ICS^3*Fuzzer*: A Framework for Discovering Protocol Implementation Bugs in ICS Supervisory Software by Fuzzing

Dongliang Fang
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
fangdongliang@iie.ac.cn

Zhanwei Song
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
songzhanwei@iie.ac.cn

Le Guan
Department of Computer Science, the University of Georgia
USA
leguan@cs.uga.edu

Puzhuo Liu
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
liupuzhuo@iie.ac.cn

Anni Peng
National Computer Network Intrusion Protection Center, UCAS
Beijing, China
pengan@nipc.org.cn

Kai Cheng
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
chengkai@iie.ac.cn

Yaowen Zheng
Nanyang Technological University
Singapore
yaowen.zheng@ntu.edu.sg

Peng Liu
College of Information Sciences and Technology, Pennsylvania State University
USA
pxl20@psu.edu

Hongsong Zhu
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
zhuhongsong@iie.ac.cn

Limin Sun*
Institute of Information Engineering, CAS; School of Cyber Security, UCAS
Beijing, China
sunlimin@iie.ac.cn

ABSTRACT

The *supervisory software* is widely used in industrial control systems (ICSs) to manage field devices such as PLC controllers. Once compromised, it could be misused to control or manipulate these physical devices maliciously, endangering manufacturing process or even human lives. Therefore, extensive security testing of supervisory software is crucial for the safe operation of ICS. However, fuzzing ICS supervisory software is challenging due to the prevalent use of proprietary protocols. Without the knowledge of the program states and packet formats, it is difficult to enter the deep states for effective fuzzing.

In this work, we present a fuzzing framework to automatically discover implementation bugs residing in the communication protocols between the supervisory software and the field devices. To avoid heavy human efforts in reverse-engineering the proprietary protocols, the proposed approach constructs a state-book based on the readily-available execution trace of the supervisory software and the corresponding inputs. Then, we propose a state selection algorithm to find the protocol states that are more likely to have bugs. Our fuzzer distributes more budget on those interesting states. To quickly reach the interesting states, traditional snapshot-based method does not work since the communication protocols are time sensitive. We address this issue by synchronously managing external events (GUI operations and network traffic) during the fuzzing loop. We have implemented a prototype and used it to fuzz the *supervisory software* of four popular ICS platforms. We have found 13 bugs and received 3 CVEs, 2 are classified as critical (CVSS3.x score CRITICAL 9.8) and affected 40 different products.

CCS CONCEPTS

• **Computer systems organization** → **Embedded and cyber-physical systems**; • **Networks** → *Protocol testing and verification.*

KEYWORDS

Supervisory software, ICS security, protocol implementation bugs, fuzzing, GUI-driven fuzzer

ACM Reference Format:
Dongliang Fang, Zhanwei Song, Le Guan, Puzhuo Liu, Anni Peng, Kai Cheng, Yaowen Zheng, Peng Liu, Hongsong Zhu, and Limin Sun. 2021. *ICS*^{3}*Fuzzer*: A Framework for Discovering Protocol Implementation Bugs

*The corresponding author

ACSAC'21, December 06–10, 2021, Austin, TX, USA

ACM ISBN 978-1-4503-8579-4/21/12…$15.00
https://doi.org/10.1145/3485832.3488028

in ICS Supervisory Software by Fuzzing. In *2021 Annual Computer Security Applications Conference (ACSAC'21), December 06–10, 2021, Austin, TX, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3485832.3488028

1 INTRODUCTION

The *Industrial Internet of Things* (IIoT) is rapidly expanding the inter-connectivity of *industrial control systems* (ICSs). This openness breaks the original assumption that ICSs are isolated systems that run specialized tasks, resulting in a fairly widespread of cyber threats into the industrial control field. This is evidenced by the recent power grid attack in Ukraine [8] and the ransomware attack on TSMC's factory [24].

There are two main categories of targets in an ICS attack – the devices that control manufacturing processes and the corresponding software that manages the devices. Category I targets include devices such as Programmable Logic Controller (PLC), Remote Terminal Unit (RTU), and Programmable Automation Controller (PAC). Each device is usually deployed together with several corresponding management software which is *not* running on the device but on a designated computer. Category II targets include various management software such as Human Machine Interface (HMI), Engineering Software, and Configuration Software. We call this kind of software as ***supervisory software***.

In this paper, we focus on analyzing the insecurity of the supervisory software for PLC devices. More specifically, we focus on identifying the protocol implementation flaws within supervisory software. Supervisory software could be a very attractive target of attackers due to two main reasons. First, once compromised, the supervisory software for a PLC device can conveniently inject malicious program logic into the PLC (through the predefined network interface) without the need to obtain any additional authorizations. For example, the compromised supervisory software may compile the PLC program and perform a malicious update. Second, although the supervisory software for a PLC device is running on a designated, well-protected computer, its interactions with the PLC are not tightly protected. Such interactions are implemented through a TCP session initiated by the supervisory software. Since commercial supervisory software does not enforce strict security checks on whether the PLC it is interacting with is the right one, it is fairly feasible for the attacker to use a "stepping stone" (e.g., a malicious insider) to firstly hijack such a TCP session and then interact with the supervisory software to exploit its protocol implementation flaws. It is found in survey studies [16] that **insider attacks** are top security challenges for air-gapped ICS. Besides insider attacks, the stepping stone could be any internal computer compromised by multi-vector malware attacks (e.g., a variant of Stuxnet) launched from outside.

Fuzz testing is a simple but effective method for finding software bugs [28]. It is widely used by researchers and security analysts to identify security bugs in real-world software. However, there is few research on fuzzing supervisory software in ICSs, and the existing fuzzing tools are still quite limited in serving this new purpose. Specifically, it challenges the existing work in three aspects: (1) Supervisory software generally runs on a Windows System. It is **closed-source** with **bulky size** of the executables (see Section 5.2) and heavy use of GUI. (2) It plays a client role in the communication and harder to test with regular fuzzers. (3) Supervisory software generally uses a proprietary protocol to communicate with a PLC device. Meanwhile, protocol states are highly coupled with GUI operations (see Section 3).

Table 1: Comparison *ICS³Fuzzer* with existing work

Study	closed binary	client-role	GUI-management	Proprietary-protocols?
AFLNet [35]	✗	✗	✗	✗
PripFuzz [32]	✓	✗	✗	✓
Pulsar [15]	✓	✓	✗	✓
Polar [25]	✗	✗	✗	✗
Peach* [26]	✗	✗	✗	✗
Kif [1]	✓	✗	✗	✓
Achilles [41]	✓	✗	✗	✗
ICS³Fuzzer	✓	✓	✓	✓

Prior efforts. To our best knowledge, there is no fuzzing work geared towards the supervisory software. We summarize several mostly related work as follows. Senthivel et al. [36] discuss bug exploitation in the context of supervisory software protocol implementation, but how to discover the bugs is not mentioned. In terms of protocol implementation fuzzing, many approaches [1, 15, 25, 26, 32, 35, 41] strive to generate effective inputs based on protocol formats and states. These studies have at least one of the following limitations: (i) Some works depend on the program source code [25, 26, 35]. (ii) Some works only handle server-role programs [1, 25, 26, 32, 35, 41]. (iii) All works do not involve GUI synchronization management during protocol communication. (iv) Some works that handle proprietary protocols, but are not general enough to be applicable to fuzz supervisory software [1, 15, 32]. Pulsar [15] is similar to our work. However, it cannot be fully automated according to the limitation author claimed. Indeed, Pulsar did not involve in synchronized controls of GUI operations and network communication during the fuzzing loop. The comparison with related work is shown in Table 1.

It can be seen that the coupling of GUI and proprietary protocol implementation brings a major challenge. Conceptually, we could identify and extract the protocol implementation module to write a harness for focused fuzzing. In practice, the process is extremely difficult and buggy. WINNIE [21], a recent work, points out that a group of M.S. students spend 3 days to produce 7 harnesses. Moreover, WINNIE supports writing harness within **ONLY** two components (focuses on file parser programs), but could not be applied to highly-coupled protocol implementations in supervisory software. Besides, generating effective inputs and fuzzing the deep state space of proprietary protocol are also non-trivial.

