FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network

Andrei Bytes
Singapore University of Technology and Design
Singapore

Prashant Hari Narayan Rajput
Tandon School of Engineering
Brooklyn, New York, USA

Constantine Doumanidis
New York University Abu Dhabi
Abu Dhabi, UAE

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security
Saarbrücken, Germany

Michail Maniatakos
New York University Abu Dhabi
Abu Dhabi, UAE

Jianying Zhou
Singapore University of Technology and Design
Singapore

ABSTRACT

Networked Programmable Logic Controllers (PLCs) are proprietary industrial devices utilized in critical infrastructure that execute control logic applications in complex proprietary runtime environments that provide standardized access to the hardware resources in the PLC. These control applications are programmed in domain-specific IEC 61131-3 languages, compiled into a proprietary binary format, and process data provided via industrial protocols. Control applications present an attack surface threatened by manipulated traffic. For example, remote code injection in a control application would directly allow to take over the PLC, threatening physical process damage and the safety of human operators. However, assessing the security of control applications is challenging due to domain-specific challenges and the limited availability of suitable methods. Network-based fuzzing is often the only way to test such devices but is inefficient without guidance from execution tracing.

This work presents the FieldFuzz framework that analyzes the security risks posed by the Codesys runtime (used by over 400 devices from 80 industrial PLC vendors). FieldFuzz leverages efficient network-based fuzzing based on three main contributions: i) reverse-engineering enabled remote control of control applications and runtime components, ii) automated command discovery and status code extraction via network traffic and iii) a monitoring setup to allow on-system tracing and coverage computation. We use FieldFuzz to run fuzzing campaigns, which uncover multiple vulnerabilities, leading to three reported CVE IDs. To study the cross-platform applicability of FieldFuzz, we reproduce the findings on a diverse set of Industrial Control System (ICS) devices, showing a significant improvement over the state-of-the-art.

CCS CONCEPTS

• Security and privacy; **• Computer systems organization → Embedded and cyber-physical systems**;

KEYWORDS

industrial control systems, programmable logic controllers, fuzzing

ACM Reference Format:
Andrei Bytes, Prashant Hari Narayan Rajput, Constantine Doumanidis, Nils Ole Tippenhauer, Michail Maniatakos, and Jianying Zhou. 2023. FieldFuzz: In Situ Blackbox Fuzzing of Proprietary Industrial Automation Runtimes via the Network. In *The 26th International Symposium on Research in Attacks, Intrusions and Defenses (RAID '23), October 16–18, 2023, Hong Kong, Hong Kong.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3607199.3607226

1 INTRODUCTION

Industrial Control Systems (ICS) comprise critical infrastructure such as desalination plants, smart grids, transportation systems, and the nuclear sector. Digital control and communication in ICS are performed by proprietary Operational Technology (OT) devices which suffer from security challenges known to the IT domain (such as parsing bugs for traffic). In addition, OT devices commonly do not employ OS-level countermeasures that are standard practice in IT nowadays [2]. Exploiting such vulnerabilities could have catastrophic physical consequences, such as destroying equipment [36].

At the core of ICS, Programmable Logic Controllers (PLCs) process sensor readings while executing control logic to perform real-time control of physical processes. Engineers write control applications in programming languages defined by the IEC 61131-3 standard, and executed in proprietary runtime environments on the PLCs. The runtime environment, Integrated Development Environment (IDE), and compiler are proprietary software of the device manufacturer. The most significant Original Equipment Manufacturer (OEM) generic framework is the Codesys runtime, which is at the heart of ≈80 industrial device vendors ranging over 400 devices, including manufacturers such as Schneider Automation, Bosch Rexroth, WAGO, and Hitachi Europe [17], representing at least ≈20% of the active PLCs worldwide [38].

Security threats to this ecosystem can be introduced, mainly via the control applications (through their handling of sensor data) and the diverse software components of runtimes. In addition, vendors can extend runtimes with their third-party libraries, exacerbating *supply chain risks*, as the final runtime is a collection of components from various sources. The urgent/11 and ripple20 vulnerabilities [4, 24] are recent examples of critical supply chain security issues in ICS devices. In both cases, low-level network traffic handling libraries contained bugs that allowed privileged remote code execution by the attacker, affecting millions of devices.

Given this, how could third parties (such as operators) test PLC devices and their control logic applications for (security) bugs? For analysis of proprietary IT software, fuzzing has been very successful in recent years. Unfortunately, due to domain-specific challenges, available fuzzers cannot directly be applied to PLC runtimes or their IEC 61131-3 control applications. In particular, PLC runtimes are complex stateful multithreaded applications that interact through proprietary protocols with the environment. Furthermore, control applications must be executed within the runtime, requiring new inputs (via memory-mapped I/O) in each scan cycle. To the best of our knowledge, ICSFuzz [38] is the only reported tool in the literature for fuzzing control applications. However, it suffers from significant drawbacks, such as losses in input delivery synchronization, slow fuzzing speed, manual crash monitoring, and only supporting a specific physical device (a WAGO PLC). Applying traditional fuzzers to this problem, on the other hand, like AFL++, also has limitations as such methods cannot control the runtime, often lack network capabilities, and cannot fuzz *in situ*, i.e., correctly execute the bespoke control applications binaries used in ICS.

This work proposes a novel unified approach for vulnerability discovery throughout the computational stack of PLCs in the Codesys ecosystem (including their control applications). We realize this approach in the FieldFuzz framework, which is the first platform capable of fuzzing all components in a PLC, including the IEC 61131-3 control applications. It leverages reverse-engineered features of the proprietary Codesys platform and protocols to allow fuzzing of the runtime during its execution (in context), and the control applications in the runtime context, providing meaningful crashes. Furthermore, we designed Ghost, our system for obtaining accurate instruction-level coverage statistics, that is injected *in situ* in the closed proprietary PLCs as a control application, allowing easy and multi-platform deployment. Our experiments discovered multiple vulnerabilities that were responsibly disclosed to corresponding device vendors. Three CVE IDs have been assigned to vulnerabilities discovered by FieldFuzz.

In summary, our main contributions are the following:

(1) We reverse the Codesys runtime and its proprietary communication protocol to enable complete remote control over control application execution in the runtime, enabling fuzzing of the control applications and the runtime in their native execution environments, leading to three CVEs.
(2) We demonstrate the feasibility and efficacy of our approach through our implementation and evaluation of FieldFuzz for both the control applications and the runtime. In addition, we introduce a local monitor called Ghost that provides further fuzzing feedback when injected into the target as a control application. As we leverage the standard communication protocols and runtime APIs, our approach allows cross-architecture fuzzing of diverse Codesys targets independent of the target platform and device vendor.
(3) We demonstrate improvement over the state-of-the-art (i.e., ICSFuzz [38]) in control application fuzzing with higher performance, reliable scan cycle control, input delivery, monitoring, and breakpoint-based coverage feedback. FieldFuzz supports fuzzing runtime components that are inherently unreachable by ICSFuzz. In contrast to ICSFuzz, our approach applies to any target which supports Codesys (architecture independent).

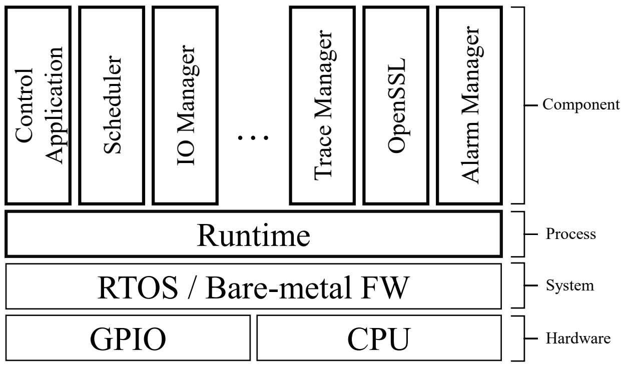

Figure 1: PLC System stack. Blocks with thick borders belong to the proprietary codebase and require reverse engineering to facilitate fuzzing.

2 BACKGROUND AND PRIOR WORK

2.1 PLC Runtimes for Control Applications

Process engineers write control logic using IEC 61131-3 compliant programming languages (such as Ladder Logic) on an engineering workstation. While certain ICS platforms interpret the intermediate representation of the control program, others employ an IEC compiler to produce a compiled control binary. These control binaries differ from known executable formats such as ELF or PE, and cannot execute independently. Therefore, they are loaded and executed in complex PLC runtimes, which control every aspect of the binary execution (see Figure 1).

In general, PLC runtimes, such as the Codesys runtime that we focus on in this work, are complex applications that encapsulate a wide array of functionality almost akin to a fully-fledged operating system. As illustrated in Figure 1, these runtimes operate on top of minimal real-time operating systems or bare-metal PLC firmware. In the first case, the runtime might rely on the functionality offered by the OS, while in the latter case, the runtime is part of the firmware image. Regardless of the runtime deployment method, the runtime usually contains distinct but interconnected components that enable core functionality such as: executing the control application, I/O delivery, network communications, cryptographic functions, logging, event and exception handling, facilitating atomic operations, and more. Some components are solely purposed to interact with an open-source dependency (such as OpenSSL). In Figure 1, we mark the component responsible for communication with OpenSSL as part of the proprietary codebase. The reason behind this is that to provide such interaction, the runtime implements additional logic and wraps the core libraries into independent components of its proprietary format.

2.2 Codesys Environment

Codesys is a multi-platform software that includes the development system and the runtime for target ICS devices. The runtime is a collection of components with a modular architecture implemented as statically compiled and dynamically linked necessary libraries [1], provided by Codesys, the device vendors, or open-source. At the same time, the increase in traditional vulnerabilities in control applications follows the evolution of the support of advanced external libraries [30]. For example, consider `CVE-2020-6081`, which exploits code execution vulnerability in `PLC_TASK` functionality of Codesys runtime 3.5.14.30, triggerable by a specially crafted network packet, enabling remote code execution. Furthermore, the runtime is also vulnerable to other classical vulnerabilities like out-of-bounds read (`CVE-2021-30194`), write (`CVE-2021-30193`), NULL pointer dereference (`CVE-2021-29241`), and more.

2.3 Prior Work

Protocol fuzzing. There is a considerable amount of work on fuzzing network protocols. For instance, AFLNET [34], a greybox fuzzer fed with a mutated corpus of recorded network messages, utilizes state-feedback for guiding the fuzzer, and KiF [3] for fuzzing session initiation protocol. In addition, Pulsar is a stateful black-box fuzzing testing technique for proprietary network protocols that utilize automated reverse engineering and simulation [15]. There are ICS network protocol-specific solutions such as Peach*[29], a coverage-based improvement over the standard Peach fuzzer [16]. PropFuzz [31] monitors the behavior of the controller along with the network connection to detect unexpectedly long jitters in the control process. Polar [28] utilizes static and dynamic taint analysis to identify vulnerable operations with semantic aware mutation to improve the fuzzing procedure. Finally, ICS^{3}Fuzzer [11] is a framework for discovering implementation bugs in the supervisory software by fuzzing the network communication protocol employed to communicate with the field devices. It synchronizes the controls of the GUI operation and network communications to fuzz the entire supervisory software. It should be noted that ICS^{3}Fuzzer fuzzes the supervisor software and is not concerned with fuzzing PLC devices. While FieldFuzz can also do protocol fuzzing (as ICS communication is a component), it focuses on vulnerability discovery of *any* component integrated into the PLC.

Control application security. Research on control application binaries focuses primarily on their safety verification. For instance, Canet et al. [6] employ formal semantics and a model checking tool to verify rich behaviors and properties of Instruction List PLC programs. Other approaches detect malicious inputs to the PLC by verifying against temporal safety properties [13], and monitoring violations in specifications [22]. In VetPLC, Zhang et al. [40] utilize static analysis for creating timed event graphs combined with invariant data traces to detect hidden safety violations. Guo et al. [18] automatically translate control logic into C and perform symbolic execution to generate test cases. On the other hand, AttkFinder [7] uses information flow-guided symbolic execution on an intermediate representation. Keliris et al. [26] reverse-engineered the Codesys v2.3 file format for control application binaries to propose an automated on-the-fly attack formulation. Similarly, CLIK [25] automatically modifies control logic executing on a PLC, and sends false data to the engineering software using captured network traffic.

The closest work on directly fuzzing industrial binaries is ICSFuzz [38], a fuzzing framework for primarily fuzzing control applications. It supplies inputs to the control binary by overwriting the value at the memory-mapped GPIO and collects coverage metrics by overwriting NOP instructions. It detected multiple crashes for a synthetic control application binary dataset and a limited subset of runtime functions. However, as further discussed in Section 3.4, it features numerous significant limitations.

3 FUZZING OF PROPRIETARY INDUSTRIAL CONTROLLERS

This work addresses the following overall problem: *How to systematically and efficiently identify (security) bugs in proprietary PLC runtimes?* As listed below, achieving this goal requires addressing several research questions. Then, we summarize research and engineering challenges for achieving the overall goal.

3.1 Research Questions

- RQ1: How to systematically (and efficiently) test control applications running in situ within a complex proprietary multi-threaded runtime such as the one used by Codesys?
- RQ2: How to guide the in situ testing of proprietary bare-metal runtimes without local access to the device?
- RQ3: Is it possible to generalize the testing approach to be cross-platform?

3.2 Research and Engineering Challenges

Fuzzing the PLC stack is a challenging task for the following reasons:

- PLC runtimes are typically closed-source proprietary software, necessitating black-box fuzzing and requiring extensive reverse engineering.
- Cross-platform PLC runtime rehosting is very difficult due to the complex interactions with hardware and peripherals. Even in the more straightforward case of IoT firmware, recent efforts have only achieved partial rehosting [10, 23]. At the same time, symbolic execution of the runtime and its control applications is also challenging, given the complexity of the binary, leaving fuzzing on the actual (proprietary) hardware as the only option.
- The PLC runtime runs as a root process (on some platforms, as a kernel module), and can be seen as a system-on-a-system since it takes over significant hardware resources (timers, I/Os, etc.) to ensure its real-time operation. Controlling it from a fuzzer is as challenging as controlling a full-blown operating system.

3.3 Threat Model and Assumptions

We assume that the owner of the PLC conducts its fuzzing to identify security vulnerabilities in the runtime or the control application executed within it. The PLC's runtime is available as a binary without debugging information, while the owner can install additional control applications on the PLC (leveraged for injecting the Ghost monitor). The goal is to identify vulnerabilities exploitable by an adversary who can monitor, intercept and modify sensor [39] or network communication to the PLC [26], essentially performing a man-in-the-middle (MITM) attack (e.g., as in Stuxnet [27] and IRONGATE [21]). Other similar research works employ the same assumption, such as TCP veto [19] and CLIK [25].

Table 1: Comparison of FieldFuzz with state-of-the-art for fuzzing control applications and runtime. ○, ◐, and ●represent non-existent, partial, and full support.

Works	Runtime					Control Application							Misc	
	Support	In-context fuzzing	Input delivery	Coverage feedback	Crash monitoring	Support	In-runtime fuzzing	Input delivery	Multi vendor support	Scan cycle control	Coverage feedback	Crash monitoring	Input over network	Bare-metal fuzzing
AFL++ [12]	◐	◐	○	○	○	○	○	○	○	○	○	○	○	○
ICSFuzz [38]	◐	○	◐	○	◐	●	○	◐	○	○	●	◐	○	○
FieldFuzz	●	●	●	◐	●	●	●	●	●	●	●	●	●	●

3.4 Limitations of existing methods

While other approaches can potentially be applied to fuzzing control logic applications, these either fail at addressing the challenges mentioned above or feature significant limitations that hinder efforts towards efficient and reliable in situ fuzzing (see Table 1). Two such approaches are AFL++ [12], a generic general-purpose fuzzer, and ICSFuzz [38], explicitly built for fuzzing control applications.

Vanilla AFL++ cannot fuzz control applications for several reasons. Firstly, control binaries are compiled into bespoke formats meant for execution within the context of their runtimes; thus, AFL++ cannot execute them outside their runtime environment. Moreover, AFL++ cannot control the runtime or the scan cycle, emulate input delivery from peripherals, monitor the control application and runtime for crashes, and fails to perform in-runtime fuzzing for such programs. Additionally, AFL++ generally requires shell access, making it incompatible with industrial controllers where the runtime is distributed as a bare-metal firmware image with no underlying OS or kernel module. Finally, AFL++ lacks network fuzzing capabilities, which prevents fuzzing proprietary PLCs.

ICSFuzz, on the other hand, can fuzz control applications running on PLCs but features significant limitations that hamper fuzzing efficiency. First, it utilizes the KBUS subsystem for input delivery to the control application, necessitating a physical PLC, and making it non-scalable. Due to a lack of scan cycle synchronization, it periodically drops fuzzing inputs and is slow. Furthermore, ICSFuzz lacks automation for observing the state of the control program and involves manual crash monitoring. Finally, it also performs limited stateless fuzzing of the shared library functions of the Codesys runtime on WAGO PLC, discovering some crashes. However, due to the stateless and out-of-context nature of the fuzzing, it misses vulnerabilities requiring the execution context of the runtime.

AFL++ and ICSFuzz cannot directly interact with the runtime by calling its specific functions, nor can they maintain the runtime state. Consequently this also prevents them from fuzzing the entirety of the runtime, limiting their reach and capabilities. FieldFuzz, on the other hand, is created explicitly for fuzzing in the ICS environment and addresses all the above limitations. Its methodology is designed to be particularly generic, and control application fuzzing is just an instance of component fuzzing, as was the case with protocol fuzzing discussed in Section 2.3.

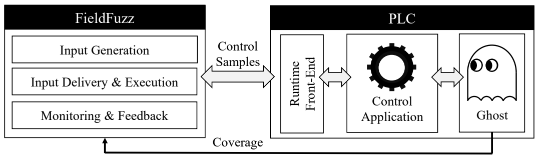

Figure 2: Overview of FieldFuzz. It controls and tests the target control application via the reversed runtime protocol while Ghost obtains additional coverage data.

4 FIELDFUZZ

We address the research questions and reach our goals through a novel custom fuzzing framework, which allows *in situ* fuzzing of the control application (and other components) *within* the context of the PLC runtime. To allow this fuzzing to run on black-box proprietary PLC hardware, we fuzz via the network interface of the runtime (addressing RQ1). To allow for more efficient fuzzing, we inject a custom application, dubbed Ghost, into the target via the control application loading mechanism (addressing RQ2), providing FieldFuzz with fuzzing feedback. Utilizing the standardized (proprietary) network interface should also allow cross-platform fuzzing, which we validate experimentally (addressing RQ3). Figure 2 provides a high-level overview of the proposed system.

4.1 Overview

As illustrated in more detail in Figure 3, our proposed methodology consists of 3 interconnected stages. 1) In the Input Generation stage, we leverage the IDE to communicate with the runtime, reverse engineer its Codesys v3 communication protocol to understand packet metadata, and build an input corpus for fuzzing. This input corpus is further mutated based on the feedback from the Monitoring & Feedback stage. 2) Since FieldFuzz utilizes the network interface to provide inputs to the control application and the runtime, the Input Delivery & Execution stage encapsulates the mutated input into the proper packet structure. This packet is parsable by the runtime and is sent over the network. It fuzzes the control application by delivering inputs and controlling its execution by sending service requests to the runtime for single scan cycle execution. 3) In the Monitoring & Feedback stage, FieldFuzz communicates with the runtime services to monitor the runtime state and collect feedback. Additionally, it receives coverage feedback from our Ghost monitor, injected into the runtime via the control application loading mechanism. Finally, the complete feedback is relayed to the Input Generation stage to generate future inputs and guide the fuzzing efforts.

4.2 Input Generation

FieldFuzz leverages the proprietary communication protocol utilized by Codesys to communicate with the runtime for enabling fuzzing over the network. This protocol facilitates hierarchical device-to-device communication between the engineering software (IDE) and the target devices (PLC, HMI touch panels, Gateways). The flexible nature of the protocol allows its routing in a single

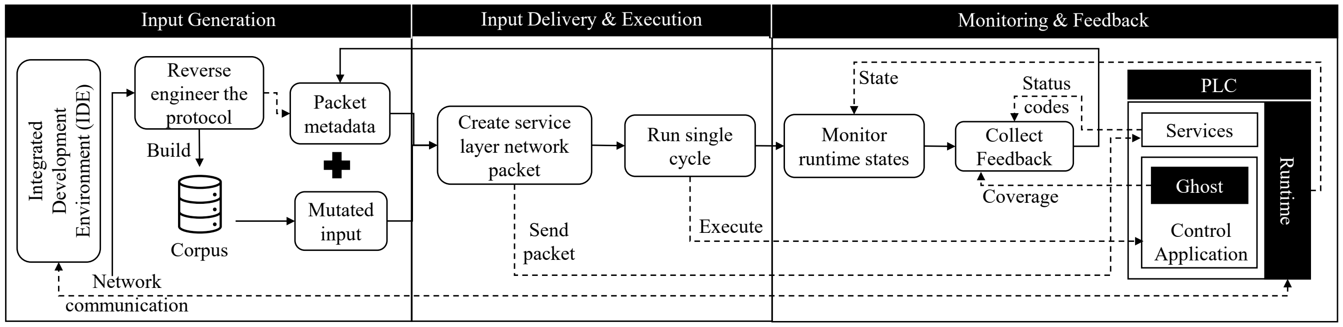

Figure 3: FieldFuzz methodology overview for fuzzing control applications and the runtime.

Table 2: Commands associated with components of our interest at the service layer. In bold, are the commands replicated by FieldFuzz that are crucial for the fuzzing routines. Additional markings correspond to the fuzzing stage: ' I ': Initialization, ' D ': Input delivery, ' E ': Execution control, ' M ': Monitoring.

Cmp	Command	ID	Cmp	Command	ID	Cmp	Command	ID	Cmp	Command	ID
CmpDevice [0x01]	**GetTargetIdent**^{I,M}	0x01	CmpApp [0x02]	DeleteApp	0x04	CmpApp [0x02]	**ReadAppList**^{I,M}	0x18	CmpApp [0x02]	ReadProjectInfo	0x31
	Login^{I}	0x02		Download	0x05		SetNextStatement	0x19		DefineFlow	0x32
	Logout^{I}	0x03		OnlineChange	0x06		ReleaseForceList	0x20		ReadFlowValues	0x33
	SessionCreate^{I}	0x0A		DeviceDownload	0x07		UploadForceList	0x21		DownloadEncrypted	0x34
	ResetOrigin	0x04		CreateDevApp	0x08		**SingleCycle**^{D}	0x22		ReadAppContent	0x35
	EchoService	0x05		**Start**^{I}	0x10		CreateBootProject	0x23		SaveRetains	0x36
	SetOperatingMode	0x06		Stop	0x11		ReInitApp	0x24		RestoreRetains	0x37
	GetOperatingMode	0x07		**Reset**^{I,E}	0x12		ReadAppStateList	0x25		**GetAreaAddress**^{D}	0x38
	InteractiveLogin	0x08		SetBP	0x13		LoadBootApp	0x26		LeaveExecpointsActive	0x39
	RenameNode	0x09		**ReadStatus**^{M}	0x14		RegisterBootApp	0x27		ClaimExecpointsForApp	0x40
CmpApp [0x02]	**Login**^{I}	0x01		DeleteBP	0x15		CheckFileConsistency	0x28	CmpMonitor2 [0x1B]	Read	0x01
	Logout^{I}	0x02		ReadCallStack	0x16		**ReadAppInfo**^{I}	0x29		**Write**^{D}	0x02
	CreateApp	0x03		GetAreaOffset	0x17		DownloadCompact	0x30	CmpPlcShell [0x11]	**Execute**^{M,E}	0x01

industrial network with diverse segments of Ethernet, CAN, Serial, Sercos, and other media. Without loss of generality, FieldFuzz connects to field devices with TCP over Ethernet.

To gain sufficient knowledge for implementing FieldFuzz, we developed a complete Wireshark dissector for the Codesys v3 communication protocol, written in Lua. Using the dissector's parsing and filtering capabilities on the captured traffic, combined with manual reverse engineering of the runtime, and published information [32], we extracted all the information needed to develop FieldFuzz. This also allowed us to collect a corpus of valid interactions with components, and communication patterns required for stateful (maintaining session and runtime state) fuzzing. In Section 5, we discuss and estimate the manual steps that are required to fuzz the runtime components.

The runtime utilizes a proprietary network stack to facilitate network communication between nodes[1], with four layers:

Block layer. This layer is responsible for communication over the physical interfaces. It processes a verification number and the total number of bytes in the packet. It then transfers the rest of the packet to the Datagram layer.

Datagram layer. This layer detects Codesys nodes in the network and routes the packets appropriately. It parses the verification number, hop information structure, a *service_group_id* identifying the destination service, a packet length field, the sender and receiver addresses, and optional padding.

Channel layer. This layer ensures synchronized communication, integrity verification, and delivery acknowledgment. Communications rely on opening, maintaining, and closing communication channels. Beyond data, packets contain an ID that indicates the desired functionality concerning channel management, a flag field, a *channel_id* for identifying the open channel, and packet metadata (e.g., checksums, acknowledgment IDs, etc.).

Service layer. This layer is responsible for querying the requested service and transmitting the operating settings. Among others, the message in this layer contains the *service_group_id* field, which refers to the unique ID that the runtime uses internally to identify available services. The message contains *command_id* to indicate the target command within a specific service. Finally, the message also features session, content, and protocol metadata fields.

We identify the invariant and variable fields in the packet structure based on the packet captures collected from the network communication between the Codesys IDE and the runtime. *Invariant* fields are the metadata information required by FieldFuzz to construct the input delivery packet, such as service group (identifying the requested service), command ID (identifying the requested runtime command), and more. Such fields do not change values between similar categories of requests. On the other hand, *variable*

[1]Detailed information about the network stack is available in [32].

fields whose value changes in the communication are the potential fuzzing inputs. For instance, if a user forces a value change for a variable in the control application, the varying fields will be limited to the user-specified forced input value, address offsets for the variable in the control application memory space, the size of the variable, session identifiers, and more. However, the invariant fields, such as service and command IDs, will remain consistent, making them easily identifiable. Using such a procedure, we build metadata information for requesting particular runtime services, and identify fuzzing input fields. Typically, Service Group IDs (2 bytes) for the vendor-added components specific to a runtime variant are enumerable and within a dedicated range beginning from `0x100`, along with the Command IDs (2 bytes). FieldFuzz sequentially enumerates the 2 bytes of the Command ID and reads the returned status code to determine existing commands. Some of the commands validate the device-level and application-level session or both. The standard status codes indicate which commands do not exist for the enumerated Service Group ID. We save the valid tuples as component interfaces for fuzzing.

Table 2 presents a subset of commands associated with a select few runtime components and is crucial for our fuzzing routines. The commands identified and replicated by FieldFuzz are denoted in bold and marked according to their utilization in different stages of our fuzzing methodology. While these are the commands whose input format is pre-programmed and known to FieldFuzz, we show later in Section 7 that FieldFuzz allows interaction with any component of choice that is reachable from the network by specifying an ID tuple for routing, and the corpus to produce the input.

4.3 Input Delivery & Execution

In literature, bare-metal IO modules are often utilized as the primary communication method for PLCs [14, 38]. However, such an approach is not scalable and is often specific to the model or series of the PLC. Another approach is to utilize the Modbus protocol from the Fieldbus family of network communication protocols. However, such an approach would require the project to include the Modbus client object to receive read and write commands, requiring explicit declarations of exports in the control project, usually disabled by default. Commonly, the HMI displays and modifies inputs to a program using the OPC Unified Architecture (OPC UA) protocol. However, OPC UA support is not present in all Codesys distributions and requires a license; otherwise, the OPC UA server shuts down after 2 hours. Furthermore, using symbolic access for input delivery requires implicit mapping of the global variable list to the project, allowing the operator workstation to access chosen variables by name.

The abovementioned approaches require explicit configuration modification in the control project to support specific Fieldbus protocols and symbolic variables. To address this problem, FieldFuzz utilizes a universal approach and does not require project modification while enabling read and write to any variable regardless of its type and visibility scope. It utilizes *tags*, a nested binary structure to send requests, command payloads, and replies to the service layer of the runtime as defined in its proprietary network stack. Each tag starts with its ID: *tag_id*, which often corresponds to the type of payload or status code, the *tag_size*, *tag_data*, and some *additional_data*. The tag also contains *data_size*; the size of the supplied data, *write_value*; the value to be written, and *write_offset*, which is the offset of the target variable from the start of the data section.

FieldFuzz initiates communication with the runtime over the network using its proprietary protocol while the runtime listens on multiple ports for connection requests (TCP 1217, 11740, and UDP 1740 to 1743). Depending on the requested service layer command, it establishes sessions in at least two stages to reliably invoke commands in the target components: the device, and the application login stage. FieldFuzz utilizes the `CmpDevice` Login command to retrieve a device-level session handle from the runtime, which the runtime associates with the channel in its mapping tables. It then retrieves a list of loaded applications from `CmpApp` component. If the application is loaded on the PLC, FieldFuzz can open an additional session via the `CmpApp` Login command to obtain the application session ID and its handle. The session information is maintained to perform stateful operations with the runtime. Finally, it implements a keepalive mechanism to keep the channel active despite the timeout imposed by the runtime.

Input delivery to the runtime components. To deliver inputs to the control application and other runtime components, FieldFuzz first creates a (*service_group_id*, *command_id*) tuple to identify the appropriate component for routing the input. Next, using the recursive tag encoder algorithm [33], it builds a binary tag structure to place the value into a corresponding field based on the corpus of pre-known Tag IDs and their value formats. The resulting binary structure is included in the body of a Service Layer packet. Next, FieldFuzz calculates a CRC32 checksum for this data, constructs a packet header, and encapsulates it with all the remaining layers.

For delivering the input, FieldFuzz sends this prepared packet containing the fuzzing input to the runtime over the active channel. The Block Layer of the runtime receives the packet, processes it, and passes it to a component that parses the *service_group_id* from the packet header and routes the packet to the target component. Finally, the communication handler of the target component performs sanity checks for the data format and passes the input binary stream to the function that implements the corresponding *command_id*.

It should be noted that since FieldFuzz provides fuzzing inputs to the control application and the runtime components over the network, it needs to maintain the session information relating to the state of the corresponding runtime component. It maintains the state of *blk_id* and *ack_id* counters required for generating valid subsequent packet headers. It also stores the application session ID, and its handle. Furthermore, identifying known states is essential for uncovering vulnerabilities since the components of the runtime are interconnected such that they call functions and access structures belonging to each other. This process provides inputs to target components, enabling stateful component fuzzing.

Input delivery to the control application. Delivering inputs to the control application involves several additional routines. FieldFuzz achieves this by performing a write operation over the network to the control application memory segment. Typically, the runtime separates data and code (as compiled instructions) into separate memory segments, referred to as Area0 and Area3, respectively. FieldFuzz reads and writes variables using relative offset addresses in the Area0 memory segment. For this, it constructs short bytecode programs as inputs for the `CmpMonitor2` component, using a set of 58 opcodes specific to the runtime. Depending on the syntax, the opcodes can read inputs from the interpreter stack and receive up to 4 inline arguments. For implementation, we extract the opcode table from the reverse-engineered runtime binary and map their names and format using a relevant decompiled dynamic library loaded by the Codesys IDE on the operator workstation. The interpreter performs various checks and returns the status code in the first byte of the tag (e.g., wrong pointer, buffer overrun, and more). FieldFuzz detects a successful read operation with the presence of tag `0x40` in the reply, and a failed attempt by the presence of tag `0x41`. In contrast, the write operation requires a more complex bytecode program, and returns no data in reply for a successful attempt. Nevertheless, FieldFuzz can still verify the write (input delivery) operation by detecting the presence of tag `0x41`.

4.4 Monitoring & Feedback

Understanding code coverage is essential to assessing the effectiveness of our fuzzing campaigns. However, this is particularly challenging since the Codesys runtime is proprietary software. As a result, we are forced to utilize blackbox fuzzing on the binary level without access to the source code, which is not straightforward since the runtime employs a binary packing mechanism as an anti-tampering measure, hindering our efforts to locate relevant code blocks corresponding to the components. Additional analysis of the dumped memory regions of the runtime shows that the runtime binary is stripped of its debug symbols during compilation, further increasing the difficulty of locating interesting components for fuzzing.

To address the challenges of visibility into the fuzzed execution of the runtime components, FieldFuzz utilizes a combination of status codes, and a Ghost monitor loaded using the standard control program loading mechanism, to enable and infer code coverage.

Status code for execution feedback. When a component executes a requested command, it returns a value to the callee, indicating the completion status of the requested operation. These internal identifiers are called status codes and differ from Service Group IDs mapped to the components.

As a code path coverage mechanism, FieldFuzz watches for the status codes returned by the different layers of the Codesys v3 network protocol stack and the output of various runtime components, maintaining the status code sequence during the fuzzing session. Based on the status code sequence changes resulting from mutated inputs, it understands and differentiates the execution path inside the component function. The status codes differ as the fuzzing input traverses the lower network layers. Essentially, a different part of the target function returns a different status code due to a change in the execution path. Therefore, whenever fuzzing uncovers a new execution path (by observing the status code), FieldFuzz adds it to the list of known states and initiates a new fuzzing campaign. For example, a reply with only Channel Layer status codes indicates a failure to reach the Service Layer for processing. The status codes can determine whether the command reached the target function or failed due to the lack of authentication, wrong Command ID,

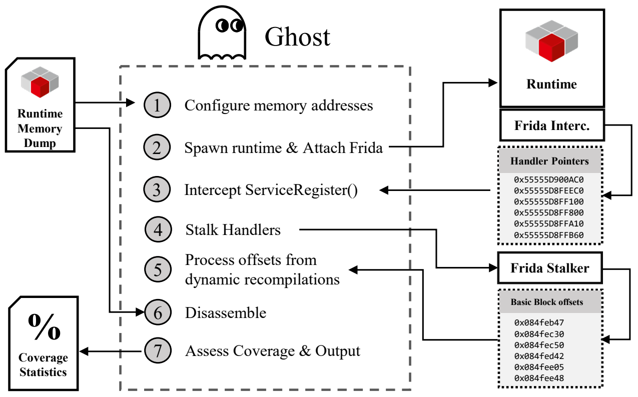

Figure 4: The steps involved in the operation of Ghost.

or incorrect payload format. Some of the notable status codes include those corresponding to missing tags (`L7TagMissing`), and non-existing command groups (`L7UnknownCmdGrp`) or commands (`L7UnknownCmd`).

To retrieve the status of the execution of the control application binary, FieldFuzz surveys the `CmpApp` and `CmpPlcShell` components. Upon exception, a core dump and crash log from `CmpLog` are retrieved from the controller remotely for further investigation. Furthermore, to detect crashes of the runtime, it constantly monitors the latency in the channel and the consistency of the Block Layer counter.

Ghost for code coverage feedback. During the runtime initialization phase, component setup functions make calls to a function used to register runtime components. These calls include memory pointers to the entry points of the runtime components. In order to capture these memory pointers, we leverage Frida[35], a dynamic instrumentation toolkit widely used for reverse engineering. Using Frida, we dynamically intercept calls to the registration function and locate the memory pointers to the component entry point. Then, using these pointers as starting points, we analyze the components and locate all the memory segments relevant to their commands. The use of Frida in our toolchain does not hamper our approach on more limited systems, as it can be compiled for a variety of architectures and utilized through its C API.

Accurate assembly instruction-level coverage requires instrumentation and monitoring of the spawned runtime threads for handling command invocation. Unfortunately, anti-tampering mechanisms hamper attempts at static instrumentation, so we utilize Frida to monitor for calls to component functions. When a call is intercepted, we launch Frida's code tracing engine that dynamically recompiles assembly code and traces the executing thread. By getting feedback from the recompilation process, we can keep track of executed basic blocks using their offsets in memory.

To automate this process, we introduce Ghost, our tool for dynamic black box instrumentation of the Codesys runtime to obtain accurate instruction-level coverage statistics (see Figure 4). Ghost is initially configured with access to the runtime memory dump and the relevant memory segments of the target components. Upon execution, it spawns the runtime and immediately independently attaches a set of scripts to capture all runtime component setup calls

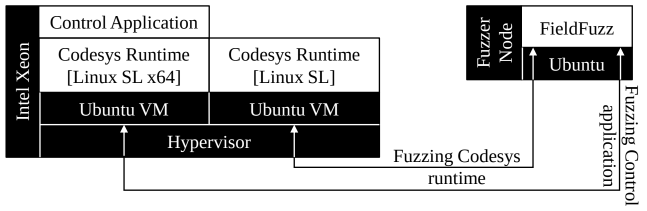

Figure 5: Experimental setup for fuzzing control applications and the runtime.

to avoid Codesys anti-tampering measures that execute on startup. Having captured the pointers to component functions, Ghost utilizes Frida to capture calls to them and consequently traces the runtime threads executing component code. While fuzzing with FieldFuzz, Ghost receives the notifications of dynamic recompilation events with their memory offsets, indicating the current execution of a basic block. Ghost then disassembles the relevant memory segment and keeps track of the executed assembly instructions. Upon exit, it saves the context information for component functions, a log of the dynamic recompilation events, and extensive instruction-level coverage information.

4.5 Protocol Dissector

To gain sufficient knowledge for implementing FieldFuzz and to support the fuzzing campaigns, we have developed a complete Wireshark dissector for the Codesys v3 communication protocol. The full-featured dissector for Codesys v3 protocol enables the use of Wireshark with advanced filter expressions and custom columns for every aspect of the Codesys v3 stack. Based on the knowledge gained in this work, we also provide the dissector with symbols to translate the packet data into human-readable component, command and field names. The dissector significantly streamlines the process of input corpus collection, crash investigation and exploit development. It is implemented using the Wireshark Lua API and provides the following capabilities:

(1) Live decoding of the network traffic capture (IDE-to-PLC, PLC-to-PLC, Gateways, HMIs)
(2) Analysis of the pre-recorded dump files
(3) Detection of the Codesys network stack on non-standard ports using magic constants
(4) Using advanced filter expressions for all layers of the Codesys v3 stack
(5) Using custom columns for all layers of the Codesys v3 stack
(6) Choosing between the use of human-readable symbols and raw values for filters and column data or combining both
(7) Direct access to the Layer 7 binary tag tree and recursive extraction of the payloads

5 FUZZING CAMPAIGNS

To address RQ1 and RQ2, we apply the FieldFuzz framework to fuzz the Codesys runtime and control application binaries. An overview of the binaries is provided in Sections 6 and 7, respectively.

Table 3: Components and service groups at the service layer.

Component	ID	Component	ID
CmpAlarmManager	0x18	CmpLog	0x05
CmpApp	0x02	CmpMonitor	0x03
CmpAppBP	0x12	CmpMonitor2	0x1B
CmpAppForce	0x13	CmpOpenSSL	0x22
CmpCodeMeter	0x1D	CmpSettings	0x06
CmpCoreDump	0x1F	CmpTraceMgr	0x0F
CmpDevice	0x10	CmpUserMgr	0x0C
CmpFileTransfer	0x08	CmpVisuServer	0x04
CmpIecVarAccess	0x09	CmpPlcShell	0x11
CmpIoMgr	0x0B	SysEthernet	0x07

5.1 Fuzzing Setup

Figure 5 shows the basic experimental setup for fuzzing the Codesys runtime and control application binaries. We utilize two virtual machines with the Codesys runtime variant for Linux, which run on the Intel Xeon-based hypervisor server. The runtime includes a standard init.d wrapper that facilitates the automatic runtime restart after a crash caused by FieldFuzz. In addition, we disable the system-wide address space layout randomization (ASLR) on these virtualized nodes to simplify the crash investigation.

Scalability options. It should be noted that FieldFuzz supports directly fuzzing physical devices, however, these are optional. FieldFuzz can utilize multiple virtual machines to scale the fuzzing setup. On the other hand, another potential possibility to scale the experiment in a single VM is to utilize the ability of the channel layer of the runtime to handle multiple channels simultaneously, a functionality supported by FieldFuzz. While FieldFuzz primarily uses TCP over Ethernet, it can fuzz any node in the plant by relaying the packets through other devices, including the nodes of the network that are not reachable by Ethernet, such as those connected by serial interface or CAN bus.

Estimation of manual efforts. Assuming that the operator is already familiar with the FieldFuzz Framework and the Wireshark dissector, the manual efforts required to derive an input corpus and setup fuzzing for a typical standard component of the runtime can be roughly estimated as one hour of work. If no traffic capture is available, an input corpus can be bootstrapped through limited reverse-engineering of the component packet handler in the runtime. In our experience during reverse engineering the runtime, we found that packet handler code is quite structured and deriving sufficient information from the binary to produce correct messages can be done in a few hours.

Identifying fuzzing runtime targets. As discussed earlier, the runtime is a collection of components (including the component responsible for executing control binaries), so fuzzing the runtime implies interaction with the components responsible for its functionality. However, despite the runtime having a single generic codebase, its actual builds can significantly vary based on the target architecture, vendor modifications, and hardware platform constraints. Therefore, the first step is to create a complete list of all instantiated components. To achieve that, we start by extracting the interfaces of components reachable from the network. The *component interfaces* are defined as the tuple: (Service Group, Command). First, we identify the runtime components loaded by the particular runtime from the boot log of the device and its device description file used by the IDE. Next, we identify the Service Group IDs of the loaded components. For the generic set of components developed by Codesys, we get this information from the decompiled libraries of the IDE and the captured network communication. Table 3 presents a subset of components in our target runtime variant.

Fuzzing inputs. To collect a dataset of valid inputs, we trigger commands with the Codesys IDE and capture its communication with the runtime. We extract Service layer payloads and decode the nested binary tag structure using our Wireshark dissector. We then save the tag IDs, structure, and valid payloads for each (Service Group, Command) tuple. Next, we determine the packet fields derived from the session identifier to identify the variant fields for maintaining state information. Finally, we save these inputs as seeds with the identified format for the current runtime distribution. After creating an initial corpus of input seeds, we utilize python bindings for libradamsa [20] mutators ported from AFL++ [12] to mutate the byte payload. We generate byte blocks of variable length to pass them inside the tags without mutating the tag ID and preserving the remaining structure of the corpus. To prevent fragmentation of the packets that carry the generated payloads, we calculate the maximum length constraint for all tags in order to not exceed the fragmentation threshold value of the runtime (512 bytes in the default installation) cumulatively, including the layer headers. We apply the set of techniques provided by the mutation engine of libradamsa in default execution mode (including Bit and Byte Flipping, Block Swapping, Value Substitution, Byte expansion) to form unexpected data inside the byte blocks. It should be emphasized that mutations are semi-random and are not currently guided by coverage (which is a topic of future work). FieldFuzz then delivers the input to the runtime over the network.

Code coverage. The Ghost monitor calculates coverage on the component, handler, and command level (see Section 4.4). On the component level, coverage includes all the memory segments associated with a component, complete with the command code and its entry-point function. Entry function coverage indicates the percentage of instructions executed within the entry function for running the various component commands. Finally, command coverage is limited to the memory segments relevant to each command.

Deployment. Performing the above steps and preparing to run Ghost to obtain coverage information can be particularly challenging on PLCs that do not offer shell access. We work around this by exploiting functionality readily available within Codesys by embedding the relevant binaries into the control application project such that it is downloaded onto the local file system while loading the control application binary on the PLC. It also loads the Ghost deployment script written in Structured Text employing the native `SysFileCopy` API (available in the `SysFile` library) to relocate the binary files in the file system. Since the runtime runs with root privileges, we utilize `SysFileOpen` and `SysFileWrite` to modify the `CODESYSControl.cfg` runtime configuration file by appending `[SysProcess]Command=AllowAll` to it for enabling arbitrary shell command execution. Having done this, we restart the runtime through the IDE or reboot the target PLC to force the configuration changes. Upon reboot, we use `SysProcessExecuteCommand2` (part of the `SysProcess` library) to perform the necessary setup and run Ghost with root privileges.

Table 4: Code coverage recorded while fuzzing commands belonging to the Codesys Trace Manager component.

Command	Command ID	Command Coverage
RecordAdd	0x0D	95.14%
PacketCreate	0x02	56.17%
PacketClose	0x06	97.56%
PacketComplete	0x04	97.56%
PacketStart	0x0A	97.56%
PacketRead	0X07	26.98%
PacketStop	0X0B	97.56%
PacketGetConfig	0X0F	71.54%
PacketOpen	0x05	92.5%
PacketReadList	0X01	90.48%

As a proof of concept, we run fuzzing campaigns on functions of three standard components: `CmpTraceMgr`, `CmpPlcShell`, and `CmpDevice`. To derive the input corpus and initialize the campaigns, approximately one hour of manual work was required per each fuzzed component. FieldFuzz was able to uncover a variety of crashes, which we then analyzed, focusing on uncovering vulnerabilities. For brevity, we provide extensive discussions for one CVE and shorter discussions for the others. It should be noted that the CVEs found through our FieldFuzz campaigns could not have been discovered using other state of the art fuzzers like ICSFuzz. This is because the fuzzing process requires interacting with runtime components which cannot be reached by these fuzzers.

5.2 Fuzzing CmpTraceMgr Component (CVE-2022-22514)

Setup. To investigate the crash, we use FieldFuzz to generate a standalone exploit from a template. It connects with a SoftPLC (VM) node with full-featured debugging capabilities and sends the service layer payload to the Codesys runtime. To observe critical runtime errors, we enable core dumps and disable the error handling behavior of the `SysExcept` component that intercepts POSIX signals from the runtime for internal interpretation. We modify `CODESYSControl.cfg` to disable the internal exception handler and instruct the runtime to append the logs to a file with a permissive log filtering mask. To record core dumps, we launch the runtime binary (`codesyscontrol.bin`) as a standalone process outside its `init.d` service wrapper and provide it with a `-d` flag for detailed logging.

Crash analysis. FieldFuzz reported a crash for service group `0x0F`, command ID `0x0D`, belonging to the `CmpTraceMgr` component. Our examination of the original pcap file with the dissector revealed that the `recordAdd` (`0x0D`) command (with a 148 bytes payload) causes the crash. This payload incorporates three levels of nested binary tags.

The `CmpTraceMgr` component consists of eight critical operations which are triggered by the service layer commands in sequence. This default component is available in most full-featured distributions of the runtime. It is a backend for the Trace program organization unit object used in control application projects for recording and visualizing variable trends in the physical process. Here, the `recordAdd` operation causes `SEGFAULT` because when the command is sent out-of-order, the component enters an unexpected state operating on a pointer to a structure of a packet object that is never correctly initialized. The offset calculation into this non-existent structure is controlled by FieldFuzz input. An adversary can exploit this offset, forcing the runtime to perform `mov` operations on invalid memory addresses.

Call stack investigation. At least 12 network stack functions handle the packet before it finally reaches the function related to `CmpSrv`, which is the top component of the network stack. Finally, `CmpSrv` calls an exported hook function of `CmpTraceMgr`, which acts as a handler for all service layer commands for the service group `0x0F`. The hook function extracts the command ID from the packet header and jumps into the condition based on command `0x0D`. Functions imported from the `CmpBinTagUtil` component parse the fuzzing input, and decode the 17 binary tags, including tag `0x40` which influences calculations of a memory address offset, the value for which is controlled by FieldFuzz. Consequently, a `SIGSEGV` occurs in the command handler function for the `recordAdd` command, caused by a `mov` instruction attempting to access the nonexistent memory address. The corrupted memory offset is from a structure that stores a tracing packet derived from the value supplied by FieldFuzz.

Status code example. The component changes its returned status codes based on the multiple execution path conditions. The `recordAdd` function does several sanity checks for the supplied value. For example, a reply containing the tag `0xFF7F` with a status code `0x02` is caused by the payloads in tag `0x40` that are outside the expected range, such as `0x0` and `0xFFFFFFFF`, preventing the crash by sanitizing inputs before reaching the vulnerable instruction. Another state of the component, indicated by the returned status code `0x11`, neither causes a crash nor forms a successful trace packet processing result. In this case, the payload falls into the allowed range and passes the elementary entry checks of the `recordAdd` function. This input reaches the vulnerable read operation; however, the exception is handled, and the component returns a packet without any data.

Coverage. Table 4 presents the coverage reported by Ghost during fuzzing campaigns on the `CmpTraceMgr` component. Our fuzzing strategy obtained high coverage for the majority of the component commands. In the cases of `PacketCreate` and `PacketRead`, upon examining the relevant instruction blocks, low coverage can be attributed to a lack of generation of stateful sequence command events during input mutation. Employing stateful input mutation strategies can improve the coverage for such command [5, 8, 41]. However, input mutation strategies are orthogonal to our work and will be explored in future research.

This vulnerability has been reported to the vendor, and was assigned CVE-2022-22514.

5.3 Fuzzing CmpDevice Component (CVE-2022-22508)

`CmpDevice` is an essential component responsible for the authentication and network discovery of the PLC. It uses the `SetNodeName` (`0x09`) command for changing an identification string employed for in-network discovery and initiating a connection with the PLC. Unfortunately, a specially crafted packet sent to the runtime prevents the IDE from communicating with the PLC, resulting in a connection error. Moreover, this issue is persistent even across reboots because the payload from the network packet ends up in a persistent runtime configuration file and keeps restoring upon device boot. The vulnerability was reported to the vendor, and `CVE-2022-22508` was assigned.

The runtime becomes unresponsive due to a specially crafted packet sent to Service Group `0x01` (`CmpDevice`), command `0x09` (`SetNodeName`), with tag `0x58`, and a long bytestring consisting of non-printable characters as a service layer payload. This crafted bytestring is not sanitized properly by `CmpDevice` before being passed to the local `SysTarget` component (potentially vendor-specific, we tested on official Codesys distributions), and then stored permanently in the `NodeNameUnicode` property field. The device is not accessible even after a reboot because the node identifier is appended to the `CODESYSControl.cfg` configuration file as a new record. `CmpSettings` processes this file which is then consumed by `SysTarget`. The connecting client attempts to perform device discovery through the channel layer and calls `CmpDevice` again to perform `GetTargetIdent` and `CreateSession` commands. System log messages suggest that `CmpNameServiceServer`, a channel layer component that exports its functions to `CmpRouter` and implements a Codesys-specific naming system protocol [32], processes the bytestring. Consequently, the device fails to respond to further scan requests, and the dynamic libraries of the Codesys IDE raise several exceptions. Manual removal of the `SysTarget` section from the runtime configuration file restores the operational state of the device after a reboot.

5.4 Fuzzing CmpPlcShell Component (CVE-2022-22507)

`CmpPlcShell` is a default built-in component that fetches information from the device, such as firmware revision and system load. It can also perform system diagnostics of the device by sending string commands of a particular format. An adversary can trigger a segmentation fault, crashing all the runtime threads by sending a specially crafted payload from the Codesys v3 network stack. The vulnerability was reported to the vendor, and `CVE-2022-22507` was assigned.

The main command body is passed inside tag `0x10`, while an additional tag `0x12` is required by some commands for handling the arguments. FieldFuzz detects the crash for the tag `0x12` because the runtime performs a memory read operation outside valid memory boundaries. By sending a sequence of packets, it is possible to force the runtime to perform memory access operations and enumerate the valid address range with the offset increments. As the offset grows in each operation by an internal loop, an unhandled `SIGSEGV` fault occurs once the operation exceeds valid memory boundaries.

6 CROSS-ARCHITECTURE GENERALIZATION

To address RQ3, we discuss the cross-architecture generalization of our approach. While the runtime has a single generic codebase, specifics for each target platform and architecture are reflected in different build variants. For instance, on platforms driven by the VxWorks real-time operating system (RTOS), the entire Codesys runtime is shipped as a kernel module. The embedded bare-metal runtime variant has a much smaller set of components but implements more complex system components to interact with the

Table 5: Codesys runtime binaries for different targets.

Device	Arch	Size (MB)	Packed
WAGO PFC200	arm32	4.6	✗
BeagleBone Black	arm32	5.8	✓
Linux SoftPLC	x64	9.7	✓
Raspberry Pi	arm32	5.5	✓
SIMATIC IOT2000	x32	6.4	✗
emPC-A/iMX6	arm32	6	✓
Windows RTE	x32	103.9	✓

hardware. In more modern ICS devices powered by RTLinux (such as WAGO Touch Panel 600 series or WAGO PFC200 PLC), the runtime operates as root in the userspace and reuses resources provided by the OS, such as network sockets, timers, and file descriptors.

Runtime binaries. As shown in Table 5, the size of the runtime binaries varies across various architectures from 4.6 MB to 103.9 MB. This is because the number of components and shared libraries linked statically or dynamically differs across different variants. In addition, some binaries are packed, involving license management and anti-tampering mechanisms. Our primary distribution of choice in this work (Codesys Control for Linux x64) employs a packing mechanism that we reverse dynamically by dumping the memory segments of the live process. On the other hand, the runtime variant for Windows devices (Codesys Control RTE x32) includes custom renamed and encrypted sections. From the section names and the protection function, we have noticed that these are managed by Wibu-Systems CodeMeter protection software [37], which has also been used by Siemens and Rockwell [9].

SysMem, SysSocket, and SysCom are some critical hardware-dependent components in the runtime. At some point in the execution path, other components rely on the exported functions provided by these lower-level system components, which can influence the behavior of the crashes. Therefore, to assess the applicability of our findings, we test the attacks against different runtime variants by employing physical devices, as shown in Figure 6. We utilize a replayer node that initiates communication with the Gateway. The latter forwards the communications to multiple platforms in parallel. In this case, the Intel Xeon server acts as a VM hypervisor and the Gateway to WAGO PFC200, Raspberry Pi 4, and Odroid C2. This setup enables FieldFuzz to quickly test the same payload across multiple architectures and variants of the runtime. We observe the differences in the behavior of the crashes to adjust the input payload and port it between architectures. As a proof of concept, we replay the fuzzing inputs for crashing the CmpTraceMgr component (CVE-2022-22514)), which is available on all of the tested devices. The payload corresponding to the input is passed through the tag 0x40 and is four bytes long. On an x86 system with ASLR disabled, the crash input causes a SIGSEGV. However, with ASLR enabled, replaying the same value does not lead to a stable SIGSEGV because the resulting offset in the recordAdd function in most of the trials points to an unexpected but valid memory address. As a result, the command function of the component returns a status code (0x11), preventing the crash. On an x64 system, even enumerating the entire 4-byte range did not cause a crash. Nevertheless, such runtime variants accept longer payloads (8 bytes), eventually leading to the crash. The payload behaves identically on Intel and ARM, causing crashes on both the VM and physical devices; it only differs between 32 and 64 bit architectures of the target device.

7 FUZZING CONTROL APPLICATION BINARIES

We are now ready to demonstrate the application of FieldFuzz to the main goal of control application binary fuzzing. The compiled control application runs in the thread spawned by the SysTask component, which is not exported to the network and thus cannot be influenced directly. Instead, FieldFuzz fuzzes the binary inside the runtime context by controlling its execution through the CmpApp component. The latter offers complete control over start, stop, cold reset, and single-cycle operations with the runtime. Table 2 shows the commands replicated for CmpApp.

To set up the experiment, FieldFuzz logs in to the device and starts the control application. Next, it takes over the execution control of the application while providing fuzzing inputs for every scan cycle. Since FieldFuzz has complete control over the scan cycle, it does not drop any inputs due to a lack of synchronization. Finally, it logs crash inputs based on the status feedback received.

Synthetic binaries. We use the same dataset of synthetic applications as used in ICSFuzz [38] for performing the experimental evaluation of FieldFuzz. The dataset comprises control applications written in Structured Text that include introduced vulnerabilities in their imported functions, such as buffer overflows and out-of-bounds write. These vulnerabilities exist due to missing bound checks in imported IEC 61131 library functions. Thus, the family of synthetic applications labeled in the dataset as bf_mmove can cause a buffer overflow under certain conditions due to insufficient buffer size validation before calling a SysMemMove library function. Similar to the control application, this library is written in Structured Text. By looking deeper into its implementation in the runtime, we observe that SysMem component of the runtime provides the backend for this library and is implemented in C. The call of this wrapper, initiated by the control application, ends up in C code which triggers the native memmove function. For this reason, the crash in a vulnerable control application causes the failure of its thread and affects the entire runtime process (running with root privileges). Out-of-bounds write vulnerabilities involve an uninitialized array with a variable index manipulated to write at an arbitrary location. The numbers in the names of the vulnerable binaries correlate with the complexity of the code. For instance, bf_mmove_1 is the simplest initialization of the SysMemMove, while bf_mmove_12 consists of multiple loops and conditional branching statements.

Table 6 shows the results of fuzzing the control application binary and its comparison with ICSFuzz. As the table demonstrates, on average FieldFuzz is ≈4.1x and ≈8.3x faster for arm32 and x64 (Intel) runtime variants, respectively, compared to ICSFuzz. The performance advantage of FieldFuzz comes from the communication protocol-based input delivery and complete control over the scan cycle. On the other hand, ICSFuzz incurs high latency and drops inputs during fuzzing when it misses the scan input cycle of the runtime. It should be noted that the number of crashes reported in Table 6 are a result of intentionally introduced vulnerabilities in the synthetic control applications used for performance evaluation. They concern solely the control applications, not the runtime itself.

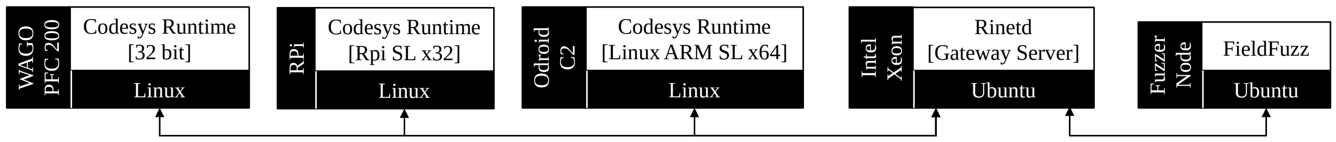

Figure 6: Experimental setup for cross-architecture validation of the fuzzing results.

Table 6: Performance comparison of FieldFuzz against ICSFuzz. The execution speed metrics for FieldFuzz and ICSFuzz correspond to employing one VM and physical PLC, respectively. Markings: 1 - BeagleBone (ARM), 2 - Linux x64 (Intel), 3 - Wago PFC100 (ARM).

Control Applications	Execution Speed (inputs/sec)			First Crash (seconds)			First Crash (inputs)			Crashes		
	FieldFuzz		[38]	FieldFuzz		[38]	FieldFuzz		[38]	FieldFuzz		[38]
	A8[1]	x64[2]	A8[3]	A8[1]	x64[2]	A8[3]	A8[1]	x64[2]	A8[3]	A8[1]	x64[2]	A8[3]
bf_mcpy_1	294.6	593	70.88	0.014	0.25	234	6	148	15270	8289	7876	32
bf_mcpy_6	286.4	642.4	64.2	0.071	1.43	188	22	898	12172	290	2384	21
bf_mcpy_8	244.6	645.6	66.06	21.34	7.08	279	5124	4566	18216	145	359	17
bf_mcpy_12	320.6	526.2	62.11	0.584	1.95	426	181	999	26645	847	977	9
bf_mset_1	223.3	560.6	64.56	0.027	0.04	208	8	22	13441	22200	18105	21
bf_mset_3	268.6	571.2	62.68	0.063	0.03	174	8	17	10906	21723	16085	24
bf_mset_5	289.3	503.2	68.8	0.008	0.56	254	2	281	17554	4447	4373	16
bf_mset_9	314.3	584.8	69.76	-	74.53	623	-	43216	43530	0	25	7
bf_mmove_1	291.6	660.2	64.63	0.025	0.005	176	1	2	11245	20006	16749	28
bf_mmove_4	245.3	578.2	63.1	0.008	0.003	159	1	1	10070	20146	15165	24
bf_mmove_7	232	573	66.31	0.007	0.005	229	1	3	15317	17010	14493	15
bf_mmove_12	257.3	508.2	64.53	-	182.14	783	-	92456	50643	0	15	6
oob_1_arr_1	278	598.8	71.86	2.06	0.14	55	556	83	3880	6121	6291	39
oob_1_arr_6	308	591	77.03	0.027	1.39	103	14	821	8085	5541	6600	28
oob_1_arr_9	284.6	571.2	69.78	-	273.8	105	-	155938	7326	0	12	27
oob_1_arr_13	297.2	507	75.2	12.11	97.86	207	3564	49165	27241	254	686	19
oob_2_arr_1	298.6	520.8	73.53	-	154.42	117	-	80080	8558	0	12	35
oob_2_arr_5	326.6	520.4	71.1	-	155.62	165	-	80662	22759	0	16	27
oob_2_arr_8	295.5	592.64	69.8	-	102.97	188	-	60384	13366	0	12	22
oob_2_arr_13	312.25	502.2	70.95	-	97.86	192	-	48694	13401	0	17	19
Average	**283.43**	**567.53**	**68.34**	**2.8**	**57.6**	**243**	**729.85**	**30921**	**17481**	**6351**	**5512**	**22**

For the scope of this evaluation, we perform one run for each fuzzing campaign. Each campaign targets a distinct vulnerability in the control application binary. To provide metrics comparable to ICSFuzz, we do not perform additional grouping or deduplication of these crashes. We record the time of the first occurred crash in the campaign and increment the number of subsequent occurring crashes to produce the total metric within a one-hour time window.

It should also be emphasized that the measurements in Table 6 are extracted for a single fuzzing instance for FieldFuzz. ICSFuzz requires a vendor-specific KBUS IO subsystem for input delivery, bounding itself to a physical device. Therefore ICSFuzz requires a physical device for fuzzing, which limits its scalability. On the other hand, FieldFuzz can parallelize fuzzing sessions by simply spawning multiple VMs.

Furthermore, FieldFuzz detects considerably more crashes than ICSFuzz, allowing it to cover a wider input space. On average, it detects ≈291x and ≈262x more crashes for the arm32 and x64 runtime variants, respectively, in the same one-hour fuzzing period. However, FieldFuzz detects fewer crashes for a select few samples across both variants. As mentioned previously, while higher-level components normally originate from exact same codebase, low-level system components can differ across various devices and architectures. We have observed that variants of memory management and exception handling implementation in device-specific system components can cause differences in crash behavior. For example, in our 32 bit runtime variant, we observe that the `SysMem` component prevented the runtime from crashing for some samples and instead wrote "Operation not permitted" in the logs, successfully sanitizing the input.

8 DISCUSSION AND LIMITATIONS

Evaluation limitations: During the evaluation, we compared FieldFuzz to ICSFuzz since it is the only state-of-the-art work specifically targeting ICS devices. However, While ICSFuzz was designed to run on the device itself, FieldFuzz interacts with remote devices over the network. Due to this architectural difference, it is not possible to compare the performance on the same hardware platform from the computational efficiency point of view. Therefore, in this evaluation, we aim to compare these two systems in their ability to discover vulnerabilities in the control programs and the runtime by common metrics. Moreover, as the Ghost coverage mechanism injects logic and operates on the fuzzed device itself, it can potentially consume significant memory and compute resources of these constrained devices. This adds a potential risk to meeting the real-time constraints of the PLC operation when fuzzing custom component functions with a larger codebase. A more thorough performance evaluation of this aspect is required and is a direction for future work. Next, the scope of our evaluation included a single run per fuzzing campaign (runtime component or control application binary) and did not address the potential run-to-run performance variety of the fuzzer. Finally, in the chosen metrics and the evaluation approach, we have mainly targeted to produce the metrics that are compatible with ICSFuzz, as was discussed in Section 7.

Runtime security mitigations. The latest runtime version enables the User Management feature by default, thwarting unauthorized login into the PLC. However, out-of-the-box credentials are the default unless manually changed, while the communication is not encrypted. The runtime also expects the client to perform the `Login` action with `CmpDevice` for establishing a session, but this process does not involve actual authentication. Furthermore, the security mitigation properties of the runtime executable differ among platforms. For instance, the WAGO PFC200 PLC (with firmware 03.00.39(12)) used in our setup contains the runtime that is compiled without all of the typical exploit mitigations (no Relocation Read-Only (RELRO), stack canary, NX bit, or Position Independent Executable (PIE)). Finally, the monitoring bytecode interpreter, involved with input delivery to the control application by performing extensive memory operations, applies its own memory access checks. For each execution, before loading the bytecode, the interpreter sets a canary to ensure the integrity of the stack. However, we found that this canary has a fixed value of `0x5AF096A5` regardless of the target platform, which defeats its purpose.

Runtime component coverage. Maximizing component coverage through different mutation strategies is orthogonal to our work and can be explored as a future research direction. Methodologies such as SNOOZE [5], program-adaptive mutational fuzzing [8], and PAVFuzz [41] can be integrated with FieldFuzz to improve coverage.

Black-box fuzzing challenges. FieldFuzz does not require access or any modifications to the controller, ensuring the universality and scalability of the proposed approach. However, this incurs limited code coverage information. We rely on the retrieved status codes to partially address this for runtime components for understanding the execution path. We have found that the debugging capabilities of the full-featured VM can emulate the functionality of the service layer without requiring actual network transmission. This requires pre-loading a harness as a shared library into the runtime and hooking the authentication and packet processing functions in the runtime process. This approach builds a more traditional and comprehensive fuzzing approach combined with full-featured code coverage. However, in the context of ICS, such a white-box fuzzing approach has substantial limitations:

(1) The compiled harness and fuzzing instance is tied to one specific target platform (architecture), while some vulnerabilities are platform-specific, reducing generalization.
(2) This approach is possible with a SoftPLC build of the runtime on top of a typical desktop-grade VM. However, real-world COTS devices hardly have such extensive debugging and instrumentation capabilities.
(3) Full shell access for controlling the device is rare, as many ICS devices embed the runtime on legacy RTOS or use bare-metal runtime variations. Gaining white-box fuzzing capabilities would require re-flashing the controller with a modified kernel image and relying on remote debugging.

9 CONCLUSION

This paper presents FieldFuzz – a fuzzing framework for control applications and industrial runtimes, capable of discovering vulnerabilities in over 400 known ICS devices from 80 industrial device vendors. It facilitates efficient network-based fuzzing by i) reverse-engineering enabled remote control of control applications and runtime components, ii) automated command discovery and status code extraction via network traffic and iii) a monitoring setup to allow on-system tracing and coverage computation. It is the first fuzzer to be able to fuzz control applications in situ, within their execution context of the runtime, even on target blackbox devices. We successfully fuzz the various runtime instances (on different architectures and by different vendors) of the Codesys runtime, reporting three CVEs. In addition, FieldFuzz achieves a speedup of ≈8.3x compared to the state-of-the-art for control application binaries and an increased crash discovery of ≈291x and ≈262x for 32 and 64 bit runtime variants, respectively. We perform fuzzing on physical and virtualized ICS devices to demonstrate automation capabilities, reliability, and performance improvements against the current state-of-the-art. With FieldFuzz, we provide researchers with a robust open-source framework to enable future research in this direction.

REPORTED CVES & PUBLISHED TOOLS

As a result of this work, our reported vulnerabilities were assigned `CVE-2022-22514`, `CVE-2022-22508`, and `CVE-2022-22507`. We release FieldFuzz, Ghost, and the Wireshark dissector for Codesys v3 protocol as open source tools with this work.

ACKNOWLEDGMENTS

Constantine Doumanidis has been supported by the NYU Abu Dhabi Center for Cyber Security, and Prashant Rajput has been supported by the NYU Abu Dhabi Global PhD Fellowship.

Jianying Zhou's research is supported by the National Research Foundation, Singapore, under its National Satellite of Excellence Programme "Design Science and Technology for Secure Critical Infrastructure: Phase II". Any opinions, findings and conclusions or recommendations expressed in this material are those of the author(s) and do not reflect the views of National Research Foundation, Singapore.

REFERENCES

[1] 2021. CODESYS Runtime (Brochure). https://www.codesys.com/products/codesys-runtime/control.html [Online; accessed 25. Jan. 2023].
[2] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. 2019. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of European Symposium on Security and Privacy (EuroS&P)*. IEEE, 31–46.

[3] Humberto J Abdelnur, Radu State, and Olivier Festor. 2007. KiF: a stateful SIP fuzzer. In *Proceedings of the 1st international Conference on Principles, Systems and Applications of IP Telecommunications*. 47–56.

[4] Armis. 2019. URGENT/11 – 11 zero day vulnerabilities impacting billions of mission-critical devices.

[5] Greg Banks, Marco Cova, Viktoria Felmetsger, Kevin Almeroth, Richard Kemmerer, and Giovanni Vigna. 2006. SNOOZE: toward a Stateful NetwOrk prOtocol fuzZEr. In *International conference on information security*. Springer, 343–358.

[6] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebelen. 2000. Towards the automatic verification of PLC programs written in Instruction List. In *Proceedings of IEEE international conference on systems, man and cybernetics*, Vol. 4. 2449–2454 vol.4. https://doi.org/10.1109/ICSMC.2000.884359

[7] John H Castellanos, Martin Ochoa, Alvaro A Cardenas, Owen Arden, and Jianying Zhou. 2021. AttkFinder: Discovering Attack Vectors in PLC Programs using Information Flow Analysis. In *Proceedings of International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*. 235–250.

[8] Sang Kil Cha, Maverick Woo, and David Brumley. 2015. Program-Adaptive Mutational Fuzzing. In *2015 IEEE Symposium on Security and Privacy*. 725–741. https://doi.org/10.1109/SP.2015.50

[9] CISA. [n. d.]. CodeMeter US-Cert. https://us-cert.cisa.gov/ics/advisories/icsa-20-203-01 [Online; accessed 20. Aug. 2021].

[10] Abraham A Clements, Eric Gustafson, Tobias Scharnowski, Paul Grosen, David Fritz, Christopher Kruegel, Giovanni Vigna, Saurabh Bagchi, and Mathias Payer. 2020. HALucinator: Firmware Re-hosting Through Abstraction Layer Emulation. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, 1201–1218. https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[11] Dongliang Fang, Zhanwei Song, Le Guan, Puzhuo Liu, Anni Peng, Kai Cheng, Yaowen Zheng, Peng Liu, Hongsong Zhu, and Limin Sun. 2021. ICS3Fuzzer: A Framework for Discovering Protocol Implementation Bugs in ICS Supervisory Software by Fuzzing. In *Annual Computer Security Applications Conference* (Virtual Event, USA) *(ACSAC)*. Association for Computing Machinery, New York, NY, USA, 849–860. https://doi.org/10.1145/3485832.3488028

[12] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. 2020. AFL++: Combining Incremental Steps of Fuzzing Research. In *Proceedings of USENIX Workshop on Offensive Technologies (WOOT)*. USENIX Association.

[13] Luis Garcia, Saman Zonouz, Dong Wei, and Leandro Pfleger De Aguiar. 2016. Detecting PLC control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*. IEEE, 67–72.

[14] Luis A. Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, and Saman A. Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. *ResearchGate* (Jan 2017). https://doi.org/10.14722/ndss.2017.23313

[15] Hugo Gascon, Christian Wressnegger, Fabian Yamaguchi, Daniel Arp, and Konrad Rieck. 2015. Pulsar: Stateful black-box fuzzing of proprietary network protocols. In *International Conference on Security and Privacy in Communication Systems*. Springer, 330–347.

[16] GitLab. [n. d.]. Peach Fuzzer. https://peachtech.gitlab.io/peach-fuzzer-community/ [Online; accessed 7. Feb. 2023].

[17] CODESYS Group. [n. d.]. CODESYS Device Directory. https://www.codesys.com/download/download-center.html [Online ; Accessed: September 2021].

[18] Shengjian Guo, Meng Wu, and Chao Wang. 2017. Symbolic Execution of Programmable Logic Controller Code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 326–336. https://doi.org/10.1145/3106237.3106245

[19] John T. Hagen and Barry E. Mullins. 2013. TCP veto: A novel network attack and its Application to SCADA protocols. In *2013 IEEE PES Innovative Smart Grid Technologies Conference (ISGT)*. 1–6. https://doi.org/10.1109/ISGT.2013.6497785

[20] Aki Helin. [n. d.]. Python bindings for libradamsa. https://github.com/tsundokul/pyradamsa [Online ; Accessed: January 2022].

[21] J Homan, Sean McBride, and R Caldwell. 2016. IronGate ICS malware: Nothing to see here... Masking malicious activity on SCADA systems. *FireEye threat research blog* (2016).

[22] Helge Janicke, Andrew Nicholson, Stuart Webber, and Antonio Cau. 2015. Runtime-monitoring for industrial control systems. *Electronics* 4, 4 (2015), 995–1017.

[23] Evan Johnson, Maxwell Bland, YiFei Zhu, Joshua Mason, Stephen Checkoway, Stefan Savage, and Kirill Levchenko. 2021. Jetset: Targeted Firmware Rehosting for Embedded Systems. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, 321–338. https://www.usenix.org/conference/usenixsecurity21/presentation/johnson

[24] JSOF Tech. 2020. Ripple 20 – 19 Zero-Day Vulnerabilities Amplified by the Supply Chain.

[25] Sushma Kalle, Nehal Ameen, Hyunguk Yoo, and Irfan Ahmed. 2019. Clik on PLCs! attacking control logic with decompilation and virtual PLC. In *Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS)*.

[26] Anastasis Keliris and Michail Maniatakos. 2019. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. *Network and Distributed System Security Symposium (NDSS)* (2019).

[27] Ralph Langner. 2011. Stuxnet: Dissecting a cyberwarfare weapon. *IEEE Security & Privacy* 9, 3 (2011), 49–51.

[28] Zhengxiong Luo, Feilong Zuo, Yu Jiang, Jian Gao, Xun Jiao, and Jiaguang Sun. 2019. Polar: Function code aware fuzz testing of ics protocol. *ACM Transactions on Embedded Computing Systems (TECS)* 18, 5s (2019), 1–22.

[29] Zhengxiong Luo, Feilong Zuo, Yuheng Shen, Xun Jiao, Wanli Chang, and Yu Jiang. 2020. ICS protocol fuzzing: coverage guided packet crack and generation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[30] MITRE. [n. d.]. CVE List. https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Codesys [Online ; Accessed: January 2023].

[31] Matthias Niedermaier, Florian Fischer, and Alexander von Bodisco. 2017. PropFuzz—An IT-security fuzzing framework for proprietary ICS protocols. In *2017 International Conference on Applied Electronics (AE)*. IEEE, 1–4.

[32] A. Nochvay. 2019. Security research: CODESYS Runtime, a PLC control framework. Part 1. https://ics-cert.kaspersky.com/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-1 [Online; accessed 20. Aug. 2021].

[33] A. Nochvay. 2019. Security research: CODESYS Runtime, a PLC control framework. Part 2. https://ics-cert.kaspersky.com/publications/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-2/#_Toc16177444 [Online; accessed 20. Aug. 2021].

[34] Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. 2020. AFLNet: a greybox fuzzer for network protocols. In *2020 IEEE 13th International Conference on Software Testing, Validation and Verification (ICST)*. IEEE, 460–465.

[35] Ole André V. Ravnås. [n. d.]. Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers. https://frida.re/ [Online; accessed 26. Jul. 2022].

[36] Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. 2018. PLC Code-Level Vulnerabilities. In *2018 International Conference on Computer and Applications (ICCA)*. 348–352. https://doi.org/10.1109/COMAPP.2018.8460287

[37] WIBU Systems. [n. d.]. CodeMeter from Wibu-Systems. https://www.wibu.com/products/codemeter.html [Online; accessed 20. Aug. 2021].

[38] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. 2021. ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications. In *Proceedings of USENIX Security Symposium (USENIX Security)*. USENIX Association. https://www.usenix.org/conference/usenixsecurity21/presentation/tychalas

[39] David Urbina, Jairo Giraldo, Nils Ole Tippenhauer, and Alvaro Cardenas. 2016. Attacking fieldbus communications in ICS: Applications to the SWaT testbed. In *Proceedings of the Singapore Cyber-Security Conference (SG-CRC) 2016*. IOS Press, 75–89.

[40] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James Moyne, and Z. Morley Mao. 2019. Towards Automated Safety Vetting of PLC Code in Real-World Plants. In *2019 IEEE Symposium on Security and Privacy (SP)*. 522–538. https://doi.org/10.1109/SP.2019.00034

[41] Feilong Zuo, Zhengxiong Luo, Junze Yu, Zhe Liu, and Yu Jiang. 2021. PAVFuzz: State-Sensitive Fuzz Testing of Protocols in Autonomous Vehicles. In *2021 58th ACM/IEEE Design Automation Conference (DAC)*. 823–828. https://doi.org/10.1109/DAC18074.2021.9586321