*, the

execution flow exercised by $\mathcal{S}$ in the *interval* can be revealed in the coverage change of adjacent results. Assuming that two sequential results in the collection sequence are represented as Cov_α and Cov_β respectively, Cov_β can be considered as the final coverage result of $\mathcal{S}$ only if Cov_β doesn't trigger any new coverage compared with Cov_α. Moreover, during the interval, if $\mathcal{S}$ is fully consumed, the ICS endpoint should not generate any more logs related to state change. Additionally, no more response packets would be sent from the under-test endpoint when it becomes relatively stable after consuming $\mathcal{S}$ (line 7). This approach is lightweight and effective, with negligible overhead instead of sacrificing time to restart the whole system of ICS protocol.

C. System Implementation

`Charon` mainly consists of two innovative modules: the cross-state guiding module to maximize code coverage in state transitions, and the program status inferring module to keep the continuous running scenario of ICS protocols. We implemented the kernel modules of `Charon` mainly in C# (6500+ lines of code in total), and also utilized several underlying tools. We leverage the structure of Pits file from Peach (community version 3.0.202) to describe state model of fuzzing and data models corresponding to states. During continuous cross-state guided fuzzing, to measure branch coverage of under-test ICS protocol implementation, we apply light-weight instrumentation technique which is based on the LLVM Clang compiler (version 12.0.0) [17].

V. Evaluation

We performed repeated experiments to evaluate the performance of `Charon`. To investigate whether the continuous fuzzing strategy with cross-state guidance improves the effectiveness and efficiency of fuzzing, we compared `Charon` with five state-of-the-art fuzzers in Section V-B, including one general fuzzer (AFL [5]) and four fuzzers mainly designed and optimized for protocol (Polar [10], AFLNET [9], Boofuzz [8], and Peach [6]). The results of these experiments demonstrate the improvement in fuzzing effectiveness. Finally, we show the vulnerability detection capabilities of `Charon` and list the previously unknown vulnerabilities exposed by `Charon` in Section V-C. Specifically, we evaluate `Charon` to answer the following research questions:

RQ.1 Is `Charon` more effective than state-of-the-art fuzzers when augmented with the proposed cross-state continuous fuzzing strategies?

RQ.2 Can `Charon` expose more previously unknown vulnerabilities than those state-of-the-art fuzzers?

A. Experiment Setup

We evaluated the performance of `Charon` on several open-source implementations of ICS protocol implementations, including `FastRTPS` [18], `CycloneDDS` [19], `FreeRTPS` [20], `OpenDDS` [21], `IEC61850-MMS`[22], and `MQTT`[23]. These real-world projects are widely used, and they come with varying degrees of complexity. Table I shows the full list of projects and their information. Note that their size is calculated in thousands lines of code.

