Safeguarding Industrial Automation: A Fuzzing Framework for PLC Control Logic

Marie Louise Uwibambe, Qinghua Li
Department of Electrical Engineering and Computer Science
University of Arkansas
{uwibambe, qinghual}@uark.edu

***Abstract*—As Programmable Logic Controllers (PLCs) become more integrated into complex industrial systems and networks, their software components face increased security vulnerabilities. Fuzz testing, which uses unexpected inputs to find vulnerabilities, has been used to enhance PLC security but is primarily focused on communication protocols, often overlooking control logic. This paper presents a fuzzing framework for PLC control logic programs. The framework is designed to handle control logic's unique execution process, which may involve translating PLC languages into general-purpose languages. It generates structurally valid test cases and is platform-independent. We implement our prototype around the OpenPLC environment and expand our experiments to a Wago PLC image to evaluate the feasibility of our method on real-world devices. Our approach exhibits strong performance in terms of code coverage, test case pass rate, and crash detection, while ensuring high time efficiency.**

***Index Terms*—fuzzing, control logic, PLC**

I. Introduction

Programmable Logic Controllers (PLCs) are widely used in modern industrial networks. However, their software components, including control logic, face increased security vulnerabilities [1]. In addition, control logic runtime environments are often adapted to support programming languages that are known to be vulnerable. Notable examples include mixing C++ into the control logic programs [2], integrating C modules in control logic [3], and translating control logic to C++ before execution [4]. This integration highlights the necessity for rigorous bug identification in control logic as a compromise in the security of control logic can allow attackers to launch significant attacks, leading to production disruptions. Stuxnet [5] exemplifies this type of scenario. Stuxnet compromises the Siemens SIMATIC STEP7 engineering software and infects the control logic of a Siemens S7-300 PLC, modifying the motor speed of centrifuges periodically from 1410 Hz to 2 Hz and then to 1064 Hz.

Techniques such as symbolic execution [6], context-aware program analysis [7], and data-driven structure learning [8] have been employed for security testing in control logic. However, their effectiveness is limited because they cannot cover a significant portion of the input space, leading to a restricted number of test cases. Additionally, these methods primarily concentrate on detecting safety violations within control logic but not software bugs that can make the system vulnerable. As an alternative approach, fuzzing has emerged, offering a broader range of test cases and deeper coverage for vulnerability discovery within control logic. Fuzz testing, or fuzzing, is a dynamic security analysis technique that involves sending unexpected inputs (test cases) into a system to discover vulnerabilities that may be exploited by malicious actors [9].

Fuzzing efforts for PLC software components have predominantly focused on fuzzing communication protocols [10]–[12] while overlooking control logic. However, research has shown that control logic can be vulnerable to memory-related flaws, which, if left unaddressed, can be exploited [13]. Following this discovery, research efforts have been directed toward fuzzing control logic programs. ICSFuzz [13] tests for vulnerabilities in CodeSys control programs and identifies potential vulnerabilities within its runtime environment. Fieldfuzz [14] adopts the same approach but uses a vendor-agnostic strategy. Sizzler [15] improves test case generation by utilizing seqGAN to enhance the fuzzing of ladder diagram code while identifying logical errors, e.g., missing coils and infinite loops.

Although previous efforts have established a foundation for fuzzing control logic, they do not address scenarios where control logic code must be translated into general-purpose languages (e.g., from Structured Text to C/C++). Several runtime environments such as those presented in [16]–[20] translate control logic code into general-purpose languages before execution. Current tools for testing control logic cannot handle these scenarios, as they are designed to process control logic in their native languages and do not account for a stage where the code is translated into a general-purpose language. Furthermore, while control logic is translated into general-purpose programming languages, standard fuzzing frameworks are insufficient for testing it. This is because control logic has unique characteristics that require it to be executed within a specialized runtime environment, making general-purpose tools ineffective for thorough testing. In other words, standard fuzzing tools are designed to test standalone binaries, rather than programs that operate within another layer (control logic running on top of its dedicated runtime environment). This is further discussed in Section II.

The absence of tools for testing these programs suggests that vulnerabilities in translated control logic may go undetected. This poses a significant issue, considering that the translation is into languages with security flaws. Moreover, control logic

979-8-3315-2096-0/25/$31.00 ©2025 IEEE

translation is an active practice in the automation industry, further exacerbating the risk. Companies like Airsoft Automation [4], GEB automation [21], and Beckhoff Automation [22] offer control logic translation and products compatible with the translated code. This also applies to environments utilized for academic research and prototyping, such as OpenPLC [20] and Beremiz [23].

Other limitations in existing tools include being vendor-specific, which makes it difficult to adapt to other problems, and lacking an understanding of the structure of expected program inputs, resulting in an inability to generate effective test cases. Table I shows a summary of these limitations.

TABLE I
COMPARISON OF EXISTING WORK FOR FUZZING CONTROL LOGIC

	ICSFuzz [13]	FieldFuzz [14]	Sizzler [15]	This work
Platform-independent	NO	YES	YES	YES
Input structure awareness	NO	NO	YES	YES
Fuzz translated code	NO	NO	NO	YES

This work provides the following contributions:

1) We present a fuzzing framework for control logic programs that are written in PLC languages and translated into general-purpose languages before execution.
2) We propose a method for running control logic that detaches it from the runtime environment. This addresses the challenge of fuzzing a program embedded within another (control logic within a runtime environment).
3) We design a structure-aware input mutation technique to generate effective test cases for control logic.
4) We conduct an experimental validation on both a soft PLC (OpenPLC) and an emulated image of a commercial PLC (WAGO).

The rest of this paper is organized as follows: Section II covers preliminary control logic concepts and challenges for fuzzing. Section III outlines our methodology. Evaluations are presented in Section IV. Related work is reviewed in Section V, and the final section concludes the paper.

II. CONTROL LOGIC FUNDAMENTALS

A. PLC Control logic

The control logic within a PLC is programmed to execute specific tasks in ICS environments (e.g., consumption optimization in electricity, emergency shutdowns in safety systems, conveyors in material handling, etc.). Digital and analog inputs representing real-world conditions are continuously monitored in these environments. The control logic processes these inputs using programmer-defined logical operations and conditional statements. Based on the preset instructions, the PLC sends corresponding outputs to the actuators, orchestrating the required automation sequence. This continuous process is known as an execution cycle and ensures that the control logic is responsive to real-time changes in the industrial environment.

Historically, control logic programs have been written using proprietary languages. Nevertheless, there have been recent efforts to establish standardization in this domain. IEC 61131-3 is an international standard that defines a set of standardized programming languages to enhance interoperability and ease of use in industrial automation and control systems. It encompasses five programming languages: Ladder Diagram (LD), Structured Text (ST), Function Block Diagram (FBD), Instruction List (IL) and Sequential Function Chart (SFC). Since its introduction in 1993, IEC 61131-3 has been widely accepted by the international user and vendor community.

B. PLC runtime environments

The PLC runtime environment is the platform for loading and executing control logic programs. This includes providing essential services such as launching networks and managing I/O operations. Runtime environments for executing control logic can be categorized into two types: those that generate machine code directly and those that convert control logic into general-purpose languages first. In this work, we focus on the latter approach.

C. PLC program translation

Control logic translation is widespread due to the absence of cross-compilers tailored for prevalent PLC programming languages. The lack of a clearly defined execution model for the IEC 61131-3 standard emphasizes this challenge. Consequently, PLC programmers opt to convert their code into languages supported by robust and effective compiler toolchains.

The translated programs include code segments that remain consistent across all programs. Such code portions are generated from the runtime environment and encompass tasks found in all programs. Examples include imports of fundamental libraries, data logging mechanisms, and undefining identifier definitions (#undef). Essentially, these sections serve as standardized templates that initiate the code translation and are compiled with the translated control logic.

D. Control logic fuzzing challenges

The nature of control logic requires it to be executed within a dedicated runtime environment. Thus, even after translating control logic, the resulting program cannot be executed without its runtime environment. Doing so would disregard the PLC run cycle, I/O operations, and response times, all of which are critical in ICS. This makes standard fuzzing tools inadequate for testing control logic, as these tools are designed for independent programs rather than those functioning within a runtime environment.

III. OUR APPROACH

The proposed method was primarily implemented on OpenPLC. Additional experiments were conducted on a WAGO PLC image to assess the feasibility of the method on a real-world device.

A. Control logic execution

Challenge. As previously discussed, translated control logic executes within the runtime environment; in fact, the runtime environment contains the main function essential for initiating control operations. Therefore, it is not possible to test the translated control logic independently, without the runtime environment. A simplistic approach would involve treating the runtime environment and control logic as a single unit and fuzzing them together. However, this method effectively fuzzes the runtime environment rather than the control logic itself. Since the runtime typically has a larger and more complex codebase, this approach shifts the focus and resources away from the control logic and fails to ensure that the testing effectively penetrates the control logic program. For instance, OpenPLC features heavy components designed for a user-friendly web server used for visualization. However, these components should not be the primary focus when testing control logic programs, as they do not influence the execution or output of the control logic itself.

Solution. To address this challenge, we propose focusing solely on the code related to control operations by extracting only the necessary components from the runtime environment. Specifically, we incorporate the lightweight main function that initiates control operations, along with the foundational template mentioned earlier. Details of this custom execution process are discussed below.

The OpenPLC runtime is typically accessed through a web server. The web server receives a target control logic and copies it into a directory reserved for control logic. Subsequently, the server utilizes a binary called iec2c to read the directory and translate the control logic code. In our case, the server graphical user interface has not been initiated to avoid running unnecessary components. Thus, we execute the iec2c binary using bash script via a command line interface and pass the target control logic as an argument. This results in a translated C code. Following this, we compile the C code and generate object files. Since the resulting object files lack the main function necessary for initialization, we compile the runtime main program with the above-mentioned object files as arguments. The resultant binary serves as our target and can run independently without the runtime. It comprises the translated control logic code and the runtime components that influence control operations. This enables fuzzing to concentrate solely on control operations. Note that we fuzz the control logic after it has been translated into general-purpose languages, in our case, C/C++. This is necessary because the execution takes place using the translated code, which is where potential bugs can be identified.

B. Structure-aware input mutation

Standard fuzzing tools do not comprehend the structure of the configuration files used as test cases in many ICS fuzzing campaigns. As shown in Table II, We attempted to fuzz with AFL++ [24] mutators, but most test cases were rejected by the target program at the initial stages of execution. To address this, we extend AFL++ and develop an input mutation approach that adheres to predefined rules during the mutation process. The rules specify the required fields and the permissible value space for each field. During mutation, the mutator modifies the field's value, ensuring it remains within the defined space. For instance, a field such as "Operating system" may have a value space comprising "Linux" or "Windows". Consequently, the mutator generates test cases where the Operating System field remains unchanged while the value oscillates between Linux and Windows. While these test cases adhere to proper syntax, it's important to note they are not always correct. For instance, configuring the operating system field as Windows while running Linux is structurally valid but incorrect from a system perspective. Such scenarios generate inputs that are structurally valid but systemically flawed, helping to uncover if the target has adequate sanitization measures and at the same time penetrates deep paths.

The initial seed is given by the user. We generate subsequent test cases with our mutation technique and leverage AFL++ instrumentation which gathers necessary feedback and decides when to stop mutating a seed. The next seed to mutate and the mutation fields are selected randomly. Note that the test cases used to fuzz the target system are various system configurations. In OpenPLC, configurations are dynamically inferred at runtime, influencing how the code executes based on settings like target I/O connections, host platform, and serial communication. To enable effective testing, we modify the runtime process to accept configurations from files containing our test cases. This approach allows us to simulate a range of scenarios that may be encountered in practice.

C. Overall workflow of the proposed framework

Fig. 1. illustrates the workflow of the fuzzing campaign after control logic is translated, which includes the target program, a fuzzing tool, and a custom structure-aware mutator. The detailed process is explained below.

① **Identification of Essential Runtime Components**. At this stage, we identify and extract essential components of the runtime that contribute to control operations as discussed in III-A.

② **Compiling the target**. The translated control logic is compiled on top of the runtime components extracted in step ①. The result is the fuzzing target binary.

③ **Transfer of target binary to the execution engine**. The resulting binary is sent to the execution engine. The execution engine is a set of resources necessary to initiate the execution of the target program, including the binary to be executed, the input fed to the binary, and the scripts required to automate this process. These components are organized into a dedicated directory.

④ **Field extraction**. At this stage, we use the seed provided by the user to extract the necessary fields. A user-defined delimiter distinguishes the field names from the rest of the content.

⑤ **Value range definition**. The user defines the range of acceptable values for each field identified in ④. This range

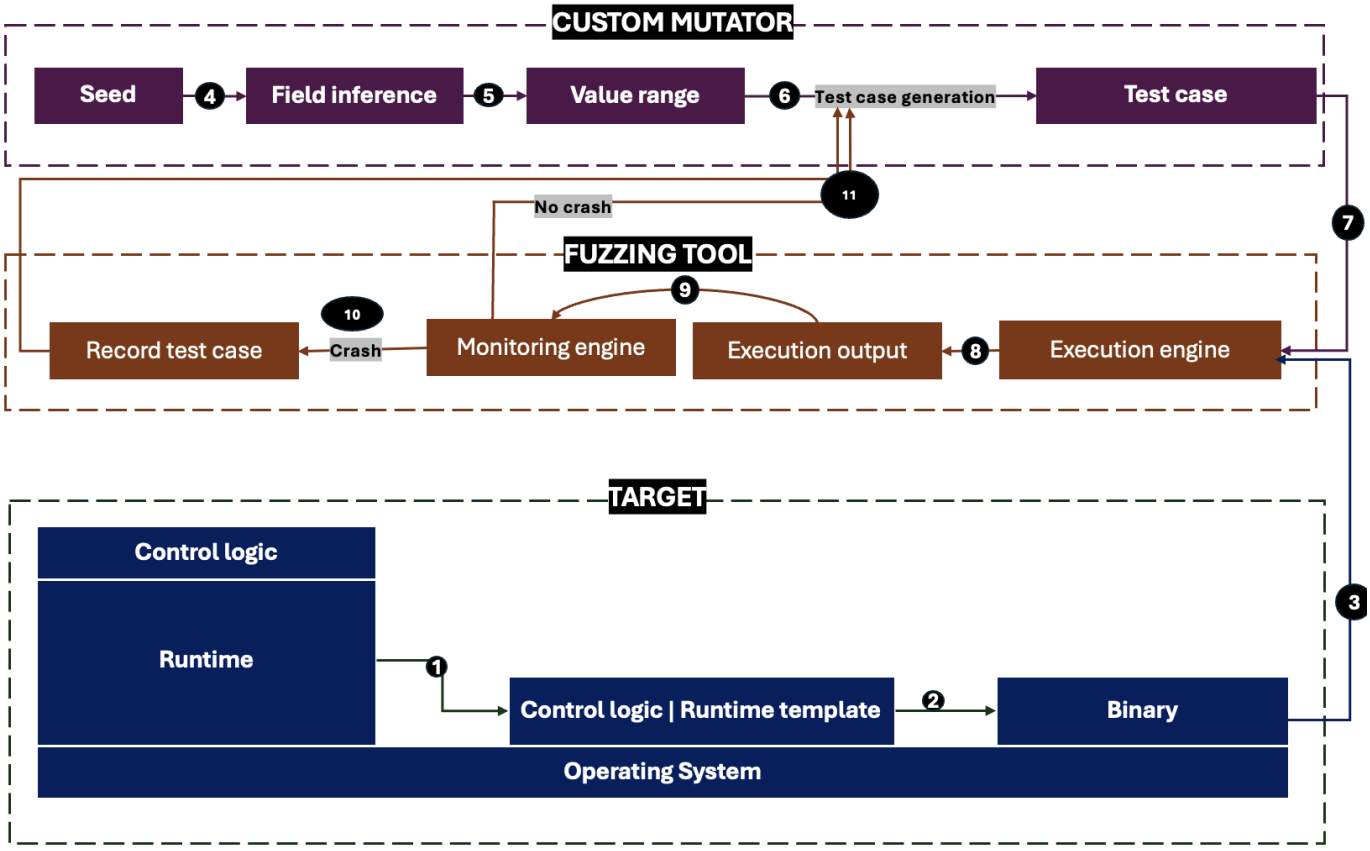

Fig. 1. Workflow of the proposed fuzzing framework

can be a numerical interval or an array of acceptable values for non-numerical data.

⑥ **Test case generation**. Based on the previously defined fields and their corresponding values, test cases (inputs to the program) are generated. While the fields remain constant, the values fluctuate within the specified ranges. For example, if there is a field named 'temperature' with a value range of 0 to 100 degrees, the keyword 'temperature' will remain the same in each input, while the value can vary as long as it falls between 0 and 100 degrees.

⑦ **Send test case to the execution engine**. The input generated in ⑥ is sent into the execution engine described in ③.

⑧ **Executing the binary**. The execution engine runs the target program against the current test case and generates an output.

⑨ **Execution monitoring**. At this stage, the monitoring engine observes the ongoing execution and provides feedback on the execution. The monitoring engine incorporates a debugger that tracks the program's current status, along with scripts designed to automatically report the occurrence of a crash and the causal test case.

⑩ **Crash recording**. Once a crash is reported, the causal input is recorded.

⑪ **Repeat the process**. If no crash is found, the next test case is generated, and the fuzzing campaign follows the same procedures until the user cancels it.

IV. EVALUATION

A. Experiment set-up

1) OpenPLC: We applied our proposed methodology to extract control operations from a traffic light program written in Structured Text (ST), omitting unnecessary runtime components. We then conducted two one-hour experiments. First, we performed standard fuzzing on the resulting binary using AFL++. Second, we perform the same experiment using our mutator. This allowed us to assess the improvements using mutation techniques tailored to control logic programs. OpenPLC was hosted on a 64-bit Linux virtual machine running Ubuntu 22.04 LTS OS and was configured with 4 virtual CPUs, 12 GB of base memory, and 50 GB of storage.

2) Wago PLC: The second phase of our experiments aimed to validate the scalability of our methodology to a real PLC. For this, we utilized an emulated image of the Wago PFC-100 from the work in [25]. Detailed instructions on running this image are available in [26]. The Wago PFC-100 typically uses the CODESYS runtime, but as a Linux-based device, it supports other runtime environments that translate control logic. We use that flexibility to perform the same experiments

as above targeting the OSCAT library [27], an open-source set of function blocks commonly used in the CodeSys framework. We use the Matiec runtime to translate and host the control logic. Since the OSCAT library does not utilize configuration files, we adapt our inputs to target the variables defined within the program itself.

We first present the findings from the main prototype designed around OpenPLC, followed by a discussion of the results obtained from the WAGO PLC image.

B. Test cases pass rate

Test cases pass rate measures the number of test cases that successfully penetrate the system. It is a crucial metric in fuzzing Industrial Control Sytems (ICS) to ensure test cases adhere to the correct structure and are accepted by the system. We categorize the passing rate into three levels as follows:

- Level 1. A test case successfully passes the code segment responsible for verifying if the received file type is the allowed type.
- Level 2. A test case fulfills level 1 requirements, contains all the necessary fields, and the data types of the values are correct.
- Level 3. A test case that fulfills the above and the values in the configuration fall within the permissible range.

Table II demonstrates that by using AFL++ mutators, 76.4 percent of the test cases had the correct file type (level 1). However, only 17.64 percent contained the necessary data fields with values in the permissible data types (level 2). None of the test cases had values within the acceptable range (level 3). In contrast, when using our mutator, all the test cases had the correct file type (level 1), contained the necessary data fields with the correct data types (level 2), and had values within the acceptable range (level 3).

C. Crashes

As shown in Fig. 2. below, our method resulted in 47.06 percent more unique crashes. This is more effective in detecting bugs.

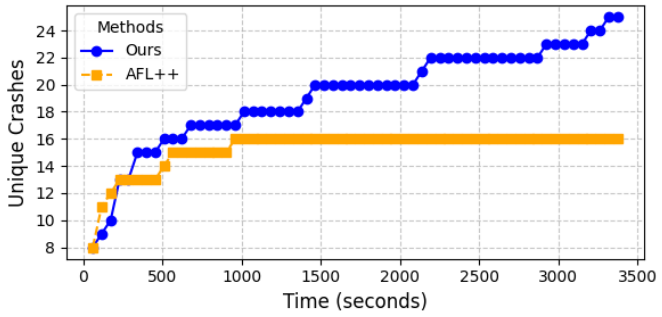

Fig. 2. Observed unique crashes over time

D. Code Exploration

We evaluate code exploration by counting the number of branch edges executed and the path depth attained during the fuzzing process. Our mutator achieves a deeper program path and more branches as shown in Table III.

E. Seed corpus effectiveness

We measure the effectiveness of the seed corpus used to generate test cases by examining the proportion of seeds that, when mutated, generate test cases that crash the system. Our method achieved a crash rate of 92% for the seeds, while AFL++ mutators resulted in a crash rate of 29.8%, and thus our method is much more effective.

TABLE II
TEST CASES PASS RATE IN OPENPLC

	Level 1 (%)	Level 2 (%)	Level 3 (%)
AFL++	76.4	17.64	0
Our mutator	100	100	100

TABLE III
CODE EXPLORATION

	Path depth	Branches
AFL++	4	428
Ours	8	447

F. Time complexity

We evaluate the time complexity based on the number of cycles completed in the fuzzing campaign. This number represents the count of queue passes completed within a specific timeframe. We achieved a better time complexity by 1.79 times. We attribute this improvement to the fact that when fuzzing with the AFL++ mutators, the fuzzer adds numerous test cases to the seed pool, necessitating the mutation of all of them. However, that approach does not improve the campaign since these seeds are generally ineffective. Fig. 3 shows the number of cycles completed over time.

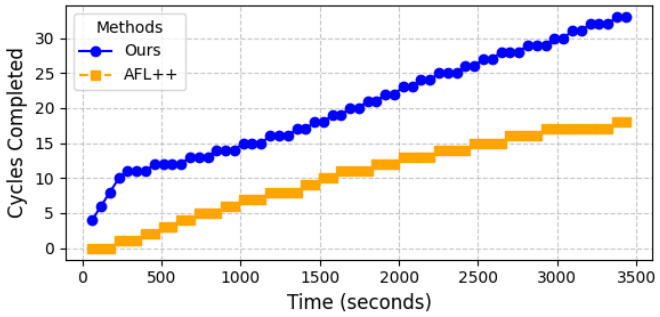

Fig. 3. Cycles done over time

G. Binary only targets

We conducted experiments using a binary-only program to learn how the fuzzing campaign is impacted when code instrumentation is not an option due to the absence of source code. We utilized a pre-compiled version of the program previously employed in our experiments. Data was collected

from two separate runs: one with the mutator enabled and another without it. The number of crashes observed with source code using standard AFL++ versus our method was approximately 3,900 and 10,000, respectively. Compared with Fig. 2., it can be observed that both methods showed an increase in the number of crashes. This increase is attributed to the lack of instrumentation, which causes the same crash to be recorded multiple times. Nevertheless, our method consistently yields a higher number of crashes overall."

H. Observations on a real PLC image

As aforementioned, we extend our testing on a Wago PLC. We begin by discussing the required manual pre-processing and lastly present the crashes identified.

Although the PLC image used is Linux-based, it lacks several components necessary for straightforward compilation. For instance, it does not include C++ compilers or libraries. To address this, we first compile the target program on a PC. The compilation was done using arm-linux-gnueabihf-gcc to match the PLC's architecture. Additionally, we statically linked the libraries to ensure that the absence of standard libraries on the PLC would not affect the target program's execution. Finally, we unmount the PLC image and copy the resulting target and fuzzer binaries to the PLC image. While these steps are manual, they require a reasonable amount of effort, as they are completed in a relatively short time and do not need to be repeated throughout the fuzzing campaign.

The number of crashes found is shown in Fig. 4. The results show that our approach is able to find crashes in the tested PLC image, and it outperforms AFL++.

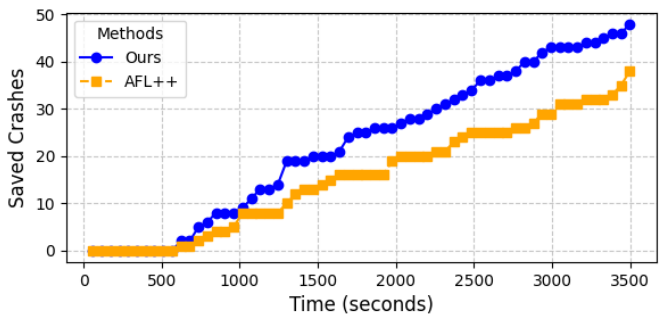

Fig. 4. Number of crashes over time in a Wago PLC.

V. Related work

A. Fuzzing control logic

The work in [13] examines binaries from IEC 61131-3 programming languages to identify compilation-based differences and potential vulnerabilities. Based on their findings, they developed a fuzzing framework to assess the security of PLC binaries and their interactions with host functions. Their approach faces several limitations, such as a lack of proper input delivery synchronization, slow fuzzing speed, manual crash monitoring, and the fact that it only supports a specific manufacturer (WAGO).

In contrast, Fieldfuzz [14] introduces a network-based framework dedicated to analyzing the security risks associated with the Codesys runtime. That study proposes a method for uncovering vulnerabilities across the computational stack of PLCs within the Codesys environment, including their control applications. However, their method relies on network traffic, which is not always easily accessible while testing control logic.

Sizzler [15] introduces a vendor-agnostic vulnerability detection framework tailored for PLC applications employing ladder diagrams. The framework adopts a mutation-based fuzzing approach leveraging Sequential Generative Adversarial Network (SeqGAN) techniques. Sizzler leverages a customized Quick Emulator (QEMU) instance and Inter-Integrated Circuit (I2C) protocols to accommodate targets with diverse architectures. However, it is noteworthy that Sizzler may not be suitable for extensive testing due to the substantial tracing overhead incurred when integrated with AFL++, resulting in a considerable performance impact of approximately 1300%.

Research efforts have also applied fuzzing on software programs run by programmable controllers (PCs). For instance, [28] introduces a traffic-based protocol fuzzing approach tailored for controllers, involving proprietary protocol fuzzing on the network daemon. However, their methodology necessitates significant manual intervention and relies heavily on real devices. Domain-specific fuzzing efforts for control software have been proposed [29]–[31], primarily targeting robotic vehicles (RVs). It is important to note that these differ from the standard PLC control logic we focus on in this study, as they are more complex and are not necessarily written languages or primarily designed to run on PLCs. Instead, they can operate on devices such as drones or RVs. The work in [32] addresses power grid fuzzing and introduces a novel method for detecting silent crashes in power grid software by monitoring electromagnetic wave emissions.

B. Other techniques used to test control logic

In addition to fuzzing, other techniques have been explored to enhance control logic security. SymPLC [33] is a symbolic execution tool designed for the automated testing of PLC software, adhering to the IEC 61131-3 standard programming languages. SymPLC takes the PLC source code as input and converts it into C before employing symbolic execution. VetPLC [34] conducts static program analysis. This process involves creating timed event causality graphs to comprehend the causal relationships among events within the PLC code. Additionally, VetPLC mines temporal invariants from data traces collected in ICS testbeds. This allows for a quantitative assessment of temporal dependencies, specifically those constrained by machine operations.

VI. Conclusion

In this paper, we proposed a fuzzing framework for control logic. The framework provides a specialized approach to fuzz translated control logic and employs a custom test case

mutation technique that understands the anticipated test case structure. Notably, our method yields a significant enhancement, resulting in a 47% increase in unique crashes and a 1.79-fold acceleration in fuzzing speed. Our next step is to conduct an in-depth study on the bugs identified and C/C++ functions associated with them.

ACKNOWLEDGMENT

This work is supported in part by the National Science Foundation under award 1751255.

REFERENCES

[1] Chih-Che Sun, Adam Hahn, and Chen-Ching Liu. Cyber security of a power grid: State-of-the-art. *International Journal of Electrical Power & Energy Systems*, 99:45–56, 2018.

[2] Beckhoff Automation. C as a programming language for machine control. https://www.beckhoff.com/en-us/company/news/c-as-a-programming-language-for-machine-control.html, 2024. Accessed: 2024-10-09.

[3] CODESYS Group. Integrating c code into the codesys development system. https://content.helpme-codesys.com/en/CODESYS%20Development%20System/_cds_integrating_c_code.html, 2024. Accessed: 2024-10-09.

[4] AirSoft Automation. vplc virtual programmable logic controller. http://us.arsoft-int.com/vplc.php, 2024. Accessed: 2024-10-09.

[5] Marie Baezner and Patrice Robin. Stuxnet. Technical report, ETH Zurich, 2017.

[6] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 326–336, 2017.

[7] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, et al. Towards automated safety vetting of plc code in real-world plants. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 522–538. IEEE, 2019.

[8] Zeyu Yang, Liang He, Hua Yu, Chengcheng Zhao, Peng Cheng, and Jiming Chen. Detecting plc intrusions using control invariants. *IEEE Internet of Things Journal*, 9(12):9934–9947, 2022.

[9] Xiaogang Zhu, Sheng Wen, Seyit Camtepe, and Yang Xiang. Fuzzing: a survey for roadmap. *ACM Computing Surveys (CSUR)*, 54(11s):1–36, 2022.

[10] Wenpeng Wang, Zhixiang Chen, Ziyang Zheng, and Hui Wang. An adaptive fuzzing method based on transformer and protocol similarity mutation. *Computers & Security*, 129:103197, 2023.

[11] Tao Fen, Deming Li, and Zhanting Yuan. An industrial network protocol fuzzing framework based on deep adversarial networks. In *2023 4th International Conference on Computer Engineering and Application (ICCEA)*, pages 590–596. IEEE, 2023.

[12] Zhenhua Yu, Haolu Wang, Dan Wang, Zhiwu Li, and Houbing Song. Cgfuzzer: A fuzzing approach based on coverage-guided generative adversarial networks for industrial iot protocols. *IEEE Internet of Things Journal*, 9(21):21607–21619, 2022.

[13] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. {ICSFuzz}: Manipulating {I/Os} and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In *30th USENIX Security Symposium (USENIX Security 21)*, pages 2847–2862, 2021.

[14] Andrei Bytes, Prashant Hari Narayan Rajput, Michail Maniatakos, and Jianying Zhou. Fieldfuzz: Enabling vulnerability discovery in industrial control systems supply chain using stateful system-level fuzzing. *arXiv preprint arXiv:2204.13499*, 2022.

[15] Kai Feng, Marco M Cook, and Angelos K Marnerides. Sizzler: Sequential fuzzing in ladder diagrams for vulnerability detection and discovery in programmable logic controllers. *IEEE Transactions on Information Forensics and Security*, 2023.

[16] M. de Sousa and A. Carvalho. An iec 61131-3 compiler for the matplc. In *EFTA 2003. 2003 IEEE Conference on Emerging Technologies and Factory Automation. Proceedings (Cat. No.03TH8696)*, volume 1, pages 485–490 vol.1, 2003.

[17] Zulfakar Aspar and Nurul Huda Abd Rahman. A basic language compiler for plc applications. In *Proceedings of the 6th International Conference on Electrical, Control and Computer Engineering: InECCE2021, Kuantan, Pahang, Malaysia, 23rd August*, pages 651–663. Springer, 2022.

[18] Tiago Henrique Acciaiuoli Azenha Catalão. An llvm based compiler for the iec 61131-3. 2020.

[19] Mário de Sousa, Jirı Baum, Andrey Romanenko, Pinhal de Marrocos, and II Pólo. Matplc: Towards real-time performance. In *proceedings of the 2003 real-time Linux workshop*, 2003.

[20] Thiago Rodrigues Alves, Mario Buratto, Flavio Mauricio De Souza, and Thelma Virginia Rodrigues. Openplc: An open source alternative to automation. In *IEEE Global Humanitarian Technology Conference (GHTC 2014)*, pages 585–589. IEEE, 2014.

[21] GEB Automation. Geb automation: Industrial automation solutions. https://www.gebautomation.com/, 2024. Accessed: 2024-10-09.

[22] Beckhoff Automation. Twincat: Automation software. https://www.beckhoff.com/en-us/products/automation/twincat/, 2024. Accessed: 2024-10-09.

[23] Beremiz. Beremiz: Open source ide for plc programming. https://beremiz.org, 2024. Accessed: 2024-10-09.

[24] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. {AFL++}: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*, 2020.

[25] Dimitrios Tychalas and Michail Maniatakos. Iffset: In-field fuzzing of industrial control systems using system emulation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 662–665. IEEE, 2020.

[26] Moma Lab. Iffset: A fuzzing framework for industrial control systems (ics). https://github.com/momalab/iffset, 2024. Accessed: 2024-10-05.

[27] CODESYS Store. Oscat basic library, 2024. Accessed: 2024-10-18.

[28] Puzhuo Liu, Yaowen Zheng, Zhanwei Song, Dongliang Fang, Shichao Lv, and Limin Sun. Fuzzing proprietary protocols of programmable controllers to find vulnerabilities that affect physical control. *Journal of Systems Architecture*, 127:102483, 2022.

[29] Hyungsub Kim, Muslum Ozgur Ozmen, Antonio Bianchi, Z Berkay Celik, and Dongyan Xu. Pgfuzz: Policy-guided fuzzing for robotic vehicles. In *NDSS*, 2021.

[30] Chijung Jung, Ali Ahad, Yuseok Jeon, and Yonghwi Kwon. Swarmflawfinder: Discovering and exploiting logic flaws of swarm algorithms. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 1808–1825. IEEE, 2022.

[31] Taegyu Kim, Chung Hwan Kim, Junghwan Rhee, Fan Fei, Zhan Tu, Gregory Walkup, Xiangyu Zhang, Xinyan Deng, and Dongyan Xu. {RVFuzzer}: Finding input validation bugs in robotic vehicles through {Control-Guided} testing. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 425–442, 2019.

[32] Marie Louise Uwibambe, Yanjun Pan, and Qinghua Li. Fuzzing for power grids: A comparative study of existing frameworks and a new method for detecting silent crashes in control devices. In *2023 IEEE Design Methodologies Conference (DMC)*, pages 1–6. IEEE, 2023.

[33] Shengjian Guo, Meng Wu, and Chao Wang. Symbolic execution of programmable logic controller code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 326–336, 2017.

[34] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, et al. Towards automated safety vetting of plc code in real-world plants. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 522–538. IEEE, 2019.