StructuredFuzzer: Fuzzing Structured Text-Based Control Logic Applications

Koffi Anderson Koffi 1, Vyron Kampourakis 2, Jia Song 1, Constantinos Kolias 1,* and Robert C. Ivans 3

1 Department of Computer Science, University of Idaho, 1776 Science Center Dr suite 304, Idaho Falls, ID 83402, USA; kkoffi@uidaho.edu (K.A.K.); jsong@uidaho.edu (J.S.)
2 Department of Information Security and Communication Technology, Norwegian University of Science and Technology, 2802 Gjøvik, Norway; vyron.kampourakis@ntnu.no
3 Idaho National Laboratory, Idaho Falls, ID 83415, USA; robert.ivans@inl.gov
* Correspondence: kolias@uidaho.edu

Abstract: Rigorous testing methods are essential for ensuring the security and reliability of industrial controller software. Fuzzing, a technique that automatically discovers software bugs, has also proven effective in finding software vulnerabilities. Unsurprisingly, fuzzing has been applied to a wide range of platforms, including programmable logic controllers (PLCs). However, current approaches, such as coverage-guided evolutionary fuzzing implemented in the popular fuzzer American Fuzzy Lop Plus Plus (AFL++), are often inadequate for finding logical errors and bugs in PLC control logic applications. They primarily target generic programming languages like C/C++, Java, and Python, and do not consider the unique characteristics and behaviors of PLCs, which are often programmed using specialized programming languages like Structured Text (ST). Furthermore, these fuzzers are ill suited to deal with complex input structures encapsulated in ST, as they are not specifically designed to generate appropriate input sequences. This renders the application of traditional fuzzing techniques less efficient on these platforms. To address this issue, this paper presents a fuzzing framework designed explicitly for PLC software to discover logic bugs in applications written in ST specified by the IEC 61131-3 standard. The proposed framework incorporates a custom-tailored PLC runtime and a fuzzer designed for the purpose. We demonstrate its effectiveness by fuzzing a collection of ST programs that were crafted for evaluation purposes. We compare the performance against a popular fuzzer, namely, AFL++. The proposed fuzzing framework demonstrated its capabilities in our experiments, successfully detecting logic bugs in the tested PLC control logic applications written in ST. On average, it was at least 83 times faster than AFL++, and in certain cases, for example, it was more than 23,000 times faster.

Keywords: PLC; fuzzing; structured text; AFL; AFL++; ICS

1. Introduction

Industrial control systems (ICSs) include special devices such as supervisory control and data acquisition (SCADA) systems, distributed control systems (DCSs), and programmable logic controllers (PLCs) [1], and are designed to monitor and control processes in critical infrastructures such as power plants, water treatment facilities, and oil refineries [2]. Focusing on PLC devices as enablers of operational technology (OT) [1], these devices were designed to support the automation of many processes in ICS with programs written in languages such as Structured Text (ST), ladder logic (LL), Sequential Function Chart (SFC), and Function Block Diagram (FBD) [3]. However, these programming languages are not widely used outside the ICS domain. As a result, the critical process of secure programming is often overlooked, leaving such critical devices vulnerable to logical errors. This, in turn, can lead to disastrous consequences. For example, in 2015, the Ukrainian power grid was attacked by the advanced persistent threat (APT) group known...
as Sandworm, using malware to shut down power to 225,000 customers [4]. In 2017, the Triton malware was used to attack a petrochemical plant in Saudi Arabia [5]. Again, in 2021, the US Cybersecurity and Infrastructure Security Agency (CISA) issued an alert warning regarding the fact that some attackers had gained remote access to a water treatment facility in Florida and attempted to poison the water supply [6].

The industry responded to such threats by increasing the security against external threats targeting PLCs [7]. However, ensuring the security of PLCs is not solely about defending against external attacks; it also involves rigorous testing down to the application level, i.e., the control logic, to prevent internal errors leading to attacks or catastrophic failures. In this direction, developing and implementing advanced fuzzing techniques for PLCs, which can uncover security vulnerabilities and logical errors, is an essential step to enhance the overall security capacity of ICSs.

Fuzzing, or fuzz testing, is an automated software testing technique that provides unexpected or invalid input to a target program to reveal some unintended behavior [8]. Put simply, the primary objective of a fuzzer is to uncover bugs and other errors in software, including security vulnerabilities. Fuzzing has been widely used to find vulnerabilities in various software programs, including web browsers, operating systems, and network protocols [9]. Fuzzing techniques may prove vitally important for enhancing cybersecurity in ICS. An effective fuzzing approach for ICS aids in identifying and rectifying vulnerabilities, thus contributing to the overall fortification of critical infrastructure and ensuring operational continuity in numerous industries. Although conventional fuzzing techniques have yielded significant results for common software systems, their effectiveness is substantially reduced when applied to PLCs due to their distinct and complex architecture. In fact, the application of fuzzing to PLCs is still in its infancy, mainly due to the unique challenges posed by the specialized nature of ICS programming languages like ST, with the need to adapt the fuzz inputs to their expected input structures during fuzzing while ensuring the safety of the physical process. ST is not foreign to these challenges, as it is one of the most widely used standardized ICS programming languages. In view of this, fuzzing techniques applied to PLCs appear extremely constrained. Actually, the majority of the existing research is focused on the communication protocols [10] and the I/O modules. Fuzzing of the control logic applications has been largely ignored, with the handful of works in this area being limited to specific PLC technologies such as Codesys [3]. These challenges are discussed further in the challenges subsection of our methodology section (Section 3.1).

This paper introduces a novel framework for fuzzing PLC control logic applications to address this shortcoming. Unlike the existing body of work that focuses on industrial protocols [10–15], our approach is optimized for fuzzing ST-based control logic applications. The proposed framework offers a general-purpose solution to fuzzing internal logic flaws, indirectly enhancing the reliability and security of ICS devices against external cyber threats. Among ICS programming languages, the proposed framework is designed for fuzzing applications written in ST, which is widely used in PLCs. The primary aim of the proposed framework is to detect logical bugs in ST programs that could potentially lead to security vulnerabilities. Although not all bugs may directly result in security vulnerabilities, the detection of these bugs is crucial for preventing potential security threats. The proposed framework is designed to detect these bugs and enhance the security and reliability of ICS devices. The framework includes a PLC runtime and a custom fuzzer that automatically generates inputs to fuzz control logic applications written in ST. The main novelty of the proposed framework lies in integrating the PLC runtime with a custom fuzzer designed to detect logical bugs that violate a correctness specification regarding the inputs and outputs of ST programs. The contributions of this paper are as follows:

- A fuzzing framework for PLC control logic applications written in ST specified by the IEC 61131-3 [16] standard. It is worth noting that the IEC 61131-3 standard is widely recognized in the industry and defines the syntax and semantics for textual and graphical programming languages for PLCs such as ST and LL or Ladder Diagram (LD). This standardization ensures that the programs are portable across different
PLC vendors. Our work, therefore, extends fuzzing to a broad range of real-world PLC applications. The novelty lies in the framework’s ability to guide the fuzzing process to fuzz internal logical flaws, thus potentially augmenting the robustness of ICS devices against external threats. This has been found crucial since ST is a widely utilized language in PLC programming, yet effective fuzzing tools specifically designed and optimized for such languages have been lacking. This framework can be a valuable tool for detecting bugs and verifying the correctness of ST programs based on a specification.

- A custom PLC runtime is integrated into the fuzzing process. This is unique, as it provides contextual execution information for ST applications, permitting a more targeted and efficient fuzzing process. This substantially improves upon conventional techniques in which the runtime and fuzzing components frequently operate in isolation.
- A custom fuzzer for effective input generation is explicitly designed to generate inputs for fuzzing ST-based control logic applications. This fuzzer’s utility stems from its competence in generating diverse inputs, considering the program’s context, thereby enhancing the chance of uncovering vulnerabilities that might otherwise be overlooked by a generic fuzzer.
- An evaluation of the efficacy of our framework is performed using a collection of carefully crafted ST programs. This rigorous evaluation reinforces confidence in our proposed framework’s utility and presents a benchmark for future research in this area.

The remainder of this paper is organized as follows. Section 2 provides background information on PLCs, fuzzing, and some motivating examples. Section 3 describes the proposed fuzzing framework. Section 4 presents the experiments and results. Finally, Section 5 presents the related work, and Section 6 concludes the paper.

2. Background

In this section, we provide background information on PLCs and fuzzing, along with two examples that indicate the need for a novel fuzzing framework optimized for PLC control logic applications.

2.1. Programmable Logic Controllers

PLCs are specialized industrial devices implementing solid-state control systems with a user-programmable memory containing instructions for performing specific functions, such as I/O control, logic, timing, counting, and arithmetic [17]. They are specialized computers that are geared toward automating various manufacturing processes, utilities, and infrastructure systems. PLCs are designed to perform control functions within harsh industrial environments while withstanding extreme temperatures, vibrations, and noise conditions, among others. In other words, PLCs are tasked with the real-time control and monitoring of machinery and processes, making them indispensable components in various sectors, including automotive manufacturing, water treatment facilities, oil refineries, and power plants [3,5,18,19]. Given the crucial role of PLCs in critical infrastructures, their security and reliability are decisive. Historically, PLCs were designed to focus on safety and control, operating in isolated environments. However, the integration of PLCs with modern IT technologies, including the Internet of Things (IoT), Artificial Intelligence (AI), and cloud computing and analytics, has increased their exposure to a plethora of cyber threats. For instance, the Stuxnet worm attack in 2010 [20] highlighted the vulnerability of PLCs to sophisticated cyberattacks. It demonstrated that PLCs could be targeted by malware designed to reprogram them and, therefore, disrupt sensitive industrial processes. Echoing the significance of PLCs in various essential services, the high degree of their reliability stems from their deterministic nature, executing control logic in a predictable and cyclical manner according to a user-defined run cycle. This deterministic execution is critical for the real-time response required in automation tasks.
At their core, PLCs are governed by user-created control logic applications, often called ladder logic or control logic, written in domain-specific languages defined by the IEC 61131-3 standard [16]. The most commonly used PLC programming languages include:

- **Structured Text (ST)**: A high-level, block-structured language resembling Pascal or C, used for complex tasks that may be cumbersome to implement with graphical languages.
- **Ladder logic (LL)**: A graphical language representing control logic in a form that emulates electrical relay logic, making it intuitive for electricians and technicians.
- **Sequential Function Chart (SFC)**: This graphical programming language is used for designing sequential control systems and complex program structures with multiple distinct states and transitions.
- **Function Block Diagram (FBD)**: Another graphical language that represents functions between input and output variables using blocks connected by lines, similar to electronic circuit diagrams.

2.2. Fuzzing

Fuzzing is a technique for automatically discovering vulnerabilities in software by submitting some arbitrary combination of inputs to the test target to reveal how it responds [8,21]. Historically, first introduced by Barton Miller [22], fuzzing has since been used extensively and in diverse ways towards identifying various vulnerabilities in numerous kinds of software [9]. Based on the degree of understanding of the target program, fuzzers can be classified as black-box [23], grey-box [24], and white-box [25], with each technique being sequential, having more information available that can be leveraged for the analysis of the test target. Otherwise, fuzzers can also be classified, based on the type of input they create, into mutation based [26], generation based [12,27], and hybrid [28]. The first category mutates existing inputs to create new inputs, while the second generates new inputs from scratch from a specification, often based on a specific grammar. Hybrid fuzzers, on the other hand, combine both mutation and generation-based fuzzing techniques. Another categorization is according to the type of the adopted optimization criteria that the fuzzer utilizes. To this end, coverage-based [29] fuzzers use code coverage information to guide the generation of new inputs, while feedback-driven [30] fuzzers use runtime feedback, such as feedback from dynamic binary instrumentation, to guide the generation of new inputs. Finally, evolutionary [31] fuzzers use evolutionary algorithms to guide the creation of new inputs, considering the efficacy of the previous test cases while generating the new test cases. In this paper, we consider mutation, generation, and coverage-based fuzzing practices. These fuzzing techniques provide state-of-the-art performance [32]. In particular, we consider the American Fuzzy Lop (AFL) fuzzer [33], a coverage and mutation-based fuzzer, and specifically the AFL++ variant [34], which incorporates several improvements towards identifying vulnerabilities. It has been utilized to find critical bugs in many open-source programs, including the Linux kernel.

2.3. Motivating Examples

Logical errors or bugs in control logic applications can lead to catastrophic failures with far-reaching consequences. These errors are oftentimes the root causes of security incidents and system malfunctions in critical infrastructures. However, traditional testing approaches, including fuzzing, face challenges in detecting such errors, mainly due to real-time constraints and physical process interactions that are the common denominator among all control logic applications. Below are two of the most common scenarios that can lead to bugs in control logic applications.

One prevalent type of bug in control logic applications occurs when the input values are later subjected to incorrect arithmetic operations [35]. These wrong values are subsequently used in conditional statements, leading to unexpected behaviors that are often difficult to detect or prevent in practice.
Consider the example of a safety-critical automated temperature control system in a power plant, illustrated in Listing 1. The program aims to regulate the reactor’s temperature for safe and efficient operation. A logical bug, marked with ① on line 14, involves incorrect arithmetic operations that affect the branch conditions on lines 16, 19, and 22, annotated with ②, ③, and ④, respectively. This bug poses potential risks during the plant’s operations.

**Listing 1.** Example of a logical bug in a control logic application for an automated temperature control system.

```
PROGRAM AutomatedTemperatureControl
VAR
  currentTemperature AT %ID0 : REAL; (* Current temperature in the reactor (in Celsius) *)
  desiredTemperature AT %ID1 : REAL; (* Desired operational temperature (in Celsius) *)
  coolingSystemActivated AT %QX0.0 : BOOL; (* State of the cooling system *)
  heatingSystemActivated AT %QX0.1 : BOOL; (* State of the heating system *)
  alertRaised AT %QX0.2 : BOOL; (* Flag for raising an alert *)
END_VAR
VAR
  temperatureDifference : REAL; (* Difference between current and desired temperature *)
END_VAR

temperatureDifference := currentTemperature - desiredTemperature; (*① Logical Bug: Incorrect arithmetic operations in temperature control logic *)
IF temperatureDifference > 10 THEN (*② Bug: missing absolute value operator can lead to the temperature going below the desired level by more than 10 degrees *)
  coolingSystemActivated := TRUE;
  heatingSystemActivated := FALSE;
ELSIF temperatureDifference > 0 THEN (*③ Bug: similar to ② (missing absolute value) *)
  coolingSystemActivated := FALSE;
  heatingSystemActivated := TRUE;
ELSE (*④ Bug: Deactivating both systems when the temperature is within the threshold *)
  coolingSystemActivated := FALSE;
  heatingSystemActivated := FALSE;
END_IF;

(* Raise an alert if the temperature deviates significantly from the desired level *)
IF ABS_REAL(temperatureDifference) > 20 THEN
  alertRaised := TRUE;
ELSE
  alertRaised := FALSE;
END_IF;
END_PROGRAM
```

More specifically, the principal error involves neglecting to apply the absolute value operation (ABS_REAL) in the evaluation of the temperature difference used in the condition used on line 16, noted with ②, leading to both cooling and heating systems deactivating when not intended on lines 21 and 22. Figure 1 illustrates the Control Flow Graph (CFG) of the program and the path taken by alternative inputs, including ones that trigger the bug. In the example, the desired temperature is 500 °C. When the current temperature is 520 °C, the temperature difference is 20 °C, which is within the acceptable range. Similarly, a current temperature of 505 °C leads to a temperature difference of 5 °C still within the valid range. However, when the current temperature is 480 °C, the difference is −20 °C outside the acceptable range. The correct approach would use the absolute value to ascertain the true magnitude of deviation from the desired temperature, ensuring that temperature calibration is applied as necessary to keep it within the acceptable range. Additionally, either the cooling or heating systems should remain active, ideally, to maintain the desired temperature. However, the control logic deactivates both systems when the temperature is within the allowed threshold. A more prudent implementation would deactivate the cooling system when the temperature is above the threshold and activate the heating system when it is below to maintain the desired temperature. Overall, such a mistake in
logic could lead to temperature fluctuations within the reactor, potentially impacting the safety and efficiency of the power plant operation.

Figure 1. Illustrative CFG of the automated temperature control example.

Another common source of control logic bugs is when the program’s outputs are assigned erroneous values. Ideally, values of certain variables should be within acceptable ranges according to the specifications of the particular application. However, this cannot always be trivially enforced in the code during the declaration or the assignment of a variable, thus allowing typos to occur during the development time and subsequently evolve into bugs during runtime.

For example, variables corresponding to actuator states are typically defined so as REAL to achieve maximum precision. However, nothing prohibits the developer from explicitly assigning the variable with a negative value by accident. This, in theory, is still valid because negative values fall within the range of real numbers, and no alert will be triggered during development time. However, in certain applications, negative values may be incorrect.

Listing 2 is an example of a control application of a nuclear power plant’s automated reactor control system. The system regulates the coolant flow rate and control rod position within a reactor to ensure safe and efficient operation. In particular, based on the reactor mode of operation, described by the variable reactorMode, the statement on line 10 redirects the control flow of the program to define appropriate actions, e.g., regarding the coolant flow and the positioning of the control rods. In this example program, there are two bugs involving the incorrect adjustments of the control rod position and coolant flow rate, marked 1 and 2, which could lead to operational risks. The first bug, marked 1, involves the control rods not being fully inserted during shutdown mode as shown in the assignment statement on line 20 in the code. The second bug, marked 2, is that the coolant flow rate is not maximized during the emergency mode as seen with the assignment statement on line 23 in the code. As an example, correct values for the control rod position and coolant flow rate for the shutdown and emergency modes, respectively, are 100% and 50.0 L per second. In the context of this example, these values can be considered safe and optimal for the reactor’s operation during these modes.

Both the aforementioned examples demonstrate the need for rigorous testing methods to ensure the security and reliability of control logic applications. In this respect, we present a framework, dubbed StructuredFuzzer, specifically designed to fuzz PLC control logic. A PLC runtime and fuzzer are also contributed to the framework. We demonstrate the effectiveness of our approach by fuzzing a collection of carefully designed ST programs of variable complexity using our custom-tailored fuzzer developed for the purpose. To compare with existing fuzzers, we also fuzzed the same set of ST programs using AFL++ fuzzer.
Listing 2. Example of a logical bug in a control logic application for an automated reactor control system.

```plaintext
PROGRAM AutomatedReactorControl
VAR
  reactorMode AT %IW0 : INT := 0; (* Operational mode of the reactor (1 for startup, 2 for normal, 3 for shutdown, 4 for emergency) *)
  coolantFlowRate AT %QD0 : REAL; (* Coolant flow rate (in liters per second) *)
  controlRodPosition AT %QD1 : REAL; (* Position of control rods (0% to 100%, where 100% is fully inserted) *)
  alarmActivated AT %QX0.0 : BOOL; (* State of the alarm system *)
END_VAR

(* Logical Bugs: Inappropriate control rod and coolant flow adjustments for some reactor modes *)
CASE reactorMode OF
  1: (* Startup mode *)
    coolantFlowRate := 30.0; (* Moderate coolant flow *)
    controlRodPosition := 20.0; (* Partially withdrawn control rods *)
  2: (* Normal operation mode *)
    coolantFlowRate := 50.0; (* High coolant flow *)
    controlRodPosition := 50.0; (* Halfway inserted control rods *)
  3: (* Shutdown mode *)
    coolantFlowRate := 20.0; (* Low coolant flow *)
    controlRodPosition := 50.0; (* Bug: Incorrectly halfway inserted control rods *)
  4: (* Emergency mode *)
    coolantFlowRate := 30.0; (* Insufficient coolant flow for emergency *)
    controlRodPosition := 100.0; (* Fully inserted control rods *)
    alarmActivated := TRUE;
ELSE
  (* Default safe state in case of an unrecognized mode *)
  alarmActivated := TRUE;
  coolantFlowRate := 0.0;
  controlRodPosition := 100.0; (* Fully insert control rods *)
END_CASE;
IF reactorMode <> 4 THEN
  alarmActivated := FALSE;
END_IF;
END_PROGRAM
```

3. Proposed Fuzzing Framework

In this section, we present challenges in fuzzing PLC applications and propose a fuzzing framework for addressing them.

3.1. Challenges in Fuzzing PLCs Control Applications

In this section, we elaborate on the most prominent challenges when it comes to fuzzing PLC control applications written in ST. More specifically, we have the following.

PLC Programming Languages and Runtime: Despite its potential, fuzzing has been under-utilized in the ICS domain, mainly due to the unique challenges posed by the specialized nature of ICS programming languages and the real-time constraints under which these systems operate. PLC control binaries are compiled into specific formats for each vendor, which makes it tedious to apply a general analysis technique to secure them. The lack of standardization of the PLC runtime environments greatly exacerbates this issue. Specifically, the runtime is the software environment, running as a process of the PLC operating system (OS) in which the control logic is executed. It is responsible not only for executing the applications but also for interfacing with the physical I/O modules. Therefore, the lack of standardization of the runtime environments makes it challenging to develop a general-purpose fuzzer for control logic applications. Additionally, the runtime is itself often proprietary and closed-source, rendering the development of a fuzzer that can work across different PLC vendors even more difficult. Since the PLC runtime is often tightly integrated with the I/O modules, dissociating and fuzzing each one independently is quite cumbersome. Finally, the runtime is often optimized for real-time performance,
which makes it nearly impossible to fuzz the control logic without a degradation in the performance of the PLC [3].

**Complex Input Structures:** Unlike traditional programming languages like C, ST requires explicit memory and I/O addressing during variable declaration as shown in Figure 2. Nevertheless, memory addressing is specific to the hardware of the target device in ST, in contrast to C, which is generally portable.

![Figure 2. Comparison of variable declarations in ST and C programming languages. Notice the presence of an explicit memory address that makes the declaration in ST non-trivial.](image)

Moreover, conventional fuzzers like AFL++ are not configured to understand structure in values assigned to variables. In most cases, that is not an issue in C-like programs. However, in ST programs, altering the value of a variable randomly permits the alteration of the memory address portion, which in turn may lead to invalid memory locations for the target hardware, with high probability as illustrated in Figure 3. These generated values are downright rejected by the runtime. Although valid values may be eventually generated and utilized by the fuzzer, a significant amount of time is wasted in producing meaningless values. Bearing this in mind, it is rather clear that conventional fuzzers are inefficient when it comes to ST programs.

![Figure 3. ST Input structure and example of mutated inputs generated by AFL++ and our fuzzer for the automated temperature control ST program. The mutated parts of the inputs are highlighted in blue. Notice how AFL++ is free to mutate the address part of the inputs, while our fuzzer is restricted to mutating the actual values.](image)

Nevertheless, it is possible to compile ST programs directly (i.e., without a runtime) to the C equivalent binary to avoid this issue. However, in this case, the behavior of the resulting binary might not be close to the actual behavior of the control application. This is because, without a PLC runtime, the resulting ST binary is not bound by the PLC run cycle and the I/Os and their response times. Consequently, many types of bugs might be missed, e.g., bugs related to the interaction with the physical environment. We address these issues with a custom input generation technique that utilizes static code analysis on
the ST program to generate fuzz inputs that conform to the input structure expected by the ST binary.

**Physical Process Interactions and Safety Concerns:** PLC roles typically involve perceiving changes in the physical environment and controlling or influencing them. This means that ST applications frequently interact with physical processes, which directly affects the status of machinery. Generally, during the fuzzing process, valid inputs are extensively altered, potentially leading to software crashes of the test target. However, in cases where the target interacts with a physical process, such inputs may have calamitous consequences. Therefore, to maintain the system’s safety and reliability, it is critical to identify these issues in advance, possibly even offline, without interacting with the environment.

In this regard, the suggested framework allows users to define beforehand what constitutes correct behavior through a mechanism known as a **fuzzing harness**, which is subsequently utilized by the fuzzer to identify harmful side effects and logical errors. Note that this is performed in a virtual setting that is decoupled from hardware components. Unlike physical environments where hardware components are tightly integrated and often depend on each other’s outputs, our framework enables the independent examination of different system components and the generation of arbitrary inputs as needed. This capability supports concurrent fuzzing as shown in Figure 4. Note that fuzzing the ST binary using our framework is possible in both real and controlled environments, as well as in virtual settings.

![Figure 4](image.png)

*Figure 4. Fuzzing the automated temperature control ST program in a virtual environment enables parallel fuzzing compared to a physical/controlled environment. Note that both the heating and cooling systems can be fuzzed independently.*

In this context, the proposed PLC runtime provides a safe environment for fuzzing, allowing the user to test control logic applications without risking any actual system. Furthermore, the proposed runtime simulates a generalized PLC environment, enabling any fuzzer to interact with the control logic applications as if they were running on a real PLC. Equally important, it can also be modified or extended to offer a more faithful simulation of the physical environment, such as the behavior of I/O modules and sensors.

The choice of focusing on ST was driven by several reasons. First, ST, specified by the IEC 61131-3 standard, is a widely used language for PLC applications and shares many similarities with high-level languages like C, which simplifies our development process and enhances the general understanding and applicability of our method [16]. Secondly, while non-textual languages such as LD are commonly utilized in ICS, they also harbor additional complexities not present in ST, primarily due to their inherent graphical nature. In the case of LD, its visual component could potentially complicate the fuzzing process and pose
challenges in effectively crafting fuzzed inputs. Notwithstanding, we fully acknowledge the relevance of LD and other IEC 61131-3 specified languages in industry, and we see the extension of our proposed framework to other ICS languages, including LD, as an important future work direction. Therefore, while our novel fuzzing framework currently focuses on ST, future work should enhance this platform’s versatility to cover additional languages, such as LD, used in PLC programming.

3.2. Proposed Framework Overview

We introduce StructuredFuzzer, a fuzzing framework that describes the effective implementation of ST-based PLC applications fuzzing, also bearing in mind potential remedies for the challenges discussed in Section 3.1. As illustrated in Figure 5, the framework is a four-step process comprising (1) the fuzzing harness specification, (2) the ST program compilation, (3) the fuzzing input corpus generation, and (4) the actual fuzzing of the PLC control logic.

Figure 5. The proposed fuzzing framework.

Step 1: Specifying a Fuzzing Harness (Optional): The first step involves creating a harness, i.e., a user-defined set of specifications regarding the correctness of the target ST programs. In particular, the harness should provide input restrictions that are considered valid when certain outputs are observed, or the system is in certain states. If these are violated, then it is assumed that it is due to a bug. Therefore, a code for signaling a fuzzer about this should exist. The latter can be achieved through code that aborts the execution of the program or emulates a crash. Thus, the harness can be implemented in C to express some violations of the desired program’s behavior (specifications) as assertion failures. Listing 3 provides an example of a harness for the automated temperature control system application. Precisely, it specifies invalid states or conditions (violations) as crashes that the fuzzer can detect, i.e., those that deviate from the specifications 1 to 4. The above-described step is optional, and it typically occurs when the fuzzing process is meant to be conducted offline and in the absence of a controlled environment.

The example harness, depicted in Listing 3, is a C program that signals to a fuzzer when to detect a crash using the abort() statements. It defines the desired behavior as specifications marked by 1 to 4. These conditions are translated into IF statements using the PLC I/O. This provides a flexible way to test for the correctness of the control logic and enforce secure behavior.

In this context, the runtime is simply a set of linked C programs that comprise (a) the C version of the ST program and (b) a set of functions for interfacing with the hardware components of the system, e.g., I/O and network modules. As depicted in Figure 6, when a harness is provided, the PLC runtime binary includes the default networking module and an I/O adapter module instead of the default I/O module, which enables the interaction with the physical I/Os. The I/O adapter replaces the interactions with the physical I/Os
with the interactions with the fuzzer. In this way, the fuzzer can provide inputs to the control logic and receive outputs from it.

### Listing 3. Example of a harness for the automated temperature control system application for a desired temperature of 500 °C.

```c
#include "iec_types.h"
#include "iec_std_lib.h"
#include <stdio.h>
#include <stdlib.h>

void config_init__(void);

void fuzzer_harness(void) {
    printf("[+] Current Temperature (ID0) = %f
", *__ID0);
    printf("[+] Desired Temperature (ID1) = %f
", *__ID1);
    printf("[+] Alert Raised (QX0.0) = %s
", __QX0_0 ? "TRUE" : "FALSE");

    // The desired temperature should be 500 degrees Celsius
    if (*__ID1 != 500) {
        printf("Test failed: Incorrect settings for desired temperature\n");
        abort();
    }

    // When cooling = TRUE and heating = FALSE, the current temperature should be
greater than 510, otherwise abort
    if (*__QX0_0 && ! *__QX0_1) {
        if (*__ID0 < 510) {
            printf("Test failed: Incorrect settings for heating and cooling systems\n");
            abort();
        }
    }

    // When cooling = FALSE and heating = TRUE, the current temperature should be
between 490 and 510, otherwise abort
    else if (! *__QX0_0 && __QX0_1) {
        if (*__ID0 >= 490 && *__ID0 <= 510) {
            printf("Test failed: Incorrect settings for heating and cooling systems\n");
            abort();
        }
    }

    // When cooling = FALSE and heating = FALSE, the current temperature should be 500
, otherwise abort
    else {
        if (*__ID0 != 500) {
            printf("Test failed: Incorrect settings for heating and cooling system\n");
            abort();
        }
    }
}
```

#### Step 2: ST Program Compilation

The second step of the framework depicted in Figure 5 is to transcode the ST programs to the corresponding PLC runtime. In our implementation, the compilation process is performed with the aid of Matiec `iec2c` compiler [36]. It is a source-to-source compiler that compiles ST programs to C. This compiler is used by many open-source projects such as OpenPLC [37].

Moreover, the harness can be omitted if the aim of the fuzzing process is not to find bugs in the control logic. When the harness is not provided, the program is fuzzed without any restrictions and executes as it would on actual hardware. In this particular case, the fuzzer is expected to find bugs mainly in the runtime. Unlike typical workflows, we also provide a wrapper for the compiler to consider the harness as an additional input. By doing so, assertions are injected into the produced binary. These cause the fuzzer to trigger
crashes for illegal conditions. Finally, the PLC runtime is responsible for executing the control logic, also capable of receiving inputs from the fuzzer.

**Figure 6.** Compilation process of the ST program to the PLC runtime.

**Step 3: Input Generation:** The third step involves generating the initial fuzzing inputs (seeds). This step is critical since using seeds from a previous fuzzing (i.e., one that was not provided with initial seeds) of the same program can be ineffective. The importance of good initial seeds for fuzzing has been demonstrated by prior studies [38,39]. For this reason, we developed a static code analysis (SCA) tool for generating good initial seeds. It utilizes the Tree-Sitter library to parse the ST program. Parsing the ST provides a more reliable way to perform various program analyses, unlike regular expressions, which are generally error-prone. Tree-Sitter [40] is a parser generator tool as well as an incremental parsing library previously used for similar purposes [41]. To parse the programs with Tree-Sitter, a grammar for the target language is required. However, there is a lack of extensive research on grammar specifically tailored for ST programming. To address this shortcoming, we construct a Tree-Sitter grammar for ST that enables us to build an abstract syntax tree (AST) from an ST source file. We utilize the AST in our analysis to extract the types and addresses of the variables used in the programs as shown in Algorithm 1. Importantly, this step is required to ensure the fuzzer generates inputs compatible with the types and addresses used within the ST program.

**Algorithm 1** Algorithm for automatic fuzzing harness generation.

| Require: | st_program ▷ ST program source file |
| Ensure: | c_input_adapter ▷ Source file for the input adapter program |
| 1: ast ← tree_sitter_parse(st_program) |
| 2: variables ← ∅ ▷ Initialize the list of variables in the ST program |
| 3: for node ∈ visit(ast.root_node) do |
| 4: if node.type = "variable_declaration" then |
| 5: variable_name ← get_node_value(node.children[1]) |
| 6: variable_type ← get_node_value(node.children[0]) |
| 7: variable_address ← get_node_value(node.children[2]) |
| 8: variables.append(variable_name, variable_type, variable_address) |
| 9: end if |
| 10: end for |
| 11: c_input_adapter ← generate_c_input_adapter(variables) |
| 12: return c_input_adapter |
During the variables extraction, the algorithmic procedure traverses the nodes of the AST from the root node to the leaves as depicted in Figure 7. If a node contains a variable declaration or definition, the variable name, type, and address are extracted into a set of PLC variables as described in Algorithm 1. These variables are subsequently fed to a function for generating the I/O adapter program (written in C) that makes them available to the fuzzer as depicted in Algorithm 2.

Algorithm 2 Function \( \text{generate}_c \_\text{input}\_\text{adapter} \) (Generate PLC Input Adapter Function).

Require: \( \text{variables} \) \( \triangleright \) A list of PLC input variables.
Ensure: \( \text{input}\_\text{adapter} \) \( \triangleright \) C source file of the input adapter program

\[
\begin{align*}
\text{function} & \quad \text{generate}_c \_\text{input}\_\text{adapter}(\text{variables}) \\
& \quad \text{input}\_\text{adapter} \leftarrow \text{c}\_\text{headers} \quad \triangleright \text{Initialize the C source file with the PLC runtime C headers file} \\
& \quad \text{set}_\text{plc}\_\text{input} \leftarrow \emptyset \quad \triangleright \text{Source of the function for setting PLC inputs from C} \\
& \quad \text{for} \text{ var} \in \text{variables} \text{ do} \\
& \quad \quad \text{cast\_type} \leftarrow \text{get\_type\_cast}(\text{var}\_\text{type}) \quad \triangleright \text{Casting from PLC to C variables} \\
& \quad \quad \text{append}((\text{set}_\text{plc}\_\text{input}, \text{var}\_\text{name}, \text{var}\_\text{address}, \text{cast\_type})) \quad \triangleright \text{Add PLC input to set} \\
& \quad \text{end for} \\
& \quad \text{input}\_\text{adapter} \leftarrow \text{format}(\text{set}_\text{plc}\_\text{input}) \quad \triangleright \text{Format to a C function file} \\
& \quad \text{return} \text{input}\_\text{adapter} \\
& \quad \text{end function}
\end{align*}
\]

The algorithm takes the list of variables for the generation of a C function. For each variable, the procedure determines its C equivalent type from the ST type. Consequently, according to line 6 of Algorithm 2, a piece of C code is added to the \( \text{set}_\text{plc}\_\text{input}() \) function for declaring and setting the value of the variable based on its name, address, and casting type. After adding all the variables of the ST program to the function, the C program is finalized, incorporating the I/O adapter program and including the necessary headers to make the function accessible from the PLC runtime.

Figure 7. Example of an AST generated by Tree-Sitter for the variable declaration of \texttt{reactorMode} in the automated reactor control program. Notice that Algorithm 1 would traverse each \texttt{variable\_declaration} node to extract the variable name, type, and address from its children.

Step 4: Fuzzing: The final step is fuzzing the control logic using the proposed PLC runtime along with a fuzzer. To do that, we implemented a custom fuzzer in Rust based on the \texttt{LibAFL} [42] fuzzing library. In further detail, LibAFL is a popular library that
provides a set of utilities for building custom fuzzers. It separates fuzzers into several modules, each providing core functionalities such as mutators, generators, feedbacks, observers, monitors, and executors. The LibAFL plugin-like system enables adding new components or reusing existing modules to construct fuzzers. To this extent, the proposed fuzzer reuses some core functionalities present in AFL++ but with a new custom mutator, namely, PLCRandomInputMutator, designed specifically for fuzzing control applications.

As depicted in Algorithm 3, the mutator is fed with inputs derived from the provided corpus, the current state of the fuzzer, and the index of the current fuzzing stage. At the end of the procedure, the mutated inputs and a new state of the fuzzer are expected to be output. In detail, first, a random number generator based on the state of the fuzzer is initialized. Next, the current inputs are parsed into a set of ST variable structures. If the list of variables is empty, the fuzzer state is set to Skipped, indicating a failed mutation; otherwise, it is set to Mutated to indicate a successful mutation. Note that for each variable in the set of PLC variables, the mutation procedure determines its size (in bits) for generating alternative inputs, using the random number generator from the fuzzer’s state to generate random values for the control logic inputs. Following, the fuzzer utilizes the mutation procedure (mutate) and the PLC runtime to fuzz the native control logic code. The execution of the program is monitored during fuzzing of the control logic until a STOP (e.g., SIGTERM or SIGKILL signals in Linux) signal is received. If the harness is provided during the compilation of the binary, the fuzzer would detect the crashes (e.g., signaled via an abort() statement). In the same vein, during fuzzing, the runtime binary would execute the code of the harness on each execution with the mutated inputs. Alternatively, a fuzzer such as AFL++ can be coupled with the proposed runtime to fuzz the control application.

**Algorithm 3 PLCRandomInputMutator (Random Input Generation for Fuzzing).**

```plaintext
Require: state \(\triangleright\) Fuzzer’s state
Require: input \(\triangleright\) Current input to fuzz
Require: stage_idx \(\triangleright\) Current fuzzing stage index
Ensure: mutated_inputs \(\triangleright\) Mutated PLC inputs
Ensure: new_state \(\triangleright\) New fuzzing state to indicate whether the mutation was applied

1: procedure MUTATE(state, input, stage_idx)
2: \hspace{1cm} mutated_inputs \(\leftarrow\) \emptyset \(\triangleright\) Initialize the mutated inputs
3: \hspace{1cm} rng \(\leftarrow\) init_rng(state) \(\triangleright\) Initialize the random number generator from state
4: \hspace{1cm} variables \(\leftarrow\) parse_input(inputs) \(\triangleright\) Get the C variables from ST
5: \hspace{1cm} if variables \(\in\) \emptyset then
6: \hspace{1cm} \hspace{1cm} new_state \(\leftarrow\) States.Skiped
7: \hspace{1cm} end if
8: \hspace{1cm} for var \(\in\) variables do
9: \hspace{1cm} \hspace{1cm} if var.type.size = 1 then \(\triangleright\) Variable size is 1 bit
10: \hspace{1cm} \hspace{1cm} \hspace{1cm} input_value \(\leftarrow\) !var.value \(\triangleright\) Flip boolean value
11: \hspace{1cm} \hspace{1cm} else if var.type.size = 8 then \(\triangleright\) Variable size is 8-bit (byte)
12: \hspace{1cm} \hspace{1cm} \hspace{1cm} input_value \(\leftarrow\) generate(rng, 8) \(\triangleright\) Generate random 8-bit
13: \hspace{1cm} \hspace{1cm} else if var.type.size = 16 then \(\triangleright\) Variable size is 16-bit (word)
14: \hspace{1cm} \hspace{1cm} \hspace{1cm} input_value \(\leftarrow\) generate(rng, 16) \(\triangleright\) Generate random 16-bit
15: \hspace{1cm} \hspace{1cm} else if var.type.size = 32 then \(\triangleright\) Variable size is 32-bit (double)
16: \hspace{1cm} \hspace{1cm} \hspace{1cm} input_value \(\leftarrow\) generate(rng, 32) \(\triangleright\) Generate random 32-bit
17: \hspace{1cm} \hspace{1cm} else if var.type.size = 64 then \(\triangleright\) Variable size is 64-bit (long)
18: \hspace{1cm} \hspace{1cm} \hspace{1cm} input_value \(\leftarrow\) generate(rng, 64) \(\triangleright\) Generate random 64-bit
19: \hspace{1cm} end if
20: \hspace{1cm} \hspace{1cm} mutated_inputs.append(input_value)
21: \hspace{1cm} \hspace{1cm} new_state \(\leftarrow\) States.Mutated
22: \hspace{1cm} end for
23: return new_state, mutated_inputs
24: end procedure
```
4. Experimental Evaluation

In this section, we present the experimental evaluation of the proposed framework, providing also a detailed discussion revolving around its key findings.

4.1. Experimental Setup

To evaluate our framework, we created a collection of ST programs that implement common PLC control logic application scenarios. Note that every program uses common programming structures, such as if statements, case statements, etc. The programs were created with incremental levels of complexity across several characteristics, including various branch depths and branch widths. More precisely, three of them exhibit increasing branch depth, three showcase increasing branch widths, and three demonstrate progressively complex branch conditions. Additionally, we designed 10 more complex programs resembling real-world PLC control logic. The list of programs and their descriptions are given in Table 1.

<table>
<thead>
<tr>
<th>Programs Characteristics</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth 1 to 3 Branch Depth</td>
<td>Programs with increasing branch depths</td>
</tr>
<tr>
<td>Width 1 to 3 Branch Width</td>
<td>Programs with increasing branch widths</td>
</tr>
<tr>
<td>Condition 1 to 3 Branch Condition</td>
<td>Programs with various branch conditions complexities</td>
</tr>
<tr>
<td>Complex 1 to 10 Real-world Program</td>
<td>Programs resembling real-world PLC programs</td>
</tr>
</tbody>
</table>

Recall that fuzzing incorporates random processes, often leading to different results in each run. To ensure the results of the evaluation are accurate, we fuzzed each test subject of Table 1 multiple times. More specifically, with the help of the HPC, we performed the fuzzing of the individual programs 20 times for 24 h, each on a node of a computing cluster. All executions were performed in the Falcon high-performance computing (HPC) cluster [43]. Each fuzzing task was given 4 GB of RAM and 4 CPUs. The fuzzers utilized for comparison purposes include our custom fuzzer and stock AFL++. One may speculate that the input generation process may be the main contributor to any speedup. To provide a more comprehensive answer, we also considered the case of AFL++ when utilizing the proposed input generation technique for creating its initial fuzzing input corpus as presented in the third step of the proposed framework (See Section 3.2). From now on, the latter test platform will be referred to as AFL++/IG. To ensure the results would be comparable, we applied filters to include only executions that lead to valid crashes (bugs). Overall, on average, 6 to 18 crashes were observed per program evaluation cycle.

4.2. Results and Discussion

We compared the results of the three fuzzers, AFL++, AFL++/IG, and our fuzzer, to quantify their crash-finding capabilities and efficiency based on the average time and number of executions required to find the logic bug. We also computed and reported the speedup achieved. We define speedup as the ratio of the average time taken by the target fuzzer (e.g., AFL++) to the average time taken by our custom-made fuzzer.

As a general remark, the results given in Table 2 show that the proposed fuzzer significantly outperforms both AFL++ and AFL++/IG. Based on the results, it is observed that (a) the proposed fuzzer can locate the bug in less than a second consistently for all test cases regardless of the complexity of their structure, (b) it usually finds the bug by at least two (frequently three and four) orders of magnitude faster than AFL++ and AFL++/IG. For example, the proposed fuzzer takes only 0.21 s to find the bug in the program with a branch width of 1 as opposed to 3954.17 and 3870.00 s, which AFL++ and AFL++/IG require, respectively. Similarly, while AFL++ and AFL++/IG take 1509.67 and 887.17 s to find the bug in a program with a branch depth of 3, the proposed fuzzer only needs 0.17 s.

Notably, a substantial improvement in terms of performance across virtually all the considered test cases is observed even in the case of the AFL++/IG fuzzer, where we simply
incorporate the proposed input generation into AFL++ to create the initial fuzzing input corpus. The performance increase in AFL++/IG is up by two orders of magnitude and a factor of two in the worst case when compared to traditional AFL++. With reference to Table 2, this performance rectification can be observed in all the tested programs except the ones of Width 3 and Depth 2, where AFL++/IG takes more time to find the bug compared to AFL++. For example, AFL++/IG requires 2298.20 and 427 s, while AFL++ needs 1114.47 and 358 s, respectively, in Width 3 and Depth 3. Especially for all the considered real-world programs, AFL++/IG always outperforms AFL++ by a factor of ×63 and ×18, on average.

Particularly, the results depicted in Figure 8 confirm the efficiency of the proposed fuzzer vis-a-vis the AFL++ and AFL++/IG fuzzers. In detail, the y-axis of the figures illustrates the time required to discover logical errors, while the x-axis indicates the varying complexity levels of the different tested ST programs. It is easy to observe that our fuzzer consistently exhibits superior performance compared to AFL++ and AFL++/IG as evidenced by the lower average times across all test cases. Notably, for all the considered real-world programs and those with various branch depths, widths, and condition test cases, our fuzzer demonstrates a distinctive cutback in the interquartile range (IQR).

Moreover, in the results with reference to Figure 8, only a few outliers were observed, affirming a consistent performance across different programs. Specifically, the outliers observed for the Depth and Width test cases (top and second figures of Figure 8) indicate that although our input generation consistently delivers strong performance—on average—when combined with AFL++, additional mutation strategies are required to improve the results on the edge cases. This gap in performance compared to our fuzzer once again underscores the importance of the custom mutator. Furthermore, the performance disparities observed in real-world programs suggest that, mainly due to the nature of the logic bugs in ST programs, a one-size-fits-all approach, like in the case of AFL++, may not be optimal. Our PLC runtime environment, designed specifically for fuzzing ST programs, enables a more precise and targeted bug discovery process through the use of a harness function, as it allows one to define program specifications used to detect violations during fuzzing. Through the experimental evaluation, we also highlight the constraints of AFL++ when applied to domain-specific languages like ST, underlining the need for specialized fuzzing tools for PLCs.

To fortify our findings, we further calculated the speedup of the proposed fuzzer against AFL++ and AFL++/IG. The results depicted in Table 2 demonstrate that the proposed fuzzer achieves a speedup ranging from a few ×100 to more than ×10,000 over AFL++ and AFL++/IG. Indicatively, the proposed fuzzer exhibits a speedup factor of ×8880.41 over AFL++ and a factor of ×5218.65 over AFL++/IG when considering the program with a branch depth of 3. For programs resembling real-world PLC control logic, the proposed fuzzer achieves a speedup of ×23,088.15 over AFL++ as opposed to ×365.69 over AFL++/IG in finding the bug in the program Complex 2.

Furthermore, Figure 9 corroborates a significant improvement in speed in terms of logic bug detection, showcasing a consistently superior speedup factor of the proposed fuzzer over the AFL++ and AFL++/IG ones. This speed discrepancy is even more apparent, particularly in programs with an increased level of the considered characteristics. This enhanced performance stems from our input generation scheme paired with our customized mutator mechanism because of the focused mutation to the PLC inputs and their adaption to the expected input structures within the ST programs. As a characteristic example, the substantial speedup observed for programs like Complex 2 and Complex 5 suggests that our fuzzer’s logic-oriented testing approach is particularly effective for programs that incorporate certain control flow statements, such as case and if statements. Moreover, integrating our input generation method with AFL++ demonstrates improved performance, albeit not as conspicuous as with our fuzzer. Despite the prevalence of AFL++/IG compared to AFL++, both are significantly slower than the proposed fuzzer. AFL++ and AFL++/IG often require several hours as opposed to less than a second to find the bug in all the considered programs. Based on these findings, the performance increase cannot solely be
attributed to the proposed input generation technique. These findings imply that while the initial fuzzing corpus generation can indeed enhance the performance of an existing fuzzer (AFL++), the primary driver of efficiency lies in our custom mutator detailed in the fourth step of the proposed framework (See Section 3.2).

Table 2. Average time in sec (lower is better) and speedup (higher is better) to find the crash for each fuzzer.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>AFL++</th>
<th>AFL++/IG</th>
<th>Our Fuzzer</th>
<th>AFL++ Fuzzer Speedup</th>
<th>AFL++/IG Fuzzer Speedup</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch Depth</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth 1</td>
<td>559.29</td>
<td>16.86</td>
<td>0.13</td>
<td>4302.23</td>
<td>129.69</td>
</tr>
<tr>
<td>Depth 2</td>
<td>358.00</td>
<td>427.00</td>
<td>0.14</td>
<td>2557.14</td>
<td>3050.00</td>
</tr>
<tr>
<td>Depth 3</td>
<td>1509.67</td>
<td>887.17</td>
<td>0.17</td>
<td>8880.41</td>
<td>5218.65</td>
</tr>
<tr>
<td><strong>Branch Width</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width 1</td>
<td>3954.17</td>
<td>3870.00</td>
<td>0.21</td>
<td>18,829.38</td>
<td>18,428.57</td>
</tr>
<tr>
<td>Width 2</td>
<td>2174.33</td>
<td>1943.50</td>
<td>0.19</td>
<td>11,443.84</td>
<td>10,228.95</td>
</tr>
<tr>
<td>Width 3</td>
<td>1114.47</td>
<td>2298.20</td>
<td>0.20</td>
<td>5572.35</td>
<td>11,491.00</td>
</tr>
<tr>
<td><strong>Branch Condition</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition 1</td>
<td>220.47</td>
<td>19.88</td>
<td>0.14</td>
<td>1574.79</td>
<td>142.00</td>
</tr>
<tr>
<td>Condition 2</td>
<td>522.14</td>
<td>140.00</td>
<td>0.15</td>
<td>3480.93</td>
<td>933.33</td>
</tr>
<tr>
<td>Condition 3</td>
<td>561.50</td>
<td>75.25</td>
<td>0.14</td>
<td>4010.71</td>
<td>537.50</td>
</tr>
<tr>
<td><strong>Real-world Program</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex 1</td>
<td>780.67</td>
<td>20.08</td>
<td>0.24</td>
<td>3252.79</td>
<td>83.67</td>
</tr>
<tr>
<td>Complex 2</td>
<td>6002.92</td>
<td>95.08</td>
<td>0.26</td>
<td>23,088.15</td>
<td>365.69</td>
</tr>
<tr>
<td>Complex 3</td>
<td>1361.44</td>
<td>844.22</td>
<td>0.19</td>
<td>7165.47</td>
<td>4443.26</td>
</tr>
<tr>
<td>Complex 4</td>
<td>5378.27</td>
<td>2655.82</td>
<td>0.41</td>
<td>13,117.73</td>
<td>6477.61</td>
</tr>
<tr>
<td>Complex 5</td>
<td>3975.45</td>
<td>909.00</td>
<td>0.20</td>
<td>19,877.25</td>
<td>4545.00</td>
</tr>
<tr>
<td>Complex 6</td>
<td>2379.80</td>
<td>44.40</td>
<td>0.16</td>
<td>14,873.75</td>
<td>277.50</td>
</tr>
<tr>
<td>Complex 7</td>
<td>993.38</td>
<td>77.62</td>
<td>0.17</td>
<td>5843.41</td>
<td>456.59</td>
</tr>
<tr>
<td>Complex 8</td>
<td>1223.70</td>
<td>1167.00</td>
<td>0.28</td>
<td>4370.36</td>
<td>4167.86</td>
</tr>
<tr>
<td>Complex 9</td>
<td>1082.22</td>
<td>380.56</td>
<td>0.28</td>
<td>3865.07</td>
<td>1359.14</td>
</tr>
<tr>
<td>Complex 10</td>
<td>1496.67</td>
<td>473.11</td>
<td>0.55</td>
<td>2721.22</td>
<td>860.20</td>
</tr>
</tbody>
</table>

The results depicted in Table 3 show the average total executions for programs resembling real-world PLC programs and those with varying branch widths, depths, and conditions. The results demonstrate that our fuzzer has fewer executions of the binary during fuzzing, which is a desirable property. That is, on average, it executes the binary less than 100 times compared to AFL++/IG, which has several hundreds of executions. Nevertheless, AFL++/IG also executes the binary fewer times than AFL++. Obviously, the proposed input generation allows AFL++/IG to apply a more focused fuzzing strategy, which reduces the number of executions required to find the bug. In contrast, AFL++ requires a significantly higher number of executions, ranging from $\times 10,000$ to $\times 100,000,000$. As evident in Figure 10, our fuzzer exhibits the best efficiency in terms of the number of executions of the binary during fuzzing. A smaller number of executions is desired because a high number of executions increases the workload of the fuzzer. This metric is correlated to the speed of the fuzzer since a faster fuzzer would find the bugs with fewer executions of the binary. The AFL++/IG has the second-best performance based on the executions of the test cases. These findings showcase the improvement enabled by the input generation technique adopted in our fuzzer and in AFL++/IG.
Figure 8. Time to find the bugs for the programs with various branch depths (top), widths (second from the top), conditions (third from the top), and real-world programs (bottom).

Through the evaluation of the proposed fuzzing framework, we showcased its efficiency. In particular, by leveraging the proposed PLC runtime to design an input generation mechanism, we significantly improved the performance of the AFL++ fuzzer, putting forward the AFL++/IG paradigm. More importantly, our custom-made fuzzer, embedding a custom mutator, consistently surpassed AFL++ and AFL++/IG in detecting logic bugs within ST control logic programs of diverse complexities.

However, the derived results should be cautiously interpreted, as the experimental setup involved carefully designed programs to evaluate the capabilities of the considered fuzzers. A more comprehensible dataset of more realistic ST programs may introduce additional challenges.

Nevertheless, the experimental evaluation validates the hypothesis that conventional fuzzing tools are sub-optimal for discovering logic errors in ST-based PLC control logic applications. In this direction, our novel fuzzer significantly accelerates the logic bug discovery process and offers a more robust and targeted testing mechanism, thereby contributing to the overall security posture of PLCs.
Figure 9. Speedup of the proposed fuzzer in finding the bugs compared against AFL++ for the programs with various branch depths (top), widths (second from the top), conditions (third from the top), and real-world programs (bottom).
Table 3. Average total number of executions for each fuzzer for the depth, width, condition test cases, and real-world programs. A small number of executions is desirable.

<table>
<thead>
<tr>
<th>Characteristics</th>
<th>AFL++</th>
<th>AFL++/IG</th>
<th>Our Fuzzer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Branch Depth</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Depth 1</td>
<td>273,970.00</td>
<td>5591.57</td>
<td>7.43</td>
</tr>
<tr>
<td>Depth 2</td>
<td>179,891.75</td>
<td>334,971.25</td>
<td>9.88</td>
</tr>
<tr>
<td>Depth 3</td>
<td>942,404.50</td>
<td>441,799.67</td>
<td>13.50</td>
</tr>
<tr>
<td><strong>Branch Width</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Width 1</td>
<td>1,882,189.50</td>
<td>1,459,524.83</td>
<td>1.0</td>
</tr>
<tr>
<td>Width 2</td>
<td>710,572.67</td>
<td>577,395.33</td>
<td>1.0</td>
</tr>
<tr>
<td>Width 3</td>
<td>765,733.93</td>
<td>1,677,735.47</td>
<td>1.0</td>
</tr>
<tr>
<td><strong>Branch Condition</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Condition 1</td>
<td>110,527.24</td>
<td>6581.53</td>
<td>3.65</td>
</tr>
<tr>
<td>Condition 2</td>
<td>91,400.43</td>
<td>16,189.00</td>
<td>1.00</td>
</tr>
<tr>
<td>Condition 3</td>
<td>122,671.00</td>
<td>7353.38</td>
<td>4.38</td>
</tr>
<tr>
<td><strong>Real-world Program</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Complex 1</td>
<td>64,996.17</td>
<td>772.92</td>
<td>10.0</td>
</tr>
<tr>
<td>Complex 2</td>
<td>2,841,554.00</td>
<td>3275.00</td>
<td>13.62</td>
</tr>
<tr>
<td>Complex 3</td>
<td>367,426.44</td>
<td>124,724.56</td>
<td>5.56</td>
</tr>
<tr>
<td>Complex 4</td>
<td>2,967,611.73</td>
<td>1,427,264.82</td>
<td>1.00</td>
</tr>
<tr>
<td>Complex 5</td>
<td>1,615,857.82</td>
<td>106,716.45</td>
<td>7.64</td>
</tr>
<tr>
<td>Complex 6</td>
<td>734,122.60</td>
<td>1464.20</td>
<td>1.00</td>
</tr>
<tr>
<td>Complex 7</td>
<td>96,081.50</td>
<td>2249.12</td>
<td>3.25</td>
</tr>
<tr>
<td>Complex 8</td>
<td>189,611.80</td>
<td>181,804.10</td>
<td>58.70</td>
</tr>
<tr>
<td>Complex 9</td>
<td>141,941.11</td>
<td>18,508.11</td>
<td>60.11</td>
</tr>
<tr>
<td>Complex 10</td>
<td>402,542.33</td>
<td>36,779.22</td>
<td>95.11</td>
</tr>
</tbody>
</table>

Figure 10. Cont.
5. Related Work

In this section, we review the existing literature, focusing on works that introduce some fuzzing techniques in the realm of ICS. Generally, ICS Fuzzing can be categorized into network protocol fuzzing and device fuzzing. Although our fuzzer lies in the second category, for completeness reasons, we also discuss studies in the field of network protocol fuzzing.

Network protocol fuzzing: Fuzzing often involves the security testing of communication protocols [10–12]. For instance, in PropFuzz [13], the authors present a protocol fuzzer designed for proprietary ICS protocols. The tool fuzzes network protocols by analyzing the communication between a device, say, a PLC, and an integrated development environment (IDE) to extract and mutate the relevant network fields. Then, it sends the sniffed (mutated) packets to the device. In addition, the fuzzer receives execution feedback by monitoring an output channel of the PLC. Similarly, Luo et al. present Polar [14], a protocol fuzzer that extracts semantic information from ICS protocol packets to identify vulnerable protocol function fields. The tool uses static analysis and dynamic taint analysis to identify the fields that are most likely to contain vulnerable operations and to fuzz them. The authors in CGFuzz [15] propose a deep learning-based approach for fuzzing IIoT protocols, focusing on the DNP3 protocol. Their tool uses a coverage-guided generative adversarial network model (GAN), namely, CovGAN, to learn the specifications of the underlying network protocol and generate fuzz test cases. The authors demonstrate the effectiveness of CGFuzzer in identifying vulnerabilities in the DNP3 protocol by evaluating the fuzzer on a public dataset and an in-house capture dataset. Overall, the discussed studies underscore the significance of protocol fuzzing in improving the security of ICS devices, emphasizing the need for tools and methodologies to address the unique challenges of ICS security testing.
**Device Fuzzing:** Fuzzing has also been applied to test the security of ICS devices like PLCs. The most common approaches include emulation-based, harness-based, and on-device fuzzing [44]. Emulation-based fuzzing involves testing the firmware of the device through an emulator, i.e., a technique also known as re-hosting. For instance, developed by Scharnowski et al., **Fuzzware** [45] is a fuzzing tool that uses memory-mapped I/O (MMIO) to model the peripherals of devices to fuzz their firmware. It is designed to identify vulnerabilities in the firmware of embedded devices via concolic execution-based fuzzing. Similarly, Tychalas et al. present **IFFSET** [46], a fuzzing tool that uses Quick-Emulation (QEMU) to emulate the Linux-based ICS firmware of Codesys PLCs. The approach relies on reverse engineering the firmware to extract the linked shared objects of I/Os and network-related code that are called from a harness during fuzzing. The authors of **P2IM** [47] propose a hardware-independent firmware fuzzing via peripherals modeling. Their tool instantiates a device model from an abstract model for the ARM Cortex-M architecture. During fuzzing, their tool can discriminate between benign and crashing inputs through the proposed model. In **Sizzler** [48], the authors present a tool for fuzzing QEMU-based emulated PLC firmware. It utilizes a sequential generative adversarial network (SeqGAN) to generate test cases for mutation-based fuzzing. Unlike emulation-based fuzzing, harness-based fuzzing involves using a harness to connect the fuzzer and the device. For example, **SP-Fuzz** [49] is a tool that generates a harness in a semi-automated fashion using context information from the execution of the PLC runtime. In contrast, on-device fuzzing involves executing the fuzzer directly on the device, e.g., **ICSFuzz** [3].

In detail, **ICSFuzz** [3] is a novel fuzzing technique designed for ICS control applications. Their framework investigates the potential for exploiting PLC binaries and their environment by repurposing binary code to perform instrumented fuzzing. Their framework stands out for its capacity to manipulate I/Os and fuzz the PLC runtime. To evaluate **ICSFuzz**, the authors use a set of in-house PLC binaries, along with functional control applications from online repositories. During instrumentation, they use **angr** [50] to identify locations within the binary, where instructions could be injected for receiving execution feedback.

Although our work shares similarities with the one presented by Tychalas et al. [3], it also addresses several limitations in the **ICSFuzz** framework. First, **ICSFuzz**’s focus on vulnerabilities that can be exploited with conventional attacks is prone to missing logical bugs arising from the unique programming paradigms of the PLCs. Contrarily, our fuzzer is designed to discern and generate inputs that explore the logical flow of ST applications, going beyond the binary execution level to reveal logic-based vulnerabilities. Second, **ICSFuzz** heavily relies on an existing cross-compiler and operates natively on the PLC, which may restrict its usability across different PLC hardware and software configurations. Conversely, our framework tackles this limitation by incorporating a PLC runtime that abstracts the hardware layer, allowing for a more flexible and adaptable testing environment that can be applied across various PLC platforms. Last, operating **ICSFuzz** is constrained by Codesys-based applications modified for Wago PLCs, posing a challenge for broader applicability. In contrast, our fuzzer is developed per the IEC 61131-3 standard, ensuring compatibility with a broader range of industrial programming environments and making it a broadly applicable tool in the domain of PLC security testing.

The work on the symbolic execution of programmable logic controller code in **SymPLC** [41] is designed to automatically test PLC software written in languages specified by the IEC 61131-3 standard and introduces PLC-specific reduction techniques to eliminate redundant interleavings. While SymPLC represents a methodological advance, particularly in its systematic approach to exploring program paths, it differs from our fuzzing-based approach in several key aspects. Symbolic execution, as implemented in SymPLC, is inherently static and relies on a priori knowledge of all possible inputs and states, which may not be feasible for complex industrial control software that interacts with dynamic and unpredictable physical environments. In contrast, our fuzzing technique does not require exhaustive knowledge of all possible states. Instead, it dynamically generates inputs that probe the control logic in ways likely to occur during actual operation, potentially revealing
logical errors that would not be manifested through static analysis alone. This dynamic aspect is particularly crucial, given the real-time operational constraints and the need for immediate response to physical process changes in industrial settings. Furthermore, SymPLC may encounter challenges when scaling to large, complex control systems due to the state explosion problem inherent in symbolic execution. On the other hand, our fuzzing methodology deftly scales as the complexity of the control logic increases since it is guided by runtime behavior and feedback, allowing it to focus on the most promising areas of the code.

6. Conclusions and Future Work

This research presents a novel approach for fuzzing PLC control logic applications, particularly targeting ST programs as specified in the IEC 61131-3 standard. The empirical evaluation of the proposed fuzzer through an extensive suite of ST programs with varying levels of complexity of alternative characteristics has yielded significant insights into the effectiveness of our methodology.

The proposed fuzzer demonstrates a consistent superiority over the conventional AFL++ fuzzer, achieving substantial speedups of several orders of magnitude and reduced bug detection times (less than a second) across all considered dimensions. This performance advantage may be attributed to our custom fuzzing mutator engine, which adapts the structure of fuzz inputs to ST programs and to the input generation mechanism, which provides good initial inputs for fuzzing. By feeding random input to targeted locations with the help of a finely tuned runtime environment, the proposed fuzzer has been proven to be an efficient tool for fuzzing PLC control applications.

The robustness of our fuzzer is evident in its ability to handle wide branches, deep nested conditions, and various program constructs, outperforming AFL++ in terms of speed. Through the evaluation of our fuzzer, we showcase its potential for improving the security and reliability of PLC software, also underscoring the limitations of applying general-purpose fuzzing tools like AFL++ to such specialized programming environments.

While the tests were conducted in a controlled software environment with carefully designed programs, the promising results advocate for the deployment and further testing of our fuzzer in real-world industrial settings. An obvious avenue for future work includes refining the fuzzing algorithms, testing the harness in a controlled environment, and validating its effectiveness on a more extensive dataset. Additionally, we plan to extend the framework to support other IEC 61131-3 languages, such as Ladder Diagram (LD), which is widely used in industry. Fuzzing LD poses unique challenges due to its graphical nature, and we aim to address these challenges in future work to enhance further the framework's applicability and utility in real-world industrial settings. Another intriguing direction for future research involves integrating open-source language models to automate the generation of the fuzzing harness as a specification of PLC software, thus further improving the performance and usability of the proposed framework.

Author Contributions: Conceptualization, C.K.; Software, K.A.K. and J.S.; Validation, V.K. and J.S.; Formal analysis, J.S.; Investigation, K.A.K.; Data curation, V.K.; Writing—original draft, K.A.K. and C.K.; Writing—review & editing, V.K., C.K. and R.C.I.; Supervision, C.K.; Project administration, C.K.; Funding acquisition, R.C.I. All authors have read and agreed to the published version of the manuscript.

Funding: Work supported through the INL Laboratory Directed Research & Development (LDRD) Program under DOE Idaho Operations Office Contract DE-AC07-05ID14517.

Data Availability Statement: All data and scripts are publicly accessible through the following link: https://github.com/kandersonko/StructuredFuzzer, accessed on 31 May 2024.

Conflicts of Interest: The authors have no other conflicts of interest to disclose.
Abbreviations

The following abbreviations are used in this manuscript:

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>IG</td>
<td>Input Generation</td>
</tr>
<tr>
<td>AFL</td>
<td>American Fuzzy Lop</td>
</tr>
<tr>
<td>AFL++</td>
<td>American Fuzzy Lop Plus Plus</td>
</tr>
<tr>
<td>ST</td>
<td>Structured Text</td>
</tr>
<tr>
<td>SCA</td>
<td>Static Code Analysis</td>
</tr>
<tr>
<td>IEC</td>
<td>International Electrotechnical Commission</td>
</tr>
<tr>
<td>PLC</td>
<td>Programmable Logic Controller</td>
</tr>
<tr>
<td>LL</td>
<td>Ladder Logic</td>
</tr>
<tr>
<td>LD</td>
<td>Ladder Diagram</td>
</tr>
<tr>
<td>FBD</td>
<td>Function Block Diagram</td>
</tr>
<tr>
<td>SFC</td>
<td>Sequential Function Chart</td>
</tr>
<tr>
<td>ICS</td>
<td>Industrial Control Systems</td>
</tr>
<tr>
<td>HMI</td>
<td>Human–Machine Interface</td>
</tr>
</tbody>
</table>

References


Disclaimer/Publisher’s Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.