TABLE I
Description of RTPS Implementations

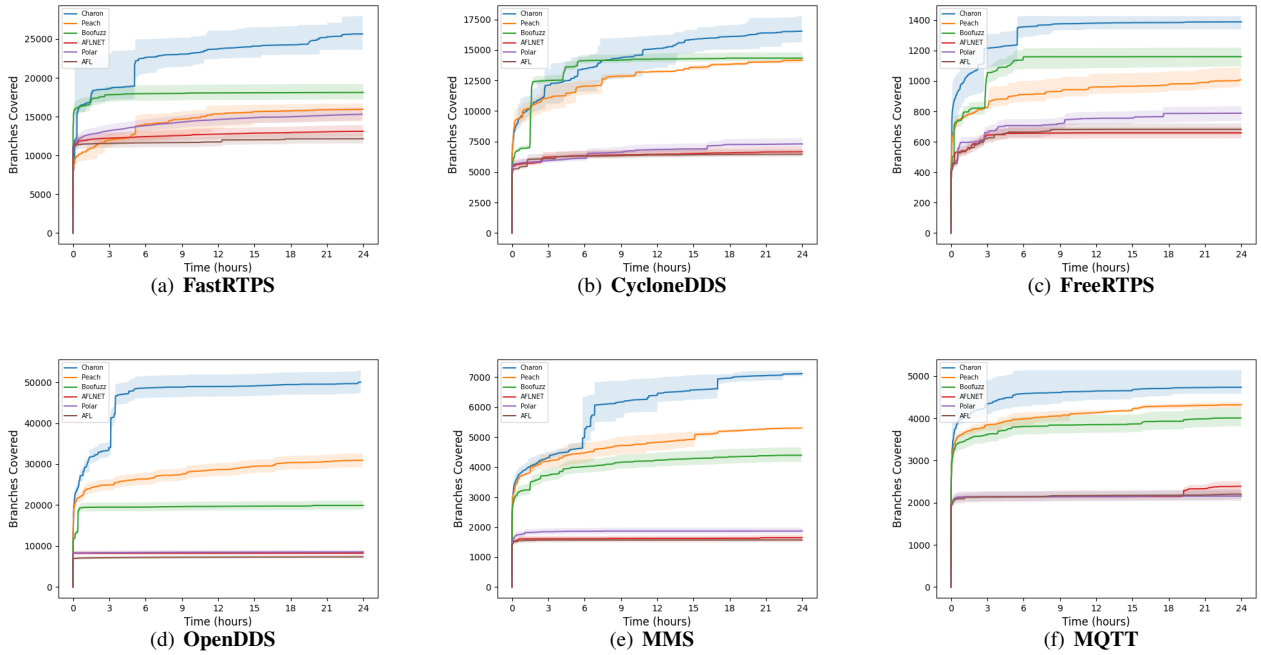
Implementation	Size	Brief Description
FastRTPS	599K	Adopted in many sectors, e.g. Robotic Operating System (ROS) [24], FIWARE Incubated GE.
CycloneDDS	85K	A part of Eclipse IoT project widespreadly used in IoT industry, also a tier-1 middleware for ROS2.
FreeRTPS	36K	A free, portable, minimalist, work-in-progress RTPS implementation in ROS software stack.
OpenDDS	2206K	Developed by Object Computing, one of the most famous DDS and RTPS implementation.
IEC61850-MMS	112K	A standard communication protocol used in the electric industry. It's based on COTP and so on.
MQTT	51K	A lightweight IoT network protocol based on TCP/IP for embedded asynchronous communication.

We used branch coverage and the number of unique bugs detected as metrics. The first metric is commonly used to measure the effectiveness of fuzzers while the second indicates the ability to detect vulnerabilities. Our experiments are conducted on a host machine running Ubuntu 20.04 with 128GiB of memory and two Intel ® Xeon ® Gold 6148 CPUs @ 2.40GHz with 80 logical cores. We ran each fuzzing tool on each project for 24 hours in independent one-core KVM environments to isolate the resources of both network and operation system, and repeated each experiment 5 times to establish statistical significance of results.

`Charon` and Peach started with the same input data models while AFL, Polar and AFLNET was initiated with same binary streams of these data models as seeds. Specifically, for AFLNET, as it requires state codes from protocol packets to distinguish different states while no such codes exist in our selected protocols, so we refer to the solution of the official repository of AFLNET [9] and use the hash value of respond packet to identify state code in this tool. Moreover, Boofuzz was seeded with the same structure information of under-test ICS protocols with `Charon` and Peach. To precisely present the influence of input packets generated by fuzzers, we did not count the code branches related to the initialization of ICS protocols. In addition, `Charon`, Peach, and Boofuzz all operate in non-restart mode during our experiments. We use the non-restart mode of Peach and Boofuzz to show the best performance of them when compared to out tool `Charon`.

B. Efficiency of Coverage Improvement

Figure 9 shows the average performance of each fuzzing tool on different ICS protocol implementations over 24 hours. Each curve in this figure is the average result taken over 5 runs. We collected and summarized the results of average covered branches and calculated the enhancement of `Charon` compared with other state-of-the-art fuzzers into Table II. The cells in first line of each ICS protocol implementation in this table represent the average code branches each fuzzer achieved respectively while those in the next line correspond to the percentage of the increase of code branches `Charon` achieved compared with each fuzzer. At the bottom of the table, we calculate the average improvement `Charon` achieved among all these implementations compared with each fuzzer.