Our Approach. In this paper, we propose ICS Supervisory Software Fuzzer (*ICS³Fuzzer*), a portable and modular framework which supports automated fuzzing of a variety of supervisory software. We avoid analyzing and extracting the protocol implementation modules to write harness. Instead, we run and fuzz the whole supervisory software, with the synchronized controls of GUI operations and network communication. Except for modest one-time manual effort required for each fuzzing target (see Section 5.5), our fuzzer is fully automated. Because the instrumentation of the bulky sized supervisory software will greatly increase the overhead, *ICS³Fuzzer* is black-box in the fuzzing loop. On this basis,

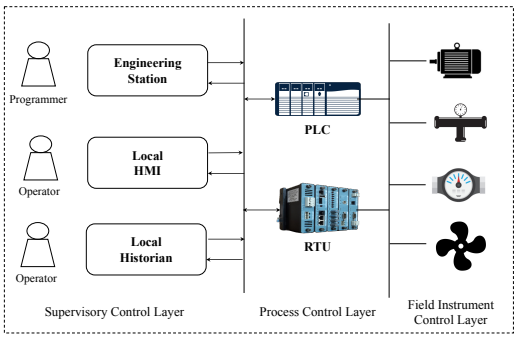

Figure 1: Typical Communication Architecture of ICS

ICS^3*Fuzzer* drives the fuzzing process continuously by automatically reaching different protocol state-space and feeding effective carefully-generated inputs. Besides, to achieve good state coverage and improve the efficiency of *ICS*^{3}*Fuzzer*, we propose a new fuzzing strategy to switch states dynamically. Although it sacrifices performance of fuzzing speed, *ICS*^{3}*Fuzzer* is effective, scalable, and easy to use. Once a bug is found, it is also convenient to reproduce it.

Finally, there are two options to deploy the proposed fuzzing approach: (a) letting a dedicated PLC device conduct the fuzz testing; (b) simulating the response of the PLC device. We tried both options and found that the first option is slow and not scalable. Therefore, we propose a hybrid solution that leverages the strengths of both the real PLC and the simulated responses while avoiding their weaknesses. As a result, we successfully simulated the responses of four different devices and used them to conduct fuzz testing against the corresponding commercial supervisory software, and 13 zero-day bugs were found.

In summary, we make the following contributions:

- We designed and implemented *ICS*^{3}*Fuzzer*, a portable, modular framework to fuzz the supervisory software of ICS. *ICS*^{3}*Fuzzer* is customizable to support the fuzz testing against different supervisory software (with different GUI operations and proprietary protocols) from various vendors.
- We propose a new fuzzing strategy, which selects input states based on execution trace and corresponding inputs.
- In our evaluation, we used *ICS*^{3}*Fuzzer* to test 4 different commercial supervisory software from different vendors. Our experiments discovered 13 zero-day bugs and received 3 CVEs, 2 of them are classified as critical (CVSS3.x score CRITICAL 9.8) and affected 40 different products in ICS. Our prototype is open source at https://github.com/boofish/ICS3Fuzzer.

2 BACKGROUND AND MOTIVATION

2.1 ICS Architecture

An ICS is a distributed computerized system that manages, monitors, controls, and automates industrial processes. ICSs have been widely deployed in critical infrastructures. As shown in Fig. 1, the architecture of a typical ICS (SCADA system) consists of three layers [11]: (i) *Field Instrument Control Layer* consists of sensors and actuators that serve as the input/output of the system; (ii) *Process Control Layer* consists of dedicated embedded devices (e.g., PLCs/RTUs) to control real-time processes. They are connected to the field devices to sample sensor readings and to operate actuators based on the control logic programmed and loaded by the supervisory control layer; (iii) *Supervisory Control Layer* consists of several workstations used to provide real-time monitoring and control capabilities of the devices in the system. For example, the system operator can monitor and control the whole ICS via the Human-Machine Interface (HMI) machines, while the system programmer can program and debug the PLCs with an Integrated Development Environment (IDE). In this paper, we collectively call these pieces of software supervisory software.

There are several distinct characteristics in an ICS environment. First, an ICS directly interacts with the physical world. As such, a defect in it could impose a significant risk to the safety of human lives or cause serious damage to the environment [37]. Second, to ensure safe operation, an ICS must provide high reliability and meet specific real-time constraints. This necessitates the use of real-time OSes at the process control layer. Also, domain-specific protocols, which are usually proprietary, are used to interconnect the three layers. To decrease overhead, the proprietary protocols usually take place in an unencrypted form [13]. Third, historically, the ICS network is air-gaped, meaning that they are disconnected from the Internet to reduce the attack surface. Therefore, the lack of needed security mechanisms such as encryption was not a very big concern. However, with the emergence of IIoT, this characteristic no longer holds in many cases. This fact exposes vulnerable legacy software open to attackers.

The supervisory software plays an important role in an ICS environment. Specifically, they issue requests to pull real-time status from the PLCs. They can also be used to program and install firmware images for PLCs. Although individual ICS vendors provide different solutions, the supervisory software shares lots of similarities regarding the implemented functionality. For example, almost all the supervisory software offer interfaces to read operational variables, start/stop PLC, and download the firmware, etc. These functionalities are communicated over a vendor-specific channel, which carries three types of data, including (1) real-time device status (e.g., sensor readings, feedback of control command execution, heartbeat messages, etc.), (2) device information (e.g., device name, version, model, manufacturer, and other information), (3) structured data block (e.g., program blocks, memory blocks, diagnostic files, etc.).

2.2 Security Risks Exposed by Supervisory Software

In manufacturing industry, both OT network and IT network are required to let a factory floor produce designated products. Supervisory software is running (under the supervision of a human operator) on a workstation inside the OT network. Although strong isolation is enforced between the IT network and the OT network to better defend cyber-attacks, APT (Advance Persistent Threats) attacks could still break the isolation and compromise part of the

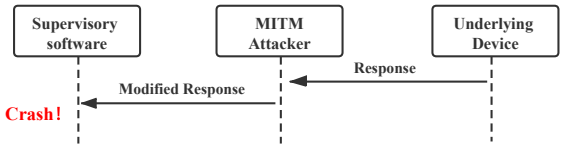

Figure 2: Crashing the supervisory software

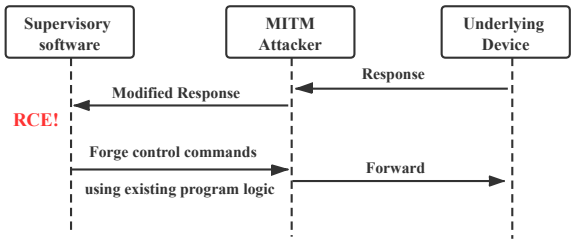

Figure 3: Forging malicious commands with RCE

OT network. In real-world exploits, reusing control logic of supervisory software in an APT attack was attractive and clearly observed (e.g., Stuxnet attack [12]).

Assumption. Before exploiting the supervisory software, we assume that a powerful attacker has accessed one host of ICS. On this basis, we also assume that the powerful attacker can then monitor, intercept, and modify the network communication of other hosts based on a man-in-the-middle (MITM) attack. In real-world ICS exploits, MITM is commonly used. This includes the infamous Stuxnet [12] and IRONGATE [20]. According to a report conducted by U.S. Department of Homeland Security (DHS), ICS protocols or communication channels are particularly vulnerable to MITM attacks [30]. Other related studies [18, 22, 36] hold the same opinion.

Attack Approaches and Consequences. With the MITM, the attackers can intercept and manipulate the network traffic between the supervisory software and a PLC. If the supervisory software is vulnerable, the attackers can achieve multiple disastrous attack goals. First, they can crash the supervisory software (Fig. 2). This kind of attack is also demonstrated by Senthivel et al. [36]. As a result, the devices in the process control layer lose control, and the real-time status cannot be updated. Second, by exploiting a remote code execution (RCE) vulnerability, the attacker can leverage existing logic of supervisory software to forge control commands, such as installing a malicious firmware into the PLC devices (Fig. 3). These attacks root in the fact that ICSs were designed for closed environments, where every node in the network is trusted and works as intended. However, this assumption no longer holds with the development of IIoT.

