FieldFuzz: Stateful Fuzzing of Proprietary Industrial Controllers using Injected Ghosts

Andrei Bytes\(^*\), Prashant Hari Narayan Raiput\(^\dagger\), Constantine Doumanidis\(^\ddagger\), Michail Maniatakos\(^\dagger\), Jianying Zhou\(^*\) and Nils Ole Tippenhauer\(^\§\)

\(^*\)Singapore University of Technology and Design
\(^\dagger\)NYU Tandon School of Engineering
\(^\ddagger\)New York University Abu Dhabi
\(^\§\)CISPA Helmholtz Center for Information Security

Abstract—With the advent of the fourth industrial revolution, networked industrial Programmable Logic Controllers (PLCs) have been introduced for critical infrastructure control. A number of recent discoveries of exploitable vulnerabilities in third-party libraries in such devices has raised concerns about their supply chain security. Supply chain security verification of software used in this context is challenging due to the proprietary nature of the platforms, and the difficulty of their runtime introspection. In particular, network-based fuzzing is often the only way to test the devices, but without guidance through execution tracing this fuzzing is inefficient.

In this work, we propose a novel approach for dynamic analysis of such platforms, leveraging two main contributions: i) a ‘Ghost' application injected into the fuzzing target to allow on-system tracing and coverage computation, and ii) stateful fuzzing based on automated command discovery and status code extraction. We present FieldFuzz, a framework that realizes this approach for the widespread Codesys runtime for PLCs used by 80 industrial device vendors ranging from over 400 devices.

Our fuzzing campaigns uncovered 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.

I. 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 is performed by proprietary Operational Technology (OT). OT devices suffer from security challenges known from the IT world (such as parsing bugs for traffic), in addition OT devices commonly do not employ OS-level countermeasures that are standard practise in IT nowadays. Exploitation of such vulnerabilities could have catastrophic physical consequences, such as destruction of equipment and loss of service for the population.

At the core of ICS, sensor readings are processed by programmable logic controllers (PLCs) to perform realtime control of physical processes. The realtime control logic is implemented in IEC 61131-3 programming languages (e.g. ladder logic), and executed in proprietary runtime environments. For convenience we also call these IEC applications in the following. Those runtime environments are part of proprietary frameworks that provide Integrated Development Environments (IDEs), compilers, and logic program structure. The most significant Original Equipment Manufacturer (OEM) generic framework is the Codesys runtime, which is at the heart of 80% of industrial device vendors ranging over 400 devices, including manufacturers such as Schneider Automation, Bosch Rexroth, Wago, and Hitachi Europe, representing at least 20% of the active PLCs worldwide.

Vendor can then extend this framework with own code (e.g., third party libraries). This, however, creates a supply chain risk, as the final runtime is a collection of components from various sources. Components can be developed by Codesys, by the vendor themselves, or can be adopted open-source libraries, such as OpenSSL. The urgent/11 and ripple20 vulnerabilities are recent examples for 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 test PLC devices (in ICS context), and in general proprietary embedded systems systematically for bugs? For analysis of proprietary IT software, fuzzing has been very successful in recent years. Unfortunately, fuzzers cannot directly be applied to PLC runtimes, or their IEC 61131-3 programs due to fundamental challenges. In particular, PLC runtimes are complex stateful multithreaded applications that interact through proprietary protocols with the environment. IEC applications have to be executed within the runtime, and require new inputs (via memory-mapped I/O) in each scan cycle. To the best of our knowledge, ICSFuzz is the only reported tool in 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 the requirement of a physical device.

In this work, we propose a novel unified approach for vulnerability discovery throughout the PLC computational stack. Our presented FieldFuzz is the first approach capable of stateful fuzzing all components in a PLC, including the IEC 61131-3 applications. FieldFuzz enables complete stateful control over the execution of the PLC runtime, in particular
the IEC application binary (the control logic), and other components that are reachable through network packets. In addition, we present Ghost, a tool co-located on the target system that enables dynamic black-box instrumentation of the runtime to obtain accurate instruction-level coverage statistics. Ghost is embedded into a IEC 61131-3 control logic program to be executed by the PLC, which allows easy and multi-platform deployment. To speed up our experiments, the target system can be run in a virtual machine, which allows us to perform snapshot-based fuzzing (e.g., fast resets of the targets). We also demonstrate fuzzing of targets that are running on native hardware.

Our experiments resulted in the discovery of multiple vulnerabilities that were responsibly disclosed to corresponding device vendors. Currently, three CVE IDs have been assigned to vulnerabilities discovered by FieldFuzz.

In summary, our main contributions are the following:
1) We propose an approach to fuzz proprietary embedded devices with the help of an injected component (Ghost) that provides feedback to guide fuzzing and allows to derive code coverage. We implement such a component for the Codesys runtime, together with a driver to partially replicate the communication between the IDE and the PLC runtime.
2) Given that framework, we perform runtime system-level fuzzing campaigns. Our methodology is fast, cross-architecture, and non-intrusive, enabling fuzzing for both IEC 61131-3 control applications and all runtime components, independent of the target platform and device vendor.
3) We demonstrate improvement over the state-of-the-art with higher performance, reliable scan cycle control, input delivery, monitoring, breakpoint-based coverage feedback, and no requirement to have a physical device.
4) We discover three previously unknown vulnerabilities and disclose them to the affected vendors with the corresponding Common Vulnerability and Exposure Identifiers (CVE IDs) assigned.

II. BACKGROUND AND PRIOR WORK
A. IEC 61131-3 Program Development

Process engineers develop the process control logic using IEC 61131-3 compliant IDE running on an engineering workstation. The compiled IEC binary is downloaded to the PLC, and the PLC runtime handles the loading and execution of the binary while also ensuring the real-time constraints. The IDE can communicate with the PLC runtime to enable debugging and monitoring of the IEC binary execution.