(a) **FastRTPS** (b) **CycloneDDS** (c) **FreeRTPS**

(d) **OpenDDS** (e) **MMS** (f) **MQTT**

Fig. 9. Average results of the covered branches of each fuzzing tool on different ICS protocol implementations within 24 hours.

TABLE II
AVERAGE NUMBER OF CODE BRANCHES ACHIEVED BY EACH FUZZING TOOL IN 24 HOURS.

Subject	AFL	Polar	AFLNet	Boofuzz	Peach	`Charon`
FastRTPS	12147	15310	13100	18114	15952	**25656**
	+111.2%	+67.58%	+95.85%	+41.64%	+60.83%	-
Cyclone	6250	7066	6444	14091	14158	**16539**
	+164.6%	+134.1%	+156.7%	+17.37%	+16.82%	-
FreeRTPS	682	788	659	1160	1008	**1389**
	+103.7%	+76.27%	+110.8%	+19.74%	+37.80%	-
OpenDDS	7360	8559	8270	19901	34431	**49998**
	+579.3%	+484.2%	+504.6%	+151.2%	+45.21%	-
MMS	1682	1886	1691	4328	5152	**7256**
	+331.4%	+284.7%	+329.1%	+67.65%	+40.84%	-
MQTT	2202	2157	2389	4019	4321	**4736**
	+115.1%	+119.6%	+98.24%	+17.84%	+9.60%	-
AVG	+234.2%	+194.4%	+215.9%	+52.58%	+35.18%	-

From Figure 9 we can infer that at the beginning of fuzzing, all of these fuzzers achieved new branches quickly. However, after a short period of time (about 1-2 hours), the baseline fuzzers (AFL, Polar, AFLNET, Boofuzz, and Peach) slowed down and gradually reached a state where they rarely explored new branches.

For AFL and it's variants, they are unaware of the structure of ICS protocol packets and therefore they can only consider them as raw binary streams and randomly flip, copy, add or delete some bits to derive new packets. These blind modifications are likely to destroy the structure of packets and thus the malformed output packets tend to fail sanity checks and cannot reach the actual program logic. More significantly, they all restart under-test ICS protocol after each iteration, which greatly slows down the fuzzing procedure. As a result, though they have coverage feedback to help evolve seeds, their performance was not such satisfactory. For Peach and Boofuzz, despite the awareness of ICS protocol packet structure, they can only generate syntactically valid seeds as it adopts a black-box fuzzing strategy and no feedback can guide it. Moreover, they only focus on exploring coverage in isolated states as introduced in Section III. Therefore, they outperformed AFL, Polar and AFLNET but there still remained a big gap with `Charon` in performance. `Charon`'s branch coverage continuously improves for a longer period of time due to the packet-level feedback in inner-state exploration and the effective cross-state propagation fuzzing strategy.

As shown in Table II, `Charon` achieved more code branches than AFL, Polar, AFLNET, Boofuzz, and Peach on all of the selected ICS protocol implementations. Specifically, its branch coverage improved 234.2% over AFL on average, proving both the importance of input structure awareness and the continuous fuzzing strategy. In particular, in OpenDDS, the most complex among these implementations we chose, AFL only achieved less than 10 thousand branches while `Charon` achieved nearly 50 thousand. We see similar patterns among the results of the other ICS protocol implementations. As to AFL's variants Polar and AFLNET, though new strategies and optimizations are introduced, the improvements to AFL were not satisfactory, and `Charon` outperforms them. Compared with Boofuzz and Peach, `Charon` improved 52.58% and 35.18% on average with the novel cross-state guiding strategy which attempts to cover as more cross-state processing logic

as possible to improve the code coverage (Boofuzz and Peach were set to run in their non-restart mode to ensure their best performance and guarantee the fairness of the conducted contrast experiments).