In both cases, it is clear that the attacker relies on an exploitable implementation bug in the communication protocol between the supervisory software and PLC devices. Finding these bugs beforehand is crucial for the security of ICS.

3 DESIGN OVERVIEW

3.1 Movitating Example

We first show a real TCP session between supervisory software GX Works2 and a PLC (see Fig. 4). During the TCP session, there are many **input states** exposed to obtain network data. Note that in each particular input state, the supervisory software waits for *a certain kind of* network data from the PLC device. In this paper, when we discuss the state-space of the supervisory software for a PLC, we are essentially discussing its different input states. To avoid ambiguity, we informally define a *session type* as a distinct functionality offered by the supervisory software, where a sequence of packets are exchanged in both directions. A **functionality** is one kind of supervisory semantic (e.g., download program to PLC, start PLC, show status). We note that given the supervisory software, we can leverage domain knowledge (e.g., manuals) to learn most if not all of its functionalities and session types.

Key observation. (a) The start point of an above-mentioned TCP session is usually a button-pushing operation performed by a human operator through the GUI provided by the supervisory software. As soon as the operator clicks the corresponding button, the supervisory software will conduct the 3-way handshake with the PLC device. Then packets will be exchanged in both directions. (b) In the middle of the session, we observe that in many cases additional button-pushing operations are performed by the operator. For example, when the functionality is downloading a program onto the PLC, the operator firstly clicks the "connect" button to create a new TCP session. However, this button-pushing operation itself cannot complete this task/functionality. Hence, a few moments later, the operator will click the "download" button to let the same TCP session fully complete its mission. Without the operator pushing the "download" button, the supervisory software will not start sending the program's content to the PLC. (c) Finally, the session ends by periodically exchanging heartbeat messages or stopping all the message exchanges.

New Insight. The key observation indicates that the supervisory software usually has different behavior (i.e., code execution) in different *interaction states*. An interaction state of the supervisory software is essentially a particular program execution context in which it interacts with the PLC device. An interaction state is determined by several factors, including the previous button-pushing operations and the previous input states. For example, without the operator firstly pushing the "connect" button, the "download" button will remain greyed out and not clickable. For another instance, whether an input state can be tested depends on whether the current interaction state is the right one. This also indicates that the protocol implementation of supervisory software is high-coupled with GUI interfaces.

3.2 Challenges

Based on the key observation and the new insight, we found that testing supervisory software is very challenging. In particular, we summarize following three challenges.

C1: Guiding the supervisory software to enter a specific input state. In the supervisory software's viewpoint, each session involves multiple input states. Accordingly, there are at least three

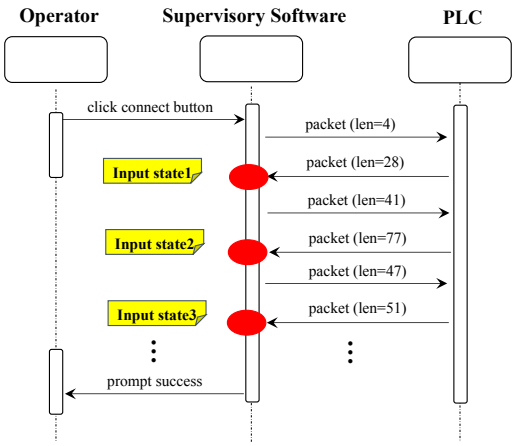

Figure 4: Interaction between supervisory software and device.

barriers to achieve a good coverage of these input states. First, because all the TCP sessions are initiated by the client-role supervisory software, the fuzzing tool has to (passively) "wait" until receiving a session-establishing request from the supervisory software. Second, to fuzz an input state, we must trigger the supervisory software to enter the right interaction state again and again, which involves the synchronization among three entities: the button-pushing GUI operation, code execution in the supervisory software, and the device response (see Fig. 4). Without designing a mechanism to manage the GUI operations and the network traffic automatically and synchronously, it is pretty hard to enable the fuzzing tool to feed mutated data repeatedly to a specific input state. Third, due to the sophisticated dependency relationships between the input states, in many cases one input state cannot be directly entered without the supervisory software firstly entering some other input states. Even worse, identifying a rather complete set of input states based on traffic is itself a challenging problem. Please note that different supervisory software have different input states.

C2: Fuzzing proprietary protocols with unknown message frame format and state-space. Supervisory software generally uses a proprietary protocol to communicate with a PLC device, and the protocol has unknown state-space (i.e., input states) and format. Unknown state-space makes it less likely to explore deep paths and deep protocol states. In addition, with unknown packet frame format, the value constraints for certain fields and the relevant dependencies between fields are hard to be inferred. This may limit a fuzzing tool's ability to generate effective inputs.

C3: Simulating the session of proprietary protocol. Before entering a specific input state, each request issued by the supervisory software, we must provide the correct response. A straightforward approach is to use the real device to maintain the interaction. However, the solution depends on hardware is expensive and not scalable. Simulating the session is a promising choice but it requires a comprehensive understanding of proprietary protocols.

To address the challenges, for C1, we design a novel control mechanism to achieve entering any identified input states by accurate synchronized control of GUI operations and network communication. For C2, we adopt an existing work [5] to reverse-engineer the packet frame format of the protocol. Besides, we also perform a differential analysis to identify the fields and recognize value constraints for such fields as session ID, sequence number, etc. In order to identify the valuable protocol states and filter duplicated states, we avoid reverse-engineering the detailed program states. Instead, we construct a state-book based on execution trace and the corresponding inputs. To achieve good state coverage and improve the efficiency of *ICS*^{3} *Fuzzer*, we propose a new fuzzing strategy to switch states dynamically. For C3, we simulated the response of the PLC device by constructing a set of communication templates/patterns based on real captured traffic. For each request from the supervisory software, *ICS*^{3} *Fuzzer* firstly identifies the corresponding response in the matched template. *ICS*^{3} *Fuzzer* then automatically finds/adjusts the corresponding value constraint (i.e., session ID, sequence number, etc.), if any, for each involved field. In case the constraints and dependencies are too complex to be found, we will resort to the real PLC device.

4 DETAILED DESIGN

We present the detailed design of *ICS*^{3} *Fuzzer*, as illustrated in Fig. 5. At a high level, *ICS*^{3} *Fuzzer* has two phases: i) In the pre-processing phase, *ICS*^{3} *Fuzzer* produces necessary information and tools to assist the fuzzing phase to be fully automated. ii) In the fuzzing phase, *ICS*^{3} *Fuzzer* switches input states dynamically, then feeds the generated test cases and monitors the status of the supervisory software.

In the pre-processing phase, we analyze the functionality and the proprietary protocol of a given supervisory software. For functionality, we focus on how to automatically start a session during the fuzzing loop. For proprietary protocol analysis, we focus on how to obtain the important knowledge of its state-space/format to assist in choosing valuable input states and generate effective mutated data. Besides, we also focus on how to simulate the communication based on captured traffic to avoid being limited by the dedicated hardware.

In the fuzzing phase, our framework is fully automated and consists of four steps to work coordinately (see right-side of Fig. 5). Firstly, it chooses a valuable input state based on information provided by state-book. Secondly, it generates mutated inputs based on protocol format knowledge inferred. Then it feeds the mutated data to the chosen input state based on automatically managing GUI operations and network traffic synchronously. Finally, it monitors the status of the supervisory software and records the malformed input for reproduction.

4.1 Functionality Analysis

The goal of this step is to prepare UI components that will lead to performing a task/functionality, producing a session through network interface. We can leverage domain knowledge (e.g., manuals) to learn most if not all its functionalities. Then we can start a session of a functionality and expose input states. In order to **automatically** perform functionality and start the session in fuzzing, we need to prepare some "activators" (e.g., pushing a specific button) in advance. An "activator" can be invoked to trigger GUI events (e.g.,

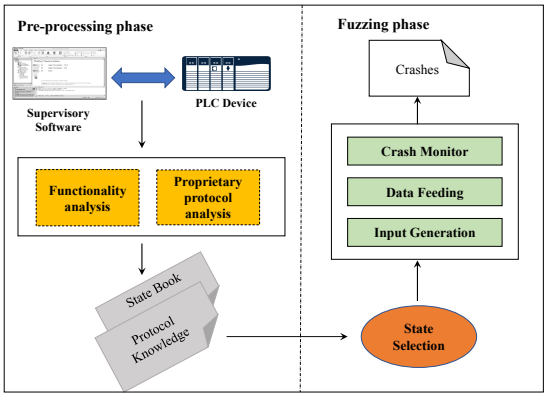

Figure 5: *ICS*^{3}*Fuzzer* System Architecture

keystrokes, mouse movement). For our discussion, we refer to this kind of "activator" as a *guiAutolit*. In practice, we write guiAutolits based on AutoIt Framework [3]. Two steps are needed: (i) Obtaining GUI handle: A GUI handle uniquely identifies a GUI Control, which can be easily obtained by AutoIt. Then we can write scripts to manipulate GUI controls by providing handles to API functions (e.g., *ControlClick()*, *MouseClick()*) defined in AutoIt. (ii) Defining operation orders: A functionality is performed by a sequence of GUI operations. Prerequisites of an operation require that the GUI has been loaded and the corresponding control is enabled. For a given functionality, its GUI operation orders can be defined manually. Finally, the *ICS*^{3}*Fuzzer* can invoke these guiAutolits to automate the fuzzing loop.

4.2 Proprietary Protocol Analysis

To fuzz supervisory software efficiently, this step obtains important knowledge about the the proprietary protocols used in the target software. The specific tasks include inferring the protocol format, obtaining state space, and building a template for simulating the communication session.

4.2.1 ***Inferring Protocol Format.*** Inferring the format of proprietary protocol helps to guide the generation of effective test cases. For this goal, there are many existing works proposed such as Discoverer [9], Dispatcher [7] and Netzob [5]. Existing work is able to meet our framework design needs, therefore, we integrated one of the tools for easy development [5]. Details of inferring protocol format are elided as it is not our contribution.

4.2.2 ***Obtaining State-Space.*** This step is to help identify and distinguish input states in a session. In the viewpoint of supervisory software, an activated session involves many input states (e.g., more than 100 input states in the session when activating "show status" functionality of "Proficy Machine Edition"). However, there are also many input states are repetitive and similar (e.g., the state corresponds to receiving heartbeat messages). To distinguish these input states, a straightforward solution is to compare the similarity of messages, but it is not accurate. Instead, we record the execution trace triggered under the corresponding input state, where even small differences in the messages but very different execution traces can be easily distinguished. Therefore, when fuzzing, excessive resources should not be spent to test these similar input states. Instead, input states that can trigger rich and different execution traces are more interesting. With this knowledge, we can switch valuable input states for more efficient fuzzing (see 4.3).

Since our framework is universal, we avoid analyzing the input states of a particular ICS protocol. Instead, we construct a state-book based on code execution trace and the corresponding inputs, without recovering the specific semantics of an input state. To collect the execution trace, we leveraged the DynamoRIO [6] to dynamically instrument the target supervisory software. Specifically, we only recorded/dumped bitmap of the basic block executed during message transmissions (by hooking and tracking "send()/recv()" function). This is because we are only interested in the message processing logic under the input state. Meanwhile, we also recorded the original messages and their orders. Therefore, each input state in state-book can be represented as a tuple, including the original message, execution trace, and an index.

4.2.3 ***Device Emulation.*** In fuzzing, we need to act as a PLC device role to feed the test cases to the supervisory software. Using a dedicated PLC device for this purpose not only increases cost but also is not scalable. To avoid being limited by the dedicated hardware, we simulate the communication based on captured traffic. Implementing this idea faces two challenges as mentioned by Sushma Kalle [22]: When supervisory software initiates a request, we need to (1) identify the corresponding response message in the captured traffic, and (2) adjust the dynamic fields related to maintaining a session in the response message, such as session ID, sequence number fields, etc.

For the first challenge, we have insight that the contents of the message are essentially the same when captured from the same functionality with the same device. In other words, for a specific run-time request from the supervisory software, there is a corresponding record in the captured traffic. Therefore, we can construct a request-response dictionary, in which the key is a request and the value is a response.

For the second challenge, assuming that the protocol has a session ID field to maintain the session. The field's value is generated in real time and varies in different sessions. If we do not adjust the field and directly replay the response identified from the captured traffic. Then the supervisory software will enter an error state, causing subsequent session to fail. As a result, we cannot reach the expected input state. We have summarized four dynamic fields for industrial control protocols based on our analyses of 4 real ICS platforms (see Section 5) and related work [4, 10].

- **Sequence number**: The field is used to represent the message received/acknowledged, and it always increases 1 after each interaction.
- **Session ID**: The field is used to represent a temporarily created network session. Generally, the value of session ID is issued by the device, then messages in the interaction will carry this value.
- **Time stamp**: The field is used to represent the freshness of the message. It relates to the real-time of running system, and the receiver will check it. If it is found to be stale, the message will not be well-processed.

- **Challenge-response**: Generally, the device sends a message with a challenge (random number) encoded, and the supervisory software will calculate a response based on the challenge. If the response is not correct, the session will be terminated or the program enters a wrong state.

To identify the dynamic fields, we have two observations:

Observation 1: The heartbeat messages are common and their contents are very similar but not the same. The difference reflects dynamic fields of the proprietary protocol, such as the design of sequence number.

Observation 2: The sequence of network traffic generated from *the same functionality* is very similar. Each request-response pair of one sequence could match a pair (most similar but not the same) in another sequence. The difference of the corresponding messages also reflects dynamic fields.

Based on these observations, we align and compare the differences between the corresponding messages to eventually obtain many dynamic fields. For dynamic fields that are too complex to be understood, we also dynamically replayed the related message (correct the known dynamic fields) and determined whether the field is related to the session management. If we cannot resolve the dynamic field, we will resort to the real PLC device. Specifically, for each request issued by the supervisory software, *ICS³Fuzzer* will forward it to the real PLC device, and analyze the response from the device, then we decide whether the response should be mutated or directly forwarded to the supervisory software. Therefore, before enter a specific input state, any request issued by the supervisory software, *ICS³Fuzzer* can provide the response correctly.

4.3 State Selection

Based on the constructed state-book, each input state has three attributes: index, execution trace, and the original message. For each attribute, we assign a numerical value so that each state has a weight. We leverage these weights to select the most promising state to fuzz against. The weight assignment is based on three hypotheses. (i) The "deeper" network communication is, the more likely there exists a bug; (ii) When a message is forwarded to the supervisory software, the more new basic blocks are hit, the more likely there exists a bug; (iii) The more complex the input is, the more likely it is to cause a crash.

Hypothesis (i) is well accepted in proprietary protocol fuzzing. For the input state of a protocol, we use the index of its order in the state-book to represent the degree of "depth". In building the state-book, we ignore states with high similarity with existing ones, such as those exhibiting the same repeated behaviors. By repeated behaviors, we mean the two states share many similar messages and execution traces (i.e., the hit counts of blocks in bitmap). In this way, we can avoid putting into the state-book the states with "high" degree of depth but are actually very superficial based on the protocol specification (e.g, a state with many heartbeat messages). We use Jaccard index to quantitatively measure message similarity, which is the number of common fields between two message sequences divided by the number of total fields in the union of the two message sequences. The bitmap similarity is calculated simply by the number common bits in the bitmap divided by the total hit counts of bitmap. The final similarity is the result of multiplying the message similarity and the bitmap similarity. To decide if a state has a high level of similarity with an existing state and thus should be ignored, we set a threshold. Obviously, the lower the threshold is, the more states would be filtered out. On the contrary, the higher the threshold is, many redundant states would be in the state-book. To find a sweet spot, we set different thresholds for similarity checks and manually examined the results. Our experiments showed that using the threshold of 70%-95% can filter out many repeated behaviors (e.g., heartbeat messages), while retain a good number of unrepeated behaviors. In our evaluation, we empirically chose 80% as the threshold to check state similarity. Hypothesis (ii) is consistent with feedback-based fuzzer such as AFL [43]. Hypothesis (iii) concerns with the complexity of the input. The more complex the input is and the more diverse the mutation is, the more likely it is to trigger new execution paths. We use the number of identified fields within a message to indicate its complexity. In summary, we use state depth, basic blocks count, and field count within a message (depth, bb_count, field_count) to represent the three state attributes mentioned before. We use these values to calculate the weight of each input state via the following formula.

$$w_i = \frac{1}{3}\left(\frac{depth_i}{\sum_{j=0}^{N-1} depth_j} + \frac{bb\_cnt_i}{\sum_{j=0}^{N-1} bb\_cnt_j} + \frac{fld\_cnt\_i}{\sum_{j=0}^{N-1} fld\_cnt\_j}\right)$$

In the formula, $\frac{1}{3}$ is used to normalize the sum of weights. N is the number of states within a given session. Since selected input state is computation intensive, it is only performed periodically during fuzzing. The whole process is summarized in Algorithm 1. **Line 1** filters out those repeated input states based on pre-processing phase results. **Line 2** calculates the weight of each input state according to the state-book. **Line 4** performs state selection according to state weights.

Algorithm 1 State-selection based fuzzing loop

Input: Limited state count for a functionality, N
Limited test case for a specific input state, M
State book of a functionality, S
Output: Crashes Record, C

```
1:  states ← FILTERED(S)
2:  weights ← GETWEIGHTS(states)
3:  for i ← 0; i < N; i++ do
4:      cur_state ← CHOOSE(states, weights)
5:      j ← 0
6:      while j < M do
7:          mutated_input ← GENERATE(cur_state)
8:          result ← RUN(cur_state, mutated_input)
9:          if IsCrash(result) then
10:             item ← TUPLE(cur_state, mutated_input)
11:             C ← APPEND(C, item)
12:         end if
13:         j ← j + 1
14:     end while
15: end for
16: return C
```

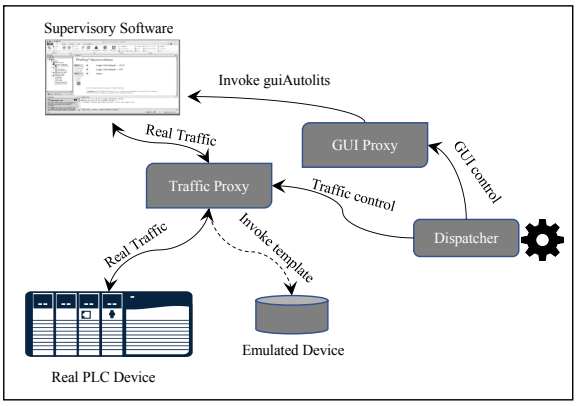

Figure 6: Communication topology of the *ICS³Fuzzer*

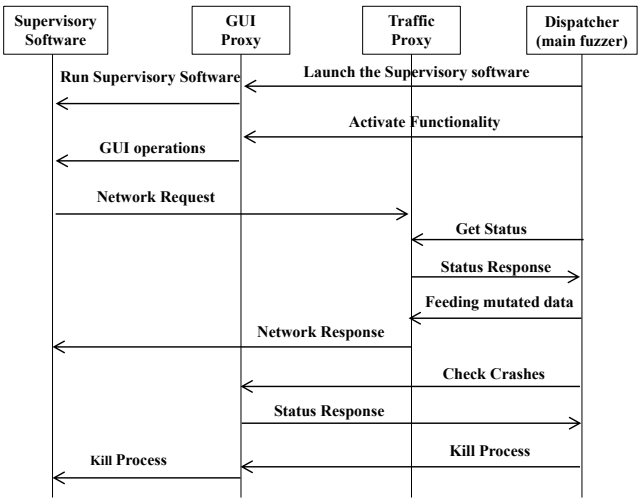

Figure 7: Workflow of proxy-based data feeding mechanism

4.4 Inputs Generation

This step leverages the obtained protocol format to generate mutated inputs. There are two tasks to be completed: (i) generate inputs according to the reverse-engineered protocol format and (ii) correct it with the discovered constraint relationship. For the first task, instead of fully mutating one field, *ICS³Fuzzer* tries to cover all fields as much as possible. After data is generated, we will correct it with constraint relationships such as length field, session ID, and sequence number constraints. After this step, the input state and the mutated input are well-prepared.

4.5 Data Feeding

To make the supervisory software process the mutated data, feeding data is the basis. Different from a regular target, fuzzing supervisory software needs to synchronize network behaviors and GUI operations. *ICS³Fuzzer* achieves this synchronization goal through a proxy-based control mechanism. Specifically, it combines the *traffic proxy*, *the GUI proxy*, and *the Dispatcher*. Fig. 6 shows its communication topology. The traffic proxy intercepts the network traffic between the supervisory software and "the device" (Note that we use emulated device based on traffic simulation and we would resort to a real PLC device if the simulation fails). The GUI proxy controls the GUI operations, including launching the program, pushing buttons and killing the process. The Dispatcher sends commands to the traffic proxy to manage network traffic, and sends commands to the GUI proxy to control GUI operations of the supervisory software. On this basis, *ICS³Fuzzer* can feed data to the supervisory software automatically. Fig. 7 depicts the workflow of data feeding.

4.6 Crash Monitor

This step is to monitor the status of supervisory software to capture the triggered crashes. Our monitoring is based on the Windows *EventLog* Service. Once an application crashes, a record will be added in the *Eventlog* of Windows System with a tag *Application Error*. The record not only includes crash information, but also provides the context of crash to assist diagnoses. Therefore, in each fuzzing cycle, we check the real-time records of the *Application* category of the Eventlog. Note that we do not instrument the supervisory software during fuzzing due to performance consideration (see Section 5.3). We only instrument the program during the pre-processing phase to help analyze the protocol.

Program hangs are also widely used as an indicator of program bugs which can potentially be exploited in DoS attacks. However, based on our empirical study, detecting delays of ICS operations and using them as bug indicators incur lots of false positives. This is because whether a long delay should be considered as a bug or not depends on specific scenarios. There is no standard to specify the cutoff value to separate a normal response and a delayed response, which makes it hard to filter out false positives. In our evaluation, we therefore adopted a conservative strategy that does not consider bugs causing by program hangs. However, this feature can be easily enabled in the current prototype.

5 IMPLEMENTATION AND EVALUATION

5.1 Framework Implementation

We have implemented a prototype of *ICS³Fuzzer*, with 4000 lines of codes, including 2400 lines of Python code, 900 lines of AutoIt script, 750 lines of C/C++ code. It could be conceptually split into two phases of operation. In the pre-processing phase, we prepared and collected necessary scripts/data for each target with minor manual efforts. Specifically, we prepared a *guiAutolits* script for automatic GUI operation of supervisory software based on AutoIt [3], and obtained the protocol format based on Netzob Framework [5]. We then leveraged the DynamoRIO Framework [6] to instrument the supervisory software so that we could collect its run-time execution trace. In the fuzzing phase, we leverage the reverse-engineered protocol formats to generate mutated inputs based on the work [34]. We then fed the inputs with the synchronous control of GUI proxy and traffic proxy, which was implemented from scratch in Python. Finally, we used a script to check the Windows Eventlog service to detect crashes.

Software Selection. We target four pieces of supervisory software as part of ICS platforms from different vendors. We summarize

the detailed information of the targets (name, vendor, version, binary size, and protocol) in the Table 2.

Testing Environment. *ICS³Fuzzer* ran on a Windows 7 machine powered by an Intel Xeon E3-1505M v6 CPU running at 3.00 GHz with 8 GB DRAM. The complete experiment environment also included two proxies (traffic proxy and GUI proxy) on the Window machine, and the target supervisory software.

5.2 Effectiveness and Comparison

Effectiveness in discovering bugs. By performing fuzz testing on four different pieces of supervisory software with the *ICS³Fuzzer* framework, we find 13 memory corruption bugs. All of them have been responsibly reported to the vendors or the third-party vulnerability database maintainers and got email confirmed. We summarize the type of the bugs and the influenced vendors in Table 3. Among them, 3 bugs have been assigned CVE numbers (CVE-2019-16353, CVE-2021-20587, CVE-2021-20588). For bugs assigned with CVEs, 2 have been fixed by the vendors after receiving our report. These memory corruption bugs are of high impact when exploited. By crafting a proof-of-exploitation (PoE), we find that they can cause denials of service (DoS) or remote code execution (RCE). For example, according to an evaluation conducted by NVD (National Vulnerability Database) based on CVSS 3.x, the threat of the vulnerability CVE-2021-20587 and CVE-2021-20588 is "CRITICAL" (basic score is 9.8), and *40 different products* are affected [33]. The wide applicability of this bug can be explained by the fact that the ICS manufacturers often implement their proprietary protocols as shared libraries and reuse them in all the relevant product lines. This also highlights the necessity of discovering protocol implementation vulnerabilities as early as possible.

Effectiveness in pruning input states. During the session, to avoid wasting too many resources to fuzz these duplicate states, we prune the states to build a deduplicated state-book (assigning the weights of input states to 0). It can be seen from Table 4, these include 22 sessions of 4 supervisory software, and we get 203 input states by pruning nearly 1500 original input states.

Comparison the effectiveness of state selection. We are interested in the effectiveness of state selection strategy proposed in *ICS³Fuzzer*. As mentioned before, there is no fuzzer designed for testing supervisory software. For example, no fuzzer supports guiding the target software to a specific input state since no GUI or traffic proxy is provided. To this end, we kept the original strategy that randomly selects an input state from the unpruned state-book. We call this fuzzing tool *ICS³Fuzzer*-less. We used *ICS³Fuzzer* and *ICS³Fuzzer*-less to test 22 different functionalities. For each functionality, we allocated 48 hours. The tasks were paralleled using device simulation. *ICS³Fuzzer* still performs better than *ICS³Fuzzer*-less and can find 5 more vulnerabilities within the same time.

5.3 Performance

Due to the non-deterministic nature of fuzzing, after each fuzzing cycle, the GUI of the supervisory software is also non-deterministic. For example, the GUI could be frozen or some unforeseen interface would pop up. In order to ensure the stability of the fuzzing process and for the convenience of implementation, after feeding the test case and checking the status of the program, our prototype implementation will kill the process and restart the software for a new iteration. Therefore, the time-cost is mainly composed of four parts: restarting the supervisory software, operating the GUI, network communication, and others. Table 5 gives the average time cost of our experiments. Although the speed of a single test case is slow (ranging from 5-20 sec a test case), it can be fully automated and thus effectively fuzz commercial supervisory software. In the meantime, since our solution does not rely on dedicated hardware, the test speed can be increased by running multiple instances in parallel. In the following paragraphs, we show the results of two experiments to explain the implementation choice of our prototype.

Why NOT use Feedback-based Method. During early development, we tried to implement a coverage-based feedback mechanism to guide the test case generation using the genetic algorithm. Specifically, we used the DynamoRIO framework [6] to instrument the whole supervisory software to dynamically collect branch information. Similar to AFL, we maintained a run-time bitmap of branch coverage which was fed to the AFL-based mutator. The initial seed was a file consisting of messages that cover all input states of a given TCP session. Finally, the mutator sent the test case to the dispatcher, which is responsible for feeding the test case to the target supervisory software. In our test on GX works2, no bug was found during a 48-hour fuzzing campaign for a session.

Taking a close look at the fuzzing process, we found that the fuzzing speed greatly impacted the fuzzing efficiency. In fact, we found that GX Works2 spent 34.9 seconds to restart with instrumentation, while this number was 1.1 seconds without instrumentation. Due to the significant slowdown, only about 4.5K test cases were tested for 48 hours. Without a sufficient number of test cases to be tested, the genetic algorithm can rarely make any progress. In fact, in this experiment, we observed that most of the test cases (more than 95%) cannot make through the initial input state $\mathcal{S}_0$. As a result, the deep state space of the protocol implementation cannot be touched at all. To achieve a particle fuzzing system, *ICS³Fuzzer* sacrifices the advanced feedback mechanism for a faster fuzzing speed. Future research is needed to further improve the feedback-based methods in terms of high fuzzing speed (e.g., via parallelization) and better protocol state management mechanisms.

Why NOT use Snapshot-based Method. Based on Virtualbox, we have implemented a prototype similar to [39]. Where a proxy server and supervisory software are in the same virtual machine, it can keep the state of the live connection between supervisory software and the proxy. Once the virtual machine is rollbacked, our outside fuzzer feeds the mutated input via the proxy server. However, the process is also very slow. Time is mainly spent on: 1) snapshot/rollback of a complete Windows System. 2) establish the connection to the proxy server with slow network initialization and routing. It almost costs 32 seconds to complete feeding the mutated data to GX Works2. Moreover, if the protocol is time-sensitive, the connection between the proxy and supervisory software will be closed due to slow network recovery. For example, one of our targets - Proficy Machine Edition, close the inner connection, then we could not feed the mutated data. In addition, maintaining Windows System snapshots of different protocol states would take up massive storage space.

Table 2: Summary of Supervisory Software under Testing

Vendor	Software	Version	Image Size	Device	Proprietary Protocol? (Yes,No)
Mitsubishi	GX Works2	1.591	671M	Q06UDEHCPU	Yes
Emerson	Proficy Machine Edition	9.0	1200M	GE RX 7i	Yes
Schneider	TwidoSuite	2.20.11	79M	TWDLCAE40DRF	Yes
Panasonic	FPWIN GR	2.95	106M	FP-X C60R	Yes

Table 3: Summary of vulnerabilities found

Software	Vulnerability Type	Number	status
GX Works2	HeapOverflow Unknown crash	6 1	confirmed
Proficy Machine Edition	NullPointer Dereference Unknown crash	1 3	confirmed
TwidoSuit	Unknown crash	1	confirmed
FPWIN GR	Unknown crash	1	confirmed

Table 4: Summary of sessions under Testing

Software	# Selected session	# Origin states	# Post-pruned states
GX Works2	8	326	93
Proficy Machine Edition	8	842	52
TwidoSuit	2	136	27
FPWIN GR	4	200	31
TOTAL	22	1504	203

Table 5: Time-cost for fuzzing supervisory software (sec)

Software	Launch	GUI Operations	Network Communication	others
GX Works2	1.095	2.363	1.850	3.415
Proficy Machine Edition	4.115	4.349	9.374	5.356
FPWIN GR	1.114	0.361	1.274	2.371
TwidoSuite	6.765	0.592	5.444	2.157

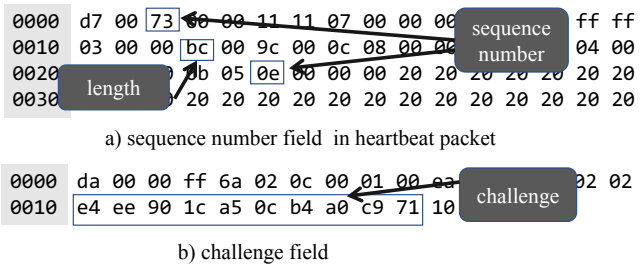

Figure 8: Dynamic fields in the protocol of GX Works2

5.4 Case Study: GX Works2

In this section, we discuss a bug and a dynamic field *ICS³Fuzzer* discovered in the GX Works2. GX Works2 is an integrated PLC Engineering Software from Mitsubishi Electric. Note that we have obtained the permission from the vendor for the disclosure, and the bug has been fixed.

5.4.1 Dynamic fields analysis. To identify dynamic fields, we collected and analyzed a series of heartbeat messages. We then easily identified that the offsets of 2 and 37 correspond to the dynamic fields (sequence number field). After collecting sequenced traffic of the same functionality twice, we found a new dynamic field when comparing the corresponding messages (as shown in Fig. 8).

For the same input state, every time the same session is triggered, there are 10 bytes that are always different, and it changes dynamically. By replaying the dynamic field, we can enter the subsequent state, indicating that the dynamic field does not need to be adjusted.

To further understand the reasons for not adjusting, we reverse engineered the related processing logic of the program. Specifically, when GX Works2 receives the message, we set a memory breakpoint towards the dynamic field through a debugging tool. By tracking the subsequent execution trace, we located this field to be processed by the function *sub_17224* of module *ECUNIT_PLC_QN.dll*, which then called the function *sub_61247* to generate a 32-byte output. Combining further analysis of communication messages, we figure out that it is a challenge-response mechanism: the 10-byte challenge initiated by the PLC device and the supervisory software generates a 32-byte response by processing it. Since we are replaying the challenge field from the PLC device, the dynamic field does not need to be adjusted. This also proves that the supervisory software has default trust on an isolated PLC device.

5.4.2 Vulnerability discovery. In Fig. 9, we show a crash log caused by a bug (CVE-2021-20587) we found in GX Works2. We also list the input message that were used to trigger this bug. The crash was caused by feeding the 3329-th mutated data under state $\mathcal{S}_1$ within the functionality "Connection Test". To reproduce the crash, we automatically controlled the proxies to steer the supervisory software to go through state $\mathcal{S}_0$ to state $\mathcal{S}_1$. Then we fed the recorded data to the input state, and the crash was reproduced.

By manually analyzing the crash, we have confirmed that it was a heap overflow bug, which was triggered by function sub_121A of the ECUNIT_PLC_QN.dll module. The root cause of the vulnerability is the lack of length check when supervisory software receives the message under state $\mathcal{S}_1$. Interestingly, when we tried to feed the mutated data to other input states, we found it could not cause a crash. This proves that the bug is strongly related to the specific input state.

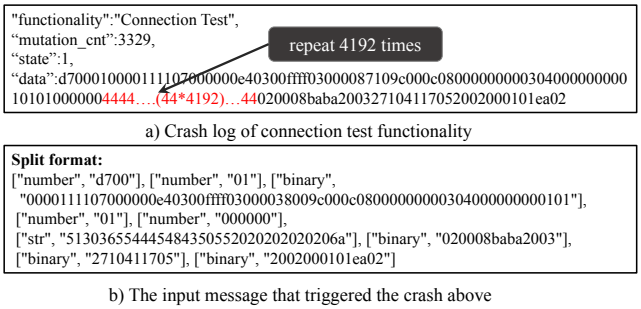

Figure 9: Crash log of the GX Works2 and the triggering input message

5.5 Manual Work

Automating the whole fuzzing process is not easy, since necessary pre-processing is needed to handle heavy GUI and proprietary protocol, which we admit is manual effort. The manual effort mainly includes a) exploring GUI interface, b) writing activators, c) obtaining protocol knowledge, and d) verifying the accuracy of the analysis. Generally speaking, several hours are needed for each ICS supervisory software. Note it is a one-time job and the following fuzzing process is fully automated.

For a), we use expertise knowledge to identify the GUI interface. It generally takes several minutes. For b), we pre-define the GUI operation orders and locate the positions of the corresponding button. It typically costs dozens of minutes assuming the user is familiar with AutoIt. For c), we collect traffic and obtain bitmap of execution traces. Then we write scripts to analyze these data. We manually identify the type of dynamic field. It costs several dozens of minutes or even several hours for the complex protocols. For d), we verify the stability by automating each involved functionality. In the tests, we indeed encountered issues of instability, mainly due to inaccuracy of analysis. We have analyzed the root cause and provided general approaches to handle these issues. Specifically, there are three common reasons: (i) The device state is changed when capturing traffic. We solved it by keeping the device with the same target program. (ii) The software pops up abnormal windows or hangs for incorrect simulation. We solved it checking the communication template and dynamic fields. (iii) Wrong buttons are pushed because of the flaw in pre-defined GUI operations. We need to correct the pre-defined operations to solve it.

6 RELATED WORK

In this section, we mainly discuss the relevant work on the following four topics: protocol re-engineering, protocol fuzzing, GUI-related testing, and security of supervisory software.

Protocol Re-engineering. The goal of re-engineering a network protocol is to obtain knowledge of message format and protocol state machine. Existing techniques fall into two categories. One analyzes program execution semantics such as execution traces and memory traces [7, 31, 40]. The other focuses on analyzing network traces such as aligning message sequences [5, 9, 23, 42]. These methods do help to a better understanding of the proprietary protocols. However, in terms of obtaining accurate semantics of protocol such as challenge-response mechanism, session ID, timestamp field, etc, the existing work still needs manual work and is time-consuming. In contrast, our framework adopts an existing work to re-engineering the protocol formats and proposes differential analysis to help manually obtain accurate semantics of protocol for device simulation.

Protocol Fuzzing. Existing studies focus on protocol fuzzing can be mainly categorized as 1) black-box fuzzing [1, 15, 32, 41] where the approach can only leverage the input and output of the target program. 2) grey-box fuzzing [25, 26, 35] where the approach can obtain internal program state information from binary or source code. To achieve testing deep state-space of programs, both categories need to carefully manage different protocol states during fuzzing. SNOOZE [2] and Achilles [41] use user-defined scenarios and protocol specifications to fuzz different protocol states. Rui et al. [27] proposed a "stateful rule tree" based solution to test and iterate different protocol states. Suzaki et al. [39] proposed a snapshot/rollback mechanism to fuzz and recover different protocol states. Pulsar [15] built a communication template and used the Markov chain to represent state transition. Most of the work focus on the server-role target and the protocol state transition can be easily achieved only by sending data. In the viewpoint of supervisory software, protocol states are high-coupled with GUI operations, few works can be applied to achieve deep state-space fuzzing. In contrast, *ICS³Fuzzer* can well-manage input states based on a strategy (see section 4.3) and automatically enter any identified input state by letting the two proxies enforce coordinated synchronized controls of GUI operations and network.

GUI-related Testing. For fuzzing closed-source Windows GUI binaries, existing work focuses on identifying the relevant module to write harness for focused fuzzing, such as WINNIE [21] and WinAFL [14]. However, WinAFL cannot be used for its strict requirements of target and the protocol is not implemented as a pure function. WINNIE cannot be applied either due to high-coupled protocol implementations with GUI in supervisory software. In addition, our work is fundamentally different from existing GUI fuzzing works (e.g., [38] [17]) that attempts to find bugs by GUI operations. Our work relies on GUI operations to restore the target program to a pre-determined state and then find bugs via the network interface.

Security of supervisory software. In addition to exploiting the vulnerability of supervisory software, the attacker can make the GUI of the supervisory software display incorrect data, presenting a false picture to the operators [18, 22, 29]. Besides, the attacker can hijack the process that supervisory software loads DLLs, then the malicious codes can be executed with the same privileges as the application, which is reported by ICS-CERT [19]. To defend against these attacks, in addition to efforts in security management such as adding a firewall, and improving personnel security awareness, the best alternative solution is to find and fix the bugs. Our work is an attempt at this kind of approach.

CONCLUSION

We present the first fuzzing framework *ICS³Fuzzer* towards supervisory software, which is customizable to support detecting protocol implementation vulnerabilities in different supervisory software

from various vendors. Given a functionality of supervisory software, we can build a communication template to simulate the session based on captured messages. With the help of automated synchronization of network and GUI behavior, *ICS*^{3}*Fuzzer* can reach any input state and feed mutated inputs directionally. By conducting experiments in 4 different commercial supervisory software, we successfully identified 13 vulnerabilities.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their comments on an earlier version of this paper and our shepherd, Dr Berkay Celik, for helping us improve the presentation of the final version. This work is supported in part by Guangdong Province Key Area R&D Program of China under Grant No. 2019B010137004. and Key Program of National Natural Science Foundation of China under Grant No. U1766215.

REFERENCES

[1] Humberto J Abdelnur, Radu State, and Olivier Festor. 2007. KiF: a stateful SIP fuzzer. In *Proceedings of the 1st international conference on Principles, systems and applications of IP telecommunications*. 47–56.
[2] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *International Conference on Information Security*. Springer, 343–358.
[3] Jonathan Bennett. 2020. AutoIt Script Home Page. https://www.autoitscript.com/site/
[4] Eli Biham, Sara Bitan, Aviad Carmel, Alon Dankner, Uriel Malin, and Avishai Wool. 2019. Rogue7: Rogue engineering-station attacks on S7 Simatic PLCs. *Black Hat USA* (2019).
[5] Georges Bossert, Frédéric Guihéry, and Guillaume Hiet. 2014. Towards automated protocol reverse engineering using semantic information. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*. 51–62.
[6] Derek Bruening, Qin Zhao, and Saman Amarasinghe. 2012. Transparent dynamic instrumentation. In *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*. 133–144.
[7] Juan Caballero, Pongsin Poosankam, Christian Kreibich, and Dawn Song. 2009. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proceedings of the 16th ACM conference on Computer and communications security*. 621–634.
[8] Defense Use Case. 2016. Analysis of the cyber attack on the Ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)* 388 (2016).
[9] Weidong Cui, Jayanthkumar Kannan, and Helen J Wang. 2007. Discoverer: Automatic Protocol Reverse Engineering from Network Traces.. In *USENIX Security Symposium*. 1–14.
[10] George Denton, Filip Karpisek, Frank Breitinger, and Ibrahim Baggili. 2017. Leveraging the SRTP protocol for over-the-network memory acquisition of a GE Fanuc Series 90-30. *Digital Investigation* 22 (2017), S26–S38.
[11] Mohamed Endi, YZ Elhalwagy, et al. 2010. Three-layer PLC/SCADA system architecture in process automation and data monitoring. In *2010 the 2nd international conference on computer and automation engineering (iccae)*, Vol. 2. IEEE, 774–779.
[12] Nicolas Falliere, Liam O Murchu, and Eric Chien. 2011. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response* 5, 6 (2011), 29.
[13] Davide Fauri, Bart de Wijs, Jerry den Hartog, Elisa Costante, Emmanuele Zambon, and Sandro Etalle. 2017. Encryption in ICS networks: A blessing or a curse?. In *2017 IEEE International Conference on Smart Grid Communications (SmartGridComm)*. IEEE, 289–294.
[14] Ivan Fratric. 2017. WinAFL: A fork of AFL for fuzzing Windows binaries. https://github.com/googleprojectzero/winafl.
[15] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.
[16] A. Ginter. 2017. The top 20 cyber attacks against industrial control systems. White Paper, Waterfall Security Solutions.
[17] Tianxiao Gu, Chengnian Sun, Xiaoxing Ma, Chun Cao, Chang Xu, Yuan Yao, Qirun Zhang, Jian Lu, and Zhendong Su. 2019. Practical GUI testing of Android applications via model abstraction and refinement. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 269–280.
[18] John T Hagen and Barry E Mullins. 2013. TCP veto: A novel network attack and its Application to SCADA protocols. In *2013 IEEE PES Innovative Smart Grid Technologies Conference (ISGT)*. IEEE, 1–6.
[19] ICS-CERT. 2015. Cimon CmnView DLL Hijacking Vulnerability. https://us-cert.cisa.gov/ics/advisories/ICSA-15-069-01
[20] Rob Caldwell Josh Homan, Sean McBride. 2016. IRONGATE ICS Malware: Nothing to See Here...Masking Malicious Activity on SCADA Systems. https://www.fireeye.com/blog/threat-research/2016/06/irongate_ics_malware.html
[21] Jinho Jung, Stephen Tong, Hong Hu, Jungwon Lim, Yonghwi Jin, and Taesoo Kim. 2021. WINNIE: Fuzzing Windows Applications with Harness Synthesis and Fast Cloning. In *Proceedings of the 2021 Annual Network and Distributed System Security Symposium (NDSS), Virtual*.
[22] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. CLIK on PLCs! Attacking control logic with decompilation and virtual PLC. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*.
[23] Tammo Krueger, Hugo Gascon, Nicole Krämer, and Konrad Rieck. 2012. Learning stateful models for network honeypots. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*. 37–48.
[24] Mohit Kumar. 2018. TSMC Chip Maker Blames WannaCry Malware for Production Halt. *The Hacker News* (2018).
[25] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.
[26] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS protocol fuzzing: Coverage guided packet crack and generation. In *Design Automation Conference (DAC)*. York.
[27] Rui Ma, Daguang Wang, Changzhen Hu, Wendong Ji, and Jingfeng Xue. 2016. Test data generation for stateful network protocol fuzzing using a rule-based state machine. *Tsinghua Science and Technology* 21, 3 (2016), 352–360.
[28] Valentin Jean Marie Manès, HyungSeok Han, Choongwoo Han, Sang Kil Cha, Manuel Egele, Edward J Schwartz, and Maverick Woo. 2019. The art, science, and engineering of fuzzing: A survey. *IEEE Transactions on Software Engineering* (2019).
[29] Thomas Morris and Wei Gao. 2013. Industrial Control System Cyber Attacks. https://doi.org/10.14236/ewic/ICSCSR2013.3
[30] Trent Nelson and May Chaffin. 2011. Common cybersecurity vulnerabilities in industrial control systems. *Control systems security program* (2011).
[31] James Newsome, David Brumley, Jason Franklin, and Dawn Song. 2006. Replayer: Automatic protocol replay by binary analysis. In *Proceedings of the 13th ACM conference on Computer and communications security*. 311–321.
[32] Matthias Niedermaier, Florian Fischer, and Alexander von Bodisco. 2017. PropFuzz—An IT-security fuzzing framework for proprietary ICS protocols. In *2017 International Conference on Applied Electronics (AE)*. IEEE, 1–4.
[33] NVD. 2021. CVE-2021-20587 Detail. https://nvd.nist.gov/vuln/detail/CVE-2021-20587
[34] Joshua Pereyda. 2017. boofuzz: Network protocol fuzzing for humans. *Accessed: Feb* 17 (2017).
[35] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.
[36] Saranyan Senthivel, Shrey Dhungana, Hyunguk Yoo, Irfan Ahmed, and Vassil Roussev. 2018. Denial of engineering operations attacks in industrial control systems. In *Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy*. 319–329.
[37] Keith Stouffer, Joe Falco, and Karen Scarfone. 2011. Guide to industrial control systems (ICS) security. *NIST special publication* 800, 82 (2011), 16–16.
[38] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.
[39] Kuniyasu Suzaki, Toshiki Yagi, Akira Tanaka, Yutaka Oiwa, and Etsuya Shibayama. 2014. Rollback mechanism of nested virtual machines for protocol fuzz testing. In *Proceedings of the 29th Annual ACM Symposium on Applied Computing*. 1484–1491.
[40] Gilbert Wondracek, Paolo Milani Comparetti, Christopher Kruegel, Engin Kirda, and Scuola Superiore S Anna. 2008. Automatic Network Protocol Analysis.. In *NDSS*, Vol. 8. Citeseer, 1–14.
[41] Wurldtech. 2021. Achilles Test Platform. Accessed: Sep, 2021 https://www.ge.com/digital/sites/default/files/download_assets/achilles_test_platform.pdf.
[42] Yapeng Ye, Zhuo Zhang, F. Wang, X. Zhang, and D. Xu. 2021. NetPlier: Probabilistic Network Protocol Reverse Engineering from Message Traces. In *NDSS*.
[43] Michal Zalewski. 2014. American fuzzy loop. http://lcamtuf.coredump.cx/afl/