IEC 61131-3 is a standard that encompasses everything concerning software architecture and supported languages for programming a PLC, including the specifics on syntax, semantics, data types, variable attributes, configuration, and more. IEC 61131-3 defines five types of programming languages for implementing PLC logic: 1) Ladder Diagram (LD), 2) Structured Text (ST), 3) Function Block Diagram (FBD), 4) Sequential Function Chart (SFC), and 5) Instruction List (IL). LD, FBD, and SFC are graphical diagram-based, whereas ST is a high-level textual language, syntactically resembling Pascal. It should be noted that IL was deprecated in 2013.

While certain ICS platforms interpret the intermediate representation of the control program, others employ an IEC compiler to produce a control binary. The IEC binaries differ from known executable formats such as ELF or PE and cannot execute independently. Therefore, they are loaded and executed in the context of the ICS runtime, which controls every aspect of its execution.

B. Codesys Environment

Codesys is a multi-platform software that includes the development system and the runtime for target ICS devices. The Codesys Development System allows the development of control programs executed by the target device that runs Codesys Runtime. The runtime is a collection of components with a modular architecture implemented as statically compiled and dynamically linked necessary libraries. Components are integrated from various sources, such as Codesys itself, the device vendor, or open-source libraries. A Component Manager is responsible for launching and initializing all other components.

At the same time, the increase in the occurrences of traditional vulnerabilities in IEC applications follows the evolution of the support of advanced external libraries. For instance, Codesys allows integrating external C modules with the IEC application, bringing with it associated vulnerabilities such as buffer overflows: CVE-2021-30188, CVE-2021-33485. This is also evident in the increase in the number of CVEs over the year as shown in Table I. As an 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.

C. Prior Work

Protocol fuzzing. There is a considerable amount of work on fuzzing network protocols. For instance, AFLNET, a greybox fuzzer fed with a mutated corpus of recorded network messages utilizes state-feedback for guiding the fuzzer and KIF for fuzzing session initiation protocol. In addition, Pulsar is a stateful black-box fuzzing testing technique for proprietary network protocols that utilizes automatic protocol reverse engineering and simulation. There are ICS network-protocol-specific solutions such as Peach*, a coverage-based improvement over the standard Peach fuzzer. PropFuzz, unlike traditional fuzzing approaches, monitors the behavior of the controller along with the network connection to detect unexpectedly long jitters in the control process. Polar utilizes static analysis and dynamic taint analysis to

---

1We have asked the numbering authority and the vendor to anonymize the credits during the review phase.
identify vulnerable operations with semantic aware mutation to improve the fuzzing procedure. Finally, ICSFuzz is a framework for discovering implementation bugs on 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 ICSFuzz fuzzes the supervisor software by using the network communication protocol and is not concerned with fuzzing PLC devices. While FieldFuzz can also do protocol fuzzing (as ICS communication is also a component), it focuses on vulnerability discovery of any component integrated into the PLC.

**Control application security.** Research on IEC application binaries focuses primarily on their safety verification. For instance, Canet et al. 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 and monitoring violation against given specification for runtime components. In VetPLC, Zhang et al. utilize static analysis for creating timed event graphs combined with invariant data traces to detect hidden safety violations. Guo et al. propose an automated PLC control application testing software that translates PLC source into C and performs symbolic execution to generate test cases. On the other hand, AttkFinder uses information flow-guided symbolic execution on an intermediate representation of PLC code to detect attack vectors automatically. Kelliris et al. reverse engineered the Codesys v2.3 file format for IEC application binaries to propose an automated on-the-fly attack formulation. Similarly, CLIK 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, a fuzzing framework for PLC applications and system functions of the runtime. It supplies inputs to the IEC binary by overwriting the value at the memory-mapped GPIO, relayed to the runtime via the KBUS subsystem. ICSFuzz patches the NOP instructions in the compiled control application code to obtain coverage feedback. It detects multiple crashes for a synthetic IEC application binary dataset and a limited subset of runtime functions. However, it has specific limitations:

- ICSFuzz utilizes the KBUS subsystem for input delivery to the control application, necessitating the use of a physical PLC, making it non-scalable.
- Due to the loss of synchronization with the scan cycle of the control application, ICSFuzz periodically drops fuzzing inputs and is, consequently, slow.
- ICSFuzz lacks automation for observing the state of the control program and involves manual crash monitoring.
- ICSFuzz 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 execution context of the runtime.

FieldFuzz addresses all the above limitations as its methodology is generic and IEC application fuzzing is just an instance of component fuzzing, as was the case with protocol fuzzing discussed earlier.

**III. FUZZING OF PROPRIETARY INDUSTRIAL CONTROLLERS**

In this work, we address the following overall problem: Can fuzzing be used to efficiently identify bugs in proprietary PLC computational stacks? Achieving this goal will require us to address several research questions, which we will list next. Then, we summarize research and engineering challenges to achieving the overall goal. Finally, we present a high-level overview of the proposed FieldFuzz solution.

**A. Research Questions**

- **RQ1:** How to systematically test for supply chain risks in the code-base of proprietary industrial IoT devices? In particular, how to test proprietary third-party code in PLC runtimes such as Codesys?
- **RQ2:** How to guide this testing of proprietary bare-metal runtimes without local access to introspect the device?
- **RQ3:** Is it possible to generalize the testing approach to be cross platform?

**B. Research and Engineering Challenges**

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

- PLC runtimes are typically closed-source proprietary software, thus fuzzing has to be performed in a black-box manner or using extensive reverse engineering.
- PLC firmware rehosting is very challenging with some recent efforts only achieving partial rehosting. At the same time, symbolic execution is also challenging, given the complexity of the firmware blob. Therefore, comprehensive and accurate PLC fuzzing would require actual hardware.
- 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 can be as challenging as controlling a full-blown operating system.

**C. System and Threat Model**

We assume the fuzzing of a PLC is conducted by the owner to identify security vulnerabilities present in the PLC code, in particular in third-party libraries running on the system. The PLC’s firmware is available as a binary without debugging information. The owner can install additional IEC control logic on the PLC (which we will leverage for injection of our tool Ghost).