TABLE III
COMPARISON BETWEEN CHARON AND CHARON$^-$

Tool	FastRTPS	Cyclone	FreeRTPS	OpenDDS	MMS	MQTT
Charon	25656	16539	1389	49998	7256	4736
Charon$^-$	23327	15678	1321	46262	6419	4502
	-9.08%	-5.21%	-4.90%	-7.47%	-11.54%	-4.94%
AVG	-7.19%					

In order to evaluate the effectiveness of specific components of Charon, we designed an additional contrast experiment between Charon and its cut version without program status inferring module (we name it Charon$^-$). This additional experiment was also repeated 5 times over the above-mentioned six selected ICS protocol implementations for 24 hours and it also uses the number of program code branches as the evaluation metric. We present the result of this experiment in Table III where the average branches achieved by Charon and Charon$^-$ on each ICS protocol are filled in the cells. Without program status inferring module, Charon has to perform with frequent restarts to collect branch coverage corresponding to each generated ICS protocol packet and then conduct feedback. Table III shows that among all these six protocol implementations, Charon$^-$ achieved fewer branches compared with Charon in 24 hours (ranging from 4.90% on FreeRTPS to 11.54% on IEC61850-MMS). The overall reduction percentage is 7.19%. The results prove the program status inferring strategy that avoids frequent restart during ICS protocol fuzzing effectively improves overall performance within the limited fuzzing time budget.

Answer to RQ1. It is reasonable to conclude that the cross-state fuzzing guidance and continuous fuzzing strategy for ICS protocols are valuable and effective in practice. It is able to improve the branch coverage in protocol implementations within a limited time budget.

C. Previously Unknown Vulnerabilities

The amount of code coverage affects the potential for a fuzzer to detect vulnerabilities. A fuzzer can only detect any vulnerbilities when the code segment it resides in is executed. Due to the great performance Charon achieved in code coverage as presented in Section V-B, in practice, Charon was able to expose some previously unknown vulnerabilities in all four of the under-test ICS protocol implementations, greatly exceeding our expectation. We summarized these new vulnerabilities in Table IV and also presented the results of the other state-of-the-art fuzzers.

All the five tools performed fuzzing accompanied with AddressSanitizer for vulnerability detection. As shown in Table IV, Charon has exposed 21 previously unknown vulnerabilities in total and most of them are buffer overflows, which are known to be easily exploitable to perform attacks such as remote code execution (RCE), denial of service (DoS), etc. In practice, we found that AFL, Polar and AFLNET can only find 2, 3, 3 of these vulnerabilities while Boofuzz and Peach only found 11 and 13. All the other fuzzers could not find extra new vulnerabilities out of these 21 vulnerabilities exposed by Charon. From the result, we can see that Charon is effective in exposing previous unknown vulnerabilities.

TABLE IV
PREVIOUSLY UNKNOWN VULNERABILITIES **EXPOSED BY CHARON** IN UNDER-TEST ICS PROTOCOL IMPLEMENTATIONS.

Subject	Vulnerability Type	AFL	Polar	AFLNet	Boofuzz	Peach	**Charon**
FastRTPS	heap-buffer-overflow	0	0	0	0	0	**1**
	stack-buffer-overflow	0	0	0	1	1	**2**
Cyclone	heap-buffer-overflow	0	0	0	0	**1**	**1**
	stack-buffer-overflow	0	0	0	**1**	**1**	**1**
FreeRTPS	stack-use-after-scope	0	0	0	**1**	**1**	**1**
	global-buffer-overflow	2	3	2	7	7	**10**
OpenDDS	heap-buffer-overflow	0	0	1	1	1	**3**
MMS	SEGV	0	0	0	0	1	**2**
Total		2/21	3/21	3/21	11/21	13/21	**21/21**

```
177    dest[i] = msg->buffer[msg->pos + i];

@@ -230,8 +230,9 @@ inline SequenceNumberSet_t
     CDRMessage::readSequenceNumberSet
228    uint32_t numBits = 0;
229    valid &= CDRMessage::readUInt32(msg, &numBits);
     + valid &= (numBits <= 256u);
230    uint32_t n_longs = (numBits + 31ul) / 32ul;
231    uint32_t bitmap[8];
232  - for (uint32_t i = 0; i < n_longs; ++i)
     + for (uint32_t i = 0; valid && (i < n_longs);
         ++i)
233    {
234       valid &= CDRMessage::readUInt32(msg,
              &bitmap[i]);
235    }
```

Listing 4. Code snippet of a stack buffer overflow vulnerability found by Charon in FastRTPS and the corresponding patch from the vendor

Listing 4 illustrates a stack-buffer-overflow vulnerability found by Charon in FastRTPS. The vulnerability occurred in function ReadUInt32 at Line 177. It was caused by an illegal WRITE operation to dest[i]. We moved back to its caller and found the true reason: the numBits read from the package in line 237 was bigger than 256, so n_longs calculated in next line is greater than 8. However, in line 239, the array bitmap had a length of only 8 bytes. Thereafter, after entering the loop which started at line 240, the address bitmap[9] was passed to function ReadUInt32 and wrote in the reference dest[i] at Line 177. Therefore, a stack-buffer-overflow vulnerability was triggers for attempting to write to a non-allocated stack address. We reported the vulnerability to the developers and they have released the corresponding patch that is shown in the form of diff in Listing 4. The vendors added a condition statement to decide whether the variable numBits is valid and only when it is

valid, could the program execute the loop in line 232, thus fixing this vulnerability issue.

These six selected protocols are practical protocols used in industry environment and all the 21 vulnerabilities found by Charon are 0-day bugs that haven't been reported before. We had removed the duplicates and they are all unique. Specially, the two vulnerabilities we found in CycloneDDS have been assigned with CVE numbers CVE-2020-18734 and CVE-2020-18735 (submitted by us to MITRE Corporation on 08/13/2020 and we got the reply on 08/24/2021, the processes take a long time). And these two vulnerabilities are all rated as level HIGH and get the 7.5 CVSS score from the NVD. We have also reported the others of remained vulnerabilities on 04/16/2021 and we are waiting for replies. Moreover, as Table IV shows, 18 out of the 21 0-day vulnerabilities we found are related to Buffer Overflow, a vulnerability category that is extremely dangerous for ICS systems. As the OWASP Foundation describes[25], the Buffer Overflow vulnerabilities are rated 'VERY HIGH' in the metrics of severity. As to the metrics of likelihood of exploit, OWASP thinks the result is 'HIGH to VERY HIGH' for Buffer Overflow.

Answer to RQ2. The results in Table IV prove that Charon is effective in finding bugs of real-world ICS protocols. Moreover, Charon has the best performance of vulnerability detection among all selected fuzzers. It exposed 21 previous-unknown severe vulnerabilities in ICS protocol implementations and all of them have been confirmed and fixed by the vendors.

VI. Discussion on Scalability

The main contribution of Charon is to employ cross-state guidance and program status inferring to maximize code coverage and support continuous fuzzing. Hence, it is more likely to reach deeper and critical program statements before exposing ICS protocol bugs.

The main potential threat is about the scalability to other protocols. The method presented in this paper can be also applied to other multi-states ICS protocols. The fuzzing procedure of Charon is fully automatic and the continuous state-guiding fuzzing strategy is scalable across different protocols. To adapt for other protocols, one only needs to provide the state model of the target ICS protocol which stipulates what are the under-test states and how these states jump from one to another (as shown in Listing 1).

Another potential threat is to prepare high-quality data models for fuzzing ICS protocols. The accuracy of data models is crucial to guarantee the effectiveness of fuzzing. Without data models, fuzzers such as AFLNet and Polar are unaware of the ICS protocol packet structure, thus are limited to performing input structure agnostic mutations to produce new input packets. These modifications are likely to invalidate the input structure, therefore, the generated inputs tend to fail sanity checks early on and are unable to reach actual program logic. However, to build data models for generation-based fuzzers such as Peach and Charon, developers need to read through both the official protocol specifications and the documentation of the target implementation, as ICS protocols allow vendor-specific customization. Currently, we use existing data models and will try to automatic data model learning in future work.

VII. Related Work

Generic Fuzz Testing. Fuzz testing has become one of the most famous techniques in the field of software testing. It enjoys great popularity for its fully automatic running manner and promising performance. Specifically, grey-box fuzzing, which collects coverage feedback to guide generation or mutation, has proved to be effective by many practices in real-world software. AFL [5] is the representative of grey-box fuzzers. To improve the efficiency of AFL, some researchers modified its seed selection or mutation strategies, such as FairFuzz [26]. Some other researchers equipped grey-box fuzzing with taint analysis technique to analyze the trace of input streams and better guide the fuzzing procedure, such as PATA[27], DeepFuzzer [28] and REDQUEEN [29]. What's more, symbolic execution has been adopted by some researchers to help fuzzers reach the deep logic of target programs. One example of them is MoWF [30], which leverages input formats as the constraint of symbolic execution during path exploration.

Fuzzing common network protocols. Many fuzzing tools have been implemented and proposed for testing diverse network protocols [9],[31],[32],[33]. AFLNET [9] extends AFL and leverages state feedback to explore states as much as possible. Despite the ability of exploring states, it focuses on improving code coverage in isolated states, i.e. it does not enhance cross-states code coverage. Moreover, AFLNET is unable to utilize packet-level feedback, which is important to figure out the contribution of each packet in the generated sequence. More significantly, AFLNET relies on state code to distinguish between different states which is not adopted in most ICS protocols, so the applicability and effectiveness of AFLNET are both limited. AutoFuzz [32] and Protocol-state-fuzzing [31] share a similar insight with AFLNET which automatically learns the state machine of a protocol with state code to explore as many states as possible. In contrast with these fuzzers, Charon not only takes protocol state into account, but also places emphasis on maximizing cross-state code coverage on multi-state ICS protocols by leveraging packet-level feedback. Moreover, Charon conducts optimization upon the running scenario of fuzzing ICS protocols which is able to perform feedback-driven fuzzing without restarting the target ICS protocol frequently.

Testing of ICS protocol implementations. Some works try to apply fuzzing to ICS protocols[10][34][35][36][37]. Polar [10] leverages taint analysis to extract opcode of ICS protocol to optimize the fuzzing process. GANFuzz[34] uses generative adversarial network to train models for ICS protocols while fuzzing. PropFuzz[35] conducts fuzzing on PLC communication with the ICS environment. Beside fuzz testing, there are many other kinds of works focusing on testing ICS protocols[38][39][40][41]. *Jaime,et al* proposed a validation approach to test ICS protocol vulnerabilities[38] while *Riyadi,et al* performed real-time testing of ICS[39].

The work of ICS test bed[40] provides a convenient way to generate test cases of ICS. In terms of the example RTPS protocol in this paper, JFIT [42] leverages the fault injection technique to test the robustness of some RTPS systems. *Thomas et al.* conducted an investigation into some security issues in RTPS protocol [43]. Different from existing testing works of ICS protocols, Charon focuses on combining input-structure-aware fuzzing technique with cross-state transitions within such a multi-state environment of ICS protocols and keeping their incessant running scenario during fuzzing.

VIII. Conclusion

In this paper, we present Charon, an efficient continuous fuzzing platform for ICS protocol vulnerability detection. It employs an innovative solution that uses state guidance to maximize cross-state code coverage. Based on a novel packet-level feedback mechanism and program status inferring technique, Charon is able to perform continuous executions of ICS protocols during fuzzing. Finally, we evaluate Charon on six widely-used ICS protocol implementations: FastRTPS, CycloneDDS, FreeRTPS, OpenDDS, IEC61850-MMS and MQTT. It achieved significant branch coverage increases over state-of-the-art fuzzers such as Polar, AFLNET, Boofuzz, and Peach. In addition, Charon has exposed 21 previously unknown security critical vulnerabilities. We will enhance Charon with automatic data model learning and apply it for more ICS protocols in our future work.

References

[1] Wikipedia, "Industrial control system," Website, Accessed Mar. 26th, 2022, https://en.wikipedia.org/wiki/Industrial_control_system.

[2] W. Knowles, D. Prince, D. Hutchison, J. F. P. Disso, and K. Jones, "A survey of cyber security management in industrial control systems," *International journal of critical infrastructure protection*, vol. 9, pp. 52–80, 2015.

[3] R. Langner, "Stuxnet: Dissecting a cyberwarfare weapon," *IEEE Security & Privacy*, vol. 9, no. 3, pp. 49–51, 2011.

[4] M. Kumar, "Dragonfly 2.0: Hacking group infiltrated european and us power facilities," *The Hacker News*, 2017.

[5] M. Zalewski, "American fuzzy lop," 2015.

[6] Tool, "Peach fuzzing platform." Website, Accessed Mar. 26th, 2022, https://gitlab.com/gitlab-org/security-products/protocol-fuzzer-ce.

[7] P. Amini and A. Portnoy., "Sulley," 2012, accessed Mar. 26th, 2022. [Online]. Available: https://github.com/OpenRCE/sulley

[8] Boofuzz, "Boofuzz: Network protocol fuzzing for humans," Website, Accessed Mar. 26th, 2022, https://boofuzz.readthedocs.io/en/stable/.

[9] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Aflnet: A greybox fuzzer for network protocols," in *IEEE International Conference on Software Testing, Verification and Validation (Testing Tools Track)*, 2020.

[10] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, "Polar: Function code aware fuzz testing of ics protocol," in *The ACM SIGBED International Conference on Embedded Software (EMSOFT)*, 2019, pp. 1–22.

[11] OMG, "Dds interoperability wire protocol," Website, Accessed Mar. 26th, 2022, https://www.omg.org/spec/DDSI-RTPS.

[12] "Official website of omg," Website, Accessed Mar. 26th, 2022, https://www.omg.org/.

[13] O. M. Group., "Data distribution service," Website, Accessed Mar. 26th, 2022, https://www.omg.org/spec/DDS/1.4/.

[14] Autoware, "Autoware," Website, Accessed Mar. 26th, 2022, https://www.autoware.auto/.

[15] ApolloAuto, "Apollo," Website, Accessed Mar. 26th, 2022, https://github.com/ApolloAuto/apollo.

[16] Autosar, "Adaptive autosar," Website, Accessed Mar. 26th, 2022, https://www.autosar.org/standards/adaptive-platform/.

[17] llvm.org, "Clang: a c language family frontend for llvm." Website, Accessed Jan. 8th, 2021, https://http://clang.llvm.org/.

[18] eProsima, "Fastrtps," Website, Accessed Mar. 26th, 2022, https://github.com/eProsima/Fast-DDS/.

[19] eclipse cyclonedds, "Cyclonedds," Website, Accessed Mar. 26th, 2022, https://github.com/eclipse-cyclonedds/cyclonedds.

[20] objectcomputing, "Freertps," Website, Accessed March. 26th, 2022, https://github.com/objectcomputing/OpenDDS.

[21] F. Covatti, "libiccp," Website, Accessed Mar. 26th, 2022, https://github.com/fcovatti/libiec_iccp_mod.

[22] MzAutomation, "libiec61850," Website, Accessed Mar. 26th, 2022, https://github.com/mz-automation/libiec61850.

[23] Eclipse, "Eclipse mosquitto," Website, Accessed Mar. 26th, 2022, https://github.com/eclipse/mosquitto.

[24] "Robot operating system." Website, Accessed Mar. 26th, 2022, https://en.wikipedia.org/wiki/Robot_Operating_System.

[25] OWASP, "Buffer overflow," Website, Accessed Jan. 8th, 2021, https://owasp.org/www-community/vulnerabilities/Buffer_Overflow.

[26] C. Lemieux and K. Sen, "Fairfuzz: A targeted mutation strategy for increasing greybox fuzz testing coverage," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, 2018, pp. 475–485.

[27] J. Liang, M. Wang, C. Zhou, Z. Wu, Y. Jiang, J. Liu, Z. Liu, and J. Sun, "Pata: Fuzzing with path aware taint analysis," in *2022 2022 IEEE Symposium on Security and Privacy (SP)(SP). IEEE Computer Society, Los Alamitos, CA, USA*, 2022, pp. 154–170.

[28] J. Liang, Y. Jiang, M. Wang, X. Jiao, Y. Chen, H. Song, and K.-K. R. Choo, "Deepfuzzer: Accelerated deep greybox fuzzing," *IEEE Transactions on Dependable and Secure Computing*, vol. 18, no. 6, pp. 2675–2688, 2019.

[29] C. Aschermann, S. Schumilo, T. Blazytko, R. Gawlik, and T. Holz, "Redqueen: Fuzzing with input-to-state correspondence." in *NDSS*, vol. 19, 2019, pp. 1–15.

[30] V.-T. Pham, M. Böhme, and A. Roychoudhury, "Model-based whitebox fuzzing for program binaries," in *31st IEEE/ACM International Conference on Automated Software Engineering*, 2016, pp. 543–553.

[31] J. De Ruiter and E. Poll, "Protocol state fuzzing of {TLS} implementations," in *24th {USENIX} Security Symposium ({USENIX} Security 15)*, 2015, pp. 193–206.

[32] S. Gorbunov and A. Rosenbloom, "Autofuzz: Automated network protocol fuzzing framework," *IJCSNS*, vol. 10, no. 8, p. 239, 2010.

[33] J. Somorovsky, "Systematic fuzzing and testing of tls libraries," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, 2016, pp. 1492–1504.

[34] Z. Hu, J. Shi, Y. Huang, J. Xiong, and X. Bu, "Ganfuzz: a gan-based industrial network protocol fuzzing framework," in *15th ACM International Conference on Computing Frontiers*, 2018, pp. 138–145.

[35] M. Niedermaier, F. Fischer, and A. von Bodisco, "Propfuzz—an it-security fuzzing framework for proprietary ics protocols," in *2017 International conference on applied electronics*. IEEE, 2017.

[36] H. Zhao, Z. Li, H. Wei, J. Shi, and Y. Huang, "Seqfuzzer: An industrial protocol fuzzing framework from a deep learning perspective," in *2019 12th IEEE Conference on software testing, validation and verification (ICST)*. IEEE, 2019, pp. 59–67.

[37] S. J. Kim and T. Shon, "Field classification-based novel fuzzing case generation for ics protocols," *The Journal of Supercomputing*, vol. 74, no. 9, pp. 4434–4450, 2018.

[38] J. Pavesi, T. Villegas, A. Perepechko, E. Aguirre, and L. Galeazzi, "Validation of ics vulnerability related to tcp/ip protocol implementation in allen-bradley compact logix plc controller," in *International Congress of Telematics and Computing*. Springer, 2019, pp. 355–364.

[39] E. Riyadi, T. Priyambodo, and A. Putra, "Real-time testing on improved data transmission security in the industrial control system," in *2020 3rd international seminar on research of information technology and intelligent systems (ISRITI)*. IEEE, 2020, pp. 129–134.

[40] R. E. Gillen, L. A. Anderson, C. Craig, J. Johnson, R. Anderson, A. Craig, and S. L. Scott, "Design and implementation of full-scale industrial control system test bed for assessing cyber-security defenses," in *2020 IEEE 21st International Symposium on" A World of Wireless, Mobile and Multimedia Networks"(WoWMoM)*. IEEE, 2020, pp. 341–346.

[41] D. Duggan, M. Berg, J. Dillinger, and J. Stamp, "Penetration testing of industrial control systems," *Sandia national laboratories*, p. 7, 2005.

[42] A. Napolitano, G. Carrozza, A. Bovenzi, and C. Esposito, "Automatic robustness assessment of dds-compliant middleware," in *2011 IEEE 17th Pacific Rim International Symposium on Dependable Computing*. IEEE, 2011, pp. 302–307.

[43] T. White, M. N. Johnstone, and M. Peacock, "An investigation into some security issues in the dds messaging protocol," 2017.