The goal is to identify vulnerabilities that could be exploited by an attacker that is able to monitor, intercept and modify network communication to the PLC, essentially performing a man-in-the-middle (MITM) attack (e.g., as in Stuxnet and IRONGATE). Other similar research work such as TCP veto and CLIK employ the same assumption.

The adversary can deliver malicious input to the target PLC through various approaches, such as:
D. Proposed Approach

We propose to reach the goal through the introduction of two particular components: i) a Ghost application running in the target, and ii) a runtime driver to encapsulate the stateful protocol interactions with the target (see Figure 1). This addresses the research questions and challenges as follows:

- To address RQ1, FieldFuzz uses a dynamic fuzzer component to systematically generate complex CoDesys protocol traffic with the help of our FieldFuzz Runtime Driver (FRD). The fuzzer can mutate low-level protocols, which allows full testing of network-facing third-party binaries on remote PLCs. The FRD enables us to bootstrap complex sequences of protocol interactions, and parse resulting traffic from the fuzzing target.
- To address RQ2, we enable feedback from (potentially remote) embedded devices such as PLCs by proposing a dedicated Ghost component to be injected into the target process (or operating system). The Ghost monitors the target binary, attaches to newly forked processes that handle incoming network traffic, and traces their execution. The results can then be provided to the fuzzer to guide further fuzzing.
- Together, the implementation of FRD and Ghost enable us to assess performance and compare with prior work.
- To address RQ3, we will demonstrate that runtime bugs identified by our framework on a specific architecture (i.e. x86) also translate to runtimes executed on other products, and architectures (such as ARM32).

IV. GHOST AND FIELDFUZZ RUNTIME DRIVER (FRD)

We now present details on Ghost and FRD (our runtime driver), which enables stateful system-level fuzzing.

A. Ghost

Understanding code coverage is essential to assessing the effectiveness of our fuzzing campaigns. This is particularly challenging since the Codesys Runtime is proprietary software. Lacking access to the source code, we are forced to work on the binary level. This is also not straightforward, since disassembling the runtime in IDA Pro reveals that Codesys employs a binary packing mechanism as an anti-tampering measure which hinders our efforts to locate the components code. Preliminary examination did not identify the use of any widely used packer software. To overcome this, we launch the runtime on the target machine and use step-by-step execution to pinpoint the exact moment when unpacking has finished. At that point we dump the memory space of the whole runtime and import it into IDA Pro for disassembly and further analysis.

Studying the dumped runtime memory in IDA Pro shows that the runtime binary was also stripped of its debug symbols during compilation. This makes it difficult to locate in the memory dump the segments of the components we are interested in fuzzing. During the runtime initialization phase, component setup functions make calls to ServiceRegisterServiceHandler, a function used to register runtime components. These calls include memory pointers to the handler functions of the runtime components. In order to capture these memory pointers we leverage Frida\[37\], a dynamic instrumentation toolkit widely used for reverse engineering. We use Frida Interceptor to hook into ServiceRegisterServiceHandler calls and retrieve the memory pointers. Using these pointers as starting points we analyze the component handler functions and locate all the memory segments relevant to each component and its commands.

To get accurate assembly instruction-level coverage we need to instrument the runtime and monitor the threads that it spawns for command calls. Anti-tampering mechanisms hamper attempts at static instrumentation, so we again use Interceptor to monitor for calls to component handler functions. When a call is intercepted we launch Frida Stalker, a code tracing engine that dynamically recompiles assembly code and traces the executing thread. By getting feedback from the Stalker dynamic 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 2). Ghost is initially configured with access to the runtime memory dump and the relevant memory segments of the components to be fuzzed. Upon execution, it spawns the Codesys runtime and immediately independently attaches a set of Frida scripts in order to capture all
TABLE II: Commands associated with components of our interest at the service layer. In bold the commands replicated by our FRD that are crucial for the fuzzing routines. Markings denote the fuzzing stage they are required: ‘I’: Initialization, ‘E’: Execution control, ‘M’: Monitoring.

| Cmp   | Command                          | ID  | Cmp   | Command                          | ID  | Cmp   | Command                          | ID  | Cmp   | Command                          |
|-------|----------------------------------|-----|-------|----------------------------------|-----|-------|----------------------------------|-----|-------|----------------------------------|
| CmpBlkDrv [0x02] | GetTargetIdent [0x01] | 0x01 | CmpApp [0x02] | DeleteApp | 0x04 | CmpApp [0x02] | ReadAppList [0x02] | 0x18 | CmpApp [0x02] | ReadProjectInfo | 0x31 |
|       | Login                             | 0x02 |       | Download                         | 0x05 |       | SetNextStatement                | 0x19 |       | DefineFlow                       | 0x32 |
|       | Logout                            | 0x03 |       | OnlineChange                     | 0x06 |       | ReleaseForceList                | 0x20 |       | ReadFlowValues                   | 0x33 |
|       | SessionCreate                     | 0x0A |       | DeviceDownload                  | 0x07 |       | UploadForceList                 | 0x21 |       | DownloadEncrypted                | 0x34 |
|       | EchoService                       | 0x05 |       | CreateDevApp                    | 0x08 |       | SingleCycle                    | 0x22 |       | ReadAppContent                   | 0x35 |
|       | SetOperatingMode                  | 0x06 |       | Start                            | 0x10 |       | CreateBootProject               | 0x23 |       | SaveRetains                      | 0x36 |
|       | GetOperatingMode                  | 0x07 |       | Stop                             | 0x11 |       | RemoteApp                       | 0x24 |       | RestoreRetains                   | 0x37 |
|       | InteractiveLogin                  | 0x08 |       | Reset                            | 0x12 |       | ReadAppStateList                | 0x25 |       | GetAreaAddress                  | 0x38 |
|       | RenameNode                        | 0x09 |       | SetStatus                        | 0x13 |       | LoginExecApp                    | 0x26 |       | LeaveExecAppActive                | 0x39 |
|       | Logon                              | 0x01 | CmpApp [0x02] | DeleteBP | 0x15 | CmpMonitor2            | Read | 0x01 |
|       | CreateApp                         | 0x03 |       | ReadCallStack                   | 0x16 |       | ReadAppInfo                     | 0x29 |       | Write                            | 0x02 |

Fig. 3: Important fields at different layers of the Codesys v3 Protocol used by FieldFuzz.

B. Overview of the FRD

The FieldFuzz Runtime Driver (FRD) is implemented as a Python library and is the cornerstone of FieldFuzz. The key capabilities of the FRD are the following:

1) Setting up the runtime to be in a suitable system state;
2) Delivering the fuzzing inputs to runtime components of choice;
3) Receiving feedback for crash monitoring and analysis;
4) Enabling exploit development through provided templates and extended debugging capabilities.

FieldFuzz leverages the proprietary communication protocol utilized by Codesys to communicate with the runtime and enables 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 industrial network with diverse segments of Ethernet, CAN, Serial, Sercos, and other mediums. Without loss of generality, FieldFuzz connects to field devices with TCP over Ethernet.

To gain sufficient knowledge for the development of the FRD, we developed a complete Wireshark dissector for the Codesys v3 communication protocol, written in Lua. Assisted by the parsing and filtering capabilities of the dissector on the traffic captures, in combination with manual reverse engineering of the Codesys runtime and the information published

in [34], we extracted all the information needed to develop the FRD. This also allowed us to collect a corpus of valid interactions with components and communication patterns required for stateful fuzzing.

Table [II] presents a sub-set of commands associated with chosen components of the runtime. These are crucial for our fuzzing routines. The commands replicated by the FRD 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 FRD, we show later in Section [VII] that FRD 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.

C. System Initialization

The runtime utilizes a proprietary network stack that we will refer to as the Codesys v3 protocol to facilitate network communication between nodes. The Codesys v3 network stack consists of four layers:

- **Block driver layer.** The component (CmpBlkDrvTcp) at this layer is responsible for communication over the physical interfaces. It processes a magic number for verification, and the total number of bytes in the packet:

\[ 3 \text{Detailed information about the network stack is available in [34].} \]
length. CmpBlkDrvTcp transfers the rest of the packet to CmpRouter component of the Datagram layer.

Datagram layer. The CmpRouter component detects Codesys nodes in the network and routes the packets. Its RouterHandleData function parses the following fields: 1. One byte magic number (0xc5) inserted by CmpRouter, 2. hop_info bit structure, 3. packet settings, 4. service_id identifying the destination service (service ID of 1 and 2 for request and response, respectively), 5. lengths field, 6. address of the sender and receiver, and finally the optional padding. In Fig. [3] the value of the service_id is 0x40, so the packet is forwarded to the CmpChannelManager component in the channel layer.

Channel layer. This layer employs CmpChannelMGR which ensures synchronized communication, integrity verification and delivery acknowledgement. Fig. 3 shows its fields: 1. specifies certain commands (command_id) for the communication channel manager. A dedicated Layer 3 channel must be open, kept alive and closed for successful communication. The command_id of 0x1 (BLK) indicates the transfer of data to the next layer, 0x2 is ACK, and 0x3 for KEEPALIVE. Next, 2. the flag field, 3. channel_id for identifying the open channel, 4. blk_id is the ID of the current BLK message, 5. ack_id is the ID of the previous ACK message, 6. remaining_data_size field is the size of the expected data remaining in the packet, and 7. is the checksum field with the CRC32 checksum for the remaining data.

Service layer. This layer is responsible for querying the requested service and transmitting the operating settings. The message consists of: 1. protocol_id, 2. header_size, 3. service_group refers to the ID of the queried service. Runtime has a unique ID for each available service and identifies the corresponding component for processing the service request. 4. service_id refers to the commands available for a specific service, 5. session_id identifying the session, 6. content_size, 7. additional_data, and protocol_data.

The FRD initiates communication with the runtime over the network using the described protocol. Typically, the runtime listens on multiple ports for connection requests (TCP 1217, 11740, and UDP 1740 to 1743). The FRD opens a new channel using the Layer 4 commands 0x40 and 0xc3 and stores the channel ID. Depending on the requested service layer command, there is a need for establishing sessions in at least two layers for properly sending the commands to the components: The Device and the Application login stage. The FRD 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 retrieves a list of loaded applications from CmpApp. Finally, if the application is loaded on the PLC, the FRD opens a second session via the CmpApp Login command to obtain the application Session ID and its handle. Finally, it implements a keepalive mechanism to keep the channel active despite the timeout imposed by the runtime.

D. Input Delivery

In literature, often bare-metal IO modules are utilized as the primary communication method for PLCs [41], [15]. However, such an approach is not scalable and is often specific to the model or series of the PLC. Another approach would be to utilize the Modbus protocol from the Fieldbus family of network communication protocols. Such an approach would require the project to include the Modbus client object to receive read and write commands, requiring explicit declarations of such exports in the control project, not enabled by default. Commonly, the HMI displays and modifies inputs to a program using the OPC Unified Architecture (OPC UA) protocol. However, for Codesys, its support is not compiled into all distributions and requires a license verified by CmpOPCUAPICommonVarAccess component; otherwise, it shuts down the OPC UA server in 2 hours. For symbolic access, a global variable list has to be mapped in the project implicitly. In this case, the operator workstation would be able to access chosen variables by name, assisted with CmpIecVarAccess [11] component.

The approaches mentioned above require explicit modification of the configuration in the control project to support specific Fieldbus protocols and symbolic variables. To address this problem, the FRD utilizes a universal approach and does not require modification of the project while enabling read and write to any variable regardless of their 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. Each tag starts with its ID: 1. tag_id, which often corresponds to the type of payload or status code, the tag_size, and the tag_data, and some additional_data. Fig. 5 shows the tag fields for writing to a variable. 2. data_size is the size of the data to be written, 3. write_value is the value to be written and 4. write_offset is the offset of the variable from the start of the data section.

The FRD utilizes the following to deliver inputs to components and the IEC application: First, the FRD uses the (Service Group ID, Command ID) tuple to identify the appropriate component for routing the input. Then, using the recursive tag encoder algorithm [35], a binary tag structure is built 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, FRD calculates a CRC32 checksum for this data, constructs a packet header, and encapsulates it with all the remaining layers (see Subsection IV-C). It then sends this data to the runtime over the active channel. The Block Driver of the runtime receives the packet, processes it, and passes it to CmpSrv. This component 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. This process provides inputs to target components of choice, enabling stateful component fuzzing.

However, delivering inputs to an IEC application involves several additional routines. FRD achieves this by performing the write operation over the network to the IEC application memory segment. Typically, runtime separates data and IEC code (as compiled instructions) into separate memory segments, referred to as Area0 and Area3, respectively. The FRD reads and writes variables using relative offset addresses in the Area0 memory segment. For this, the FRD constructs
short bytecode programs expected by the CmpMonitor2 component as inputs, using a set of 58 opcodes specific to the runtime. The opcodes can read inputs from the interpreter stack and receive up to 4 inline arguments, depending on the syntax. To implement this routine of the FRD, we extract the opcode table from the reverse-engineered runtime binary and map their names and format with the decompiled OnlineManager.plugin.dll 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 0x41, such as wrong pointer (0x05), buffer overrun (0x08), and more. The FRD detects a successful read operation with the presence of tag 0x40 in the reply and 0x41 for a failed attempt. In contrast, the write operation requires a more complex bytecode program and returns no data in the reply for a successful attempt. However, FRD is still able to verify a write attempt in a similar manner by the presence of the tag 0x41. The complete process is shown in Figure 4.

Table III: The status codes used by FieldFuzz to uncover system states.

| Status   | ID  | Status    | ID  | Status    | ID  | Status    | ID   |
|----------|-----|-----------|-----|-----------|-----|-----------|------|
| Ok       | 0x00| Overflow  | 0x0E| NoAccessRights | 0x19| InvalidSequence | 0x3A |
| Failed   | 0x01| BufferSize| 0x08| OutOfLimits | 0x13| TooManyRetries | 0x3D |
| Parameter| 0x02A| NoObject | 0x10| EntrenRemaining | 0x1B| AlignmentError | 0x3F |
| NoInitialize | 0x03| NoMemory | 0x11| InvalidSessionId | 0x1C| PasswordChangeRequired | 0x3F |
| Timeout  | 0x04| Duplicate | 0x12| Exception | 0x1D| Reauthenticate | 0x40 |
| NoBuffer | 0x05| MemoryOverwrite | 0x13| SignatureMismatch | 0x1E| Relogin | 0x41 |
| Pending  | 0x06| InvalidHandle | 0x14| VersionMismatch | 0x1F| NotReadyYet | 0x45 |
| NumPending | 0x07| EndOfObject | 0x15| TypeMismatch | 0x20| ActivateUserMgt | 0x46 |
| NoImpl    | 0x08| NoChange | 0x16| IdMismatch | 0x21| NetUserMgmt | 0x47 |
| NoInvalidInterface | 0x09| OperationDenied | 0x17| OperationDenied | 0x22| NetFailed | 0x100 |
| InvalidId | 0x0A| NotSupported | 0x18| SecurityChecksFailed | 0x39| NetNotConnected | 0x101 |
| Status    | ID  | Status    | ID  | Status    | ID   |

E. Stateful Fuzzing

To facilitate stateful fuzzing, FRD 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. 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 other components of the runtime system. Furthermore, these components expect commands in a specific sequence, which cannot be easily uncovered by blackbox fuzzing.

Command discovery. Typically, Service Group IDs (2 bytes) for the vendor-added components specific to a runtime variant are within a dedicated range beginning from 0x100. This range can be enumerated, along with the Command IDs (2 bytes). Using the FRD, 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.

Status code extraction. The architecture patterns enforced by the runtime provide hints for identifying functions related to the components in the reverse-engineered runtime binary. Furthermore, each component follows a specific programming pattern to be integrated with the runtime, as mentioned below:

- It contains an entry function with a standard C struct passed as the argument. One example of such entry function for CmpMonitor2 component is shown in Listing 1 which we renamed as CmpMonitor2__Entry.
- Declares its internal ID (in this case 50, or 0x32 in hex) and passes a string literal of its name to the CM.
- Calls standardized import and export functions that enable inter-component communication (Line 4-5).
- Handles standardized create, delete event hooks and a version identifier function (Line 6, 8, and 9).
- Subscribes for custom events sent by other components using a pointer to an event handler function (Line 7).

```c
int STATUS CmpMonitor2__Entry(INIT_STRUCT *init)
{
  init->CmpId = 50;
  init->ExportFunctions = CmpMonitor2_Export;
  init->ImportFunctions = CmpMonitor2_Import;
  init->GetVersion = CmpMonitor2_GetVersion;
  init->HookFunction = CmpMonitor2_Hook;
  init->CreateInstance = CmpMonitor2_Create;
  init->DeleteInstance = CmpMonitor2_Delete;
  [...] 
  return 0;
}
```

Listing 1: Pseudocode for CmpMonitor2__Entry.

We parse the device description (devdesc) file that the IDE uses to communicate with the runtime, a file specific to each runtime variant to extract the set of enabled components and their IDs. These internal identifiers differ from Service Group IDs that are mapped to the components. Then, utilizing the runtime binary and the knowledge of architecture patterns, we deploy IDA Pro scripting to automate the process of extracting status codes. It begins by locating the component entry function that fills in a standardized initialization structure for the component to be fuzzed. This function passes the initialization structure to the Component Manager for communication with the component. Next, we identify the event handler function among other standard declarations in the initialization structure. This function always consists of a lengthy conditional statement that checks for generic runtime-wide event codes. Then we iterate through the event conditions to locate the call to ServiceRegisterServiceHandler function. The event code that registers the service handler can differ, but according to our observations, it is tied to events 0x03 or 0x06. The second argument passed to ServiceRegisterServiceHandler is the pointer to the service handler function of the component. The script tracks the path to the function call, which implements the given Command ID. Finally, we recursively extract all return conditions from the nested calls as these contain the status codes that are later returned by the service layer, depending on the executed path, and provide it to the FRD for implementing code coverage feedback.

Path discovery. As a code path coverage mechanism for component functions, the FRD maintains the status code sequence during the fuzzing session. Based on the status code sequence changes resulting from mutated inputs, it is possible to understand and differentiate the execution path inside the component function. The status codes differ as the fuzzing input traverses through 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, every time 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.

Execution feedback. For feedback, FRD watches for the status codes returned by the different layers of the Codesys v3 network protocol stack in hexadecimal form, as shown in Figure 4. For example, a reply with only Layer 4 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, or incorrect payload format. Table III presents a list of common status codes returned by the runtime components. As was already done for the monitoring opcode names, we find the status codes in a decompiled OnlineManager.plugin.dll shared library of the Codesys IDE. Some of the important status codes are L7TagMissing (768/0x300), L7UnknownCmdGrp (769/0x301), and L7UnknownCmd (770/0x302). In the case of a non-existing Service Group ID and Command ID, the runtime returns the status code for L7UnknownCmdGrp and L7UnknownCmd, respectively. To retrieve the status of the execution of the IEC application binary, FieldFuzz utilizes the FRD to survey the CmpApp and CmpPlcShell components. Upon exception, a core dump and crash log from CmpLog are retrieved from the controller remotely for further investigation. To detect crashes of the runtime, FRD constantly monitors the latency in the channel and the consistency of the BLK counter.

FRD exploit development capabilities. The FRD is also an essential tool for crash analysis, aiding with exploit development. These capabilities include individual interaction with each layer, a Wireshark dissector, a binary tag encoder, ability to obtain remote memory dumps, and the ability to retrieve runtime logs.

V. FieldFuzz Fuzzing Campaigns

To demonstrate we addressed RQ1 and RQ2, we now apply our framework to fuzz the Codesys runtime and IEC application binaries.

A. Fuzzing Setup

Fig. 5 shows the basic experimental setup for fuzzing the Codesys runtime and IEC application binaries. We utilize two virtual machines with CODESYS Control for Linux variant of the runtime, which runs on the Intel Xeon-based hypervisor server. The runtime includes a standard init.d wrapper that facilitates the automatic restart of the runtime after a crash caused by FieldFuzz. 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 the use of an additional physical device here is optional. FieldFuzz can utilize virtual machines, any number of which can be deployed 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. The FRD could achieve this by remotely increasing the maximum number of active channels that CmpChannelServer supports. This can be performed over the network with the SettgSetIntValue (0x02) command offered by the CmpSettings (0x06) component. However, this adds to the complexity of the crash monitoring mechanism to distinguish the crashes of the runtime caused by parallel threads. The flexible nature of the used protocol allows it to be routed in a single industrial network with mixed segments of Ethernet, CAN, Serial, Sercos, and other mediums. The devices can route the packets over the industrial network based on the hierarchy and the destination address specified by the FRD. In this way, once FieldFuzz connects to a physical device (PLC, industrial gateway, or touch panel) through the FRD, it enters the entire industrial network. While FRD uses TCP over Ethernet, depending on the destination in the packet, 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. By putting the devices in different states, FieldFuzz could potentially fuzz the interaction of the control logic between devices in the particular stage of a process segment or the entire plant.

Identifying fuzzing targets. As discussed earlier, the runtime is a collection of components (including the component responsible for executing IEC binaries), so fuzzing the runtime implies interaction with the components responsible for its functionality. However, as was mentioned in Section III 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), for the components reachable from the application layer (Layer 7). 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, assisted with the dissector. Table IV presents a subset of components that are present 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 to extract the Service Layer payloads, assisted by the dissector. More specifically, we aim to decode the nested binary tag structure, shown in Figure 4. We then save the tag IDs, structure, valid payloads for each (Service Group, Command) tuple. Next, we determine the packet fields derived from the session identifier to identify the variant fields as these need to be tracked 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 libbradamsa mutators ported from AFL++ to mutate the byte payload inside the tags without mutating the tag ID and preserving the remaining structure of the corpus. FieldFuzz then utilizes FRD to deliver the input to the appropriate component.

Code Coverage. Coverage is calculated on the component, handler and command level using Ghost (see Section IV-A). On the component level, coverage includes all the memory segments associated with a component, complete with command code and the handler function. Handler coverage indicates the percentage of instructions executed inside the component. 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. We initially embed the relevant binaries into the IEC project so that they will be downloaded onto the local file system when the project is loaded onto the PLC. Then, using the SysFileCopy API (available in the SysFile library) to relocate the binary files in the file system. Then, 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 to allow for arbitrary shell command execution. Having done this,
TABLE V: 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%           |

we restart the runtime through the IDE or reboot the target PLC so that the runtime to picks up the configuration changes. Upon reboot we use SysProcessExecuteCommand2 (part of the SysProcess library) to perform the necessary setup and run Ghost with root privileges.

As a proof of concept, we run fuzzing campaigns on functions of three common components: CmpTraceMgr, CmpPlcShell, and CmpDevice. FieldFuzz was able to uncover a variety of crashes, which we then analyzed, focusing on uncovering vulnerabilities. For brevity, we provide extensive discussion for one CVE, and shorter discussions about the other two.

B. Fuzzing CmpTraceMgr Component (CVE-2021-34604)

The CmpTraceMgr component consists of eight critical operations 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 (POU) object added to the IEC project for recording and visualizing variable trends in the physical process. Here, the recordAdd operation causes SEGFAULT for two reasons:

- As the command is sent out-of-order, the component enters an unexpected state where it operates on a pointer to a structure of a packet object that is never correctly initialized.
- The offset calculation into this non-existent structure based on the value supplied by FieldFuzz. The attacker can influence the calculation by controlling this offset, leading the runtime to perform mov operation on an invalid memory address.

Setup. We employ the corpus extracted from the communication between the Codesys IDE and the runtime and pass it to FieldFuzz. The corpus comprises Layer 7 payloads with removed headers consisting of Service Group ID and Command ID. FieldFuzz reuses the seeds by reconstructing the header while ensuring the correct length, checksum, and the desired Service Group and Command pair.

Coverage. Table V presents the coverage reported by Ghost during fuzzing campaigns on the CmpTraceMgr component. Our fuzzing strategy managed to obtain 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 stateful sequences of command events that were not generated during input mutation. Therefore, these commands are not appropriately fuzzed by the default AFL setup and stateful input mutation strategies should be employed, such as [5, 8, 44]. Input mutation strategies are orthogonal to our work and will be explored in future research.

Crash analysis. FieldFuzz reported a crash for Service Group 0x0F, command 0x0D, and upon verification, the Command ID is in the range of valid commands for the particular component 0x01 to 0x13, CmpTraceMgr. We look deeper into the original pcap file used for creating the seed input for understanding the structure of the input causing the crash. Using our dissector with a filter expression, we determine that the recordAdd (0x0D) command consisting of 148 bytes payload causes the crash. This payload incorporates three levels of nested binary tags.

Reproducing the crash. To investigate the crash, we use FRD to generate a standalone exploit from a template that establishes a connection with a SoftPLC node supporting full-featured debugging capabilities and sends the Layer 7 payload to the remote device. To observe critical runtime errors, we enable core dumps and disable the error handling behavior of SysExcept. Generally, the SysExcept component of the runtime intercepts POSIX signals from the runtime process 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 the core dumps, we adjust the ulimit and launch the runtime binary (codesyscontrol.bin) standalone outside its init.d service wrapper, provided with a -d flag for detailed logging.

Call stack. At least 12 functions handle the packet of the network stack 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 the CmpTraceMgr, which acts as a handler for all Layer 7 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 finally decode the 17 binary tags. Among these, a tag 0x40 is processed, which was an injection point selected by FieldFuzz. It calculates a memory address offset based on the extracted value. Consequently, in the command handler function for the recordAdd command, a SIGSEGV occurs, which is caused by a mov instruction attempting to access the nonexistent memory address. This is the offset from a structure that stores a tracing packet, derived from the value supplied by FieldFuzz.

Status codes. 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. A reply containing the tag 0xFF7F with a status code 0x02 was caused by the payloads in tag 0x40 that are outside the expected range, such as 0x00 and 0xFFFFFFF. This prevents the crash as the read operation is not reached. 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 entry checks of the recordAdd function. The read operation is reached, resulting in a handled exception due to returned empty packet data.
This vulnerability has been reported to the vendor, with the CVE-2021-34604 assigned.

C. Fuzzing CmpDevice Component (CVE-2022-22508)

CmpDevice is an essential component responsible for 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. 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 reboot because the payload from the network packet ends up in a persistent configuration file of the runtime 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 the Service Group 0x01 (CmpDevice), command 0x09 (SetNodeName) and the tag 0x58 with a long bytestring consisting of non-printable characters as Layer 7 payload. This crafted bytestring is not sanitized properly by CmpDevice before passing it to the local SysTarget component (which might be vendor-specific), we tested with official Codesys distributions) and then stored permanently in the NodeNameUnicode property field. The device is not accessible even after a reboot. This is appended to 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 Layer 4 and calls CmpDevice again to perform GetTargetIdent, and CreateSession commands. The system log messages suggest that CmpNameServiceServer processes the bytestring, which is a Layer 4 component that exports its functions to CmpRouter and implements a Codesys-specific naming system protocol [34]. Consequently, the device fails to respond to further scan requests and several exceptions by the dynamic libraries of the Codesys IDE. A manual remove of the SysTarget section from the runtime configuration file restores the operational state of the device, after a reboot.

D. 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 diagnosis of the device by sending string commands of a particular format. An adversary can trigger 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-2022-22508 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 occurs once the operation exceeds valid memory boundaries, crashing all the runtime threads.

| Device       | Arch | Size (MB) | Packed |
|--------------|------|-----------|--------|
| WAGO PFC200  | arm32| 4.6       | X      |
| BeagleBone Black | arm32| 5.8       | ✓      |
| Linux SoftPLC | x64  | 9.7       | ✓      |
| Raspberry Pi  | arm32| 5.5       | ✓      |
| SIMATIC IT/E2000 | x32  | 6.5       | △      |
| emPLC-AO/MX6  | arm32| 6         | ✓      |
| Windows RTE   | x32  | 103.9     | ✔      |

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

VI. CROSS-ARCHITECTURE GENERALIZATION

To address RQ3, we now discuss 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. Thus, on platforms driven by VxWorks real-time operating system (RTOS), the entire Codesys runtime is shipped as a kernel module. The embedded bare-metal variant of the runtime has a much smaller set of components but implements more complex system components to interact with the hardware. In more modern ICS devices powered by RTLinux (such as WAGO Touch Panel 600 series or WAGO PFC200 PLC), it runs as root in the userspace and reuses resources provided by the OS such as network sockets, timers, and file descriptors. As shown in Table VI, the size of the runtime binaries varies across various architecture from 4.6 MB to 103.9 MB. One reason is that the shared libraries can be linked dynamically or statically depending on the variant. The number of components also differs. Some of the binaries are packed and involve license management and anti-tampering mechanisms. Thus, the primary distribution of our choice in this work (CODESYS Control for Linux x64) employs a packing mechanism which we have reversed dynamically by dumping the memory segments of the live process. The runtime variant for Windows devices (CODESYS Control RTE x32), on the other hand, includes custom renamed and encrypted sections. From the section names and the protection function, we have noticed that these are managed by CodeMeter protection software from Wibu-Systems [39], which has also been utilized by Siemens and Rockwell [9].

The most hardware-dependent components are SysMem, SysSocket, and SysCom. At some point in the execution path, other components rely on the exported functions provided by these lower-level Sys components, which can affect 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 replay 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 both as a VM hypervisor and the Gateway to Wago PFC200, Raspberry Pi4, 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 CmpTraceMgr component (CVE-2021-34604), which is available on all of the tested devices. As was shown in Figure 6, the original payloads replicated here were detected by FieldFuzz in a virtualized environment (with ASLR disabled). 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 (eight bytes), eventually leading to the crash. It should be noted that the payload behaves identically on Intel and ARM, causing the crash on both the VM and physical devices; it only differs between the 32 vs. 64 bit architecture of the target device.

VII. FUZZING IEC APPLICATION BINARIES

Fuzzing of IEC application binaries is a special case of component fuzzing since dedicated components control the execution of these binaries. 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 VII shows the IDs of the corresponding commands replicated for CmpApp.

To set up the experiment, FieldFuzz utilizes the FRD to login into the device and start the control application. Next, FieldFuzz takes over the execution control while providing fuzzing inputs for every scan cycle. Since FieldFuzz has control over the scan cycle, it does not drop any inputs due to a lack of synchronization with the scan cycle. Finally, based on the status feedback received from the runtime, FieldFuzz logs the crash input.

To test the performance of FieldFuzz, we use the same dataset of synthetic applications as was used for the evaluation of ICSFuzz [41]. It comprises of the control applications written in Structured Text that contain introduced vulnerabilities in their called 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 itself, this library is written in Structured Text. By looking deeper into its implementation in the runtime, we observe that the backend for this library is provided by the SysMem component of the runtime and is written in C. The call of this wrapper, initiated by the IEC program, ends up in C code which triggers the native memmove function. For this reason, the crash in a vulnerable IEC application not only causes the failure of its thread but affects the entire runtime process (executed 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 application binaries correlate with the complexity of the code. For instance, bf_mmove_1 is the simplest initialization of the code, while bf_mmove_12 consists of multiple loops and conditional branching statements.

Table VII shows the results of fuzzing the IEC application binary and its comparison with ICSFuzz. As the table demonstrates, FieldFuzz on an average 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 also be emphasized that the measurements in Table VII are taken 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. FieldFuzz can parallelize fuzzing sessions by just 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 (Intel) runtime variants, respectively, in the same 1 hour fuzzing period. However, FieldFuzz detects fewer crashes for a select few samples across both variants. As mentioned previously, low-level System components are implemented differently across various devices, resulting in different bugs and vulnerabilities. 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.

VIII. DISCUSSION AND LIMITATIONS

Security mitigations by the runtime. The latest runtime version enables the User Management feature by default, thwarting unauthorized login into the PLC. However, out-of-the-box credentials are default and communication is not encrypted, unless manually changed. 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 controller that shipped with firmware 03.00.39(12) used in our setup contains the runtime that is
TABLE VII: Performance comparison with ICSFuzz. Speed for FieldFuzz is when using 1 VM; Speed for ICSFuzz is when using 1 PLC.

| Control Applications | Execution Speed (inputs/seconds) | First Crash (seconds) | First Crash (inputs) | Crashes |
|----------------------|----------------------------------|-----------------------|----------------------|---------|
| FieldFuzz            | A8 x64 A8 x64 A8 A8           | FieldFuzz             | A8 x64 A8 x64 A8 A8 | FieldFuzz |
| bf_mcpr_1            | 294.6 593 70.88               | 0.014 0.25 234       | 6 148 15270          | 8289 7876 32 |
| bf_mcpr_2            | 288.1 612.7 74.62            | 0.072 0.33 103        | 22 898 12122         | 459 2384 21 |
| bf_mcpr_8            | 246.6 645.6 66.06            | 0.13 0.70 279         | 512 4566 18121       | 145 359 17 |
| bf_mcpr_12           | 320.6 526.2 62.11            | 0.584 1.95 426        | 181 999 26645        | 847 977 9 |
| bf_mcpr_1            | 223.3 506.0 64.56            | 0.027 0.04 208        | 8 22 13441           | 22200 18105 21 |
| bf_mcpr_1            | 268.6 571.2 62.68            | 0.063 0.03 174        | 8 17 10906           | 21772 16085 24 |
| bf_mcpr_1            | 299.3 503.2 68.80            | 0.088 0.36 254        | 2 28 17554           | 4447 4373 16 |
| bf_mcpr_9            | 314.3 584.8 69.76            | - 74.53 623           | - 322 4635 0         | 0 25 |
| bf_mcpr_11           | 291.6 600.2 64.63            | 0.025 0.05 176        | 1 2 11245           | 20066 16749 28 |
| bf_mcpr_15           | 245.3 578.2 63.1             | 0.008 0.003 159       | 1 1 10090           | 20146 15165 24 |
| bf_mcpr_11           | 232 573 66.31               | 0.007 0.005 229       | 1 1 15137           | 17010 14493 15 |
| bf_mcpr_11           | 257.3 508.2 64.53            | - 182.13 783          | - 92356 9063        | 0 15 6 |
| oob_1_ari_1          | 279 598.8 71.86              | 2.06 0.14 55          | 556 83 3880          | 6121 6291 39 |
| oob_1_ari_6          | 308 591 77.03               | 0.025 1.39 103        | 14 82 8085          | 5541 6600 28 |
| oob_1_ari_9          | 284.6 571.2 69.78           | - 273.8 105           | - 135938 7326       | 0 12 27 |
| oob_1_ari_11         | 297.2 507 75.2              | 12.11 91.86 207      | 3564 49105 27241    | 254 686 19 |
| oob_2_ari_1           | 298.6 520.8 73.53           | - 154.42 117         | 80080 8558          | 0 12 35 |
| oob_2_ari_5           | 326.6 520.4 71.1            | - 155.62 165         | 80662 27259         | 0 16 27 |
| oob_2_ari_8           | 295.5 592.64 69.8           | - 102.97 188         | 60384 13566         | 0 12 22 |
| oob_2_ari_18          | 312.25 502.2 70.95          | - 97.86 192          | - 48894 13401       | 0 17 19 |

1. BeagleBone (ARM), 2. Linux x64 (Intel), 3. Wago PFC100 (ARM)

Challenges of black-box fuzzing. FieldFuzz does not require access to the controller or any modification, ensuring the universality and scalability of the 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 Layer 7 without requiring actual network transmission. This requires pre-loading a harness in the form of 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 the generalization of the approach.

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) It is rare to have a full-featured shell to control the device as many of the ICS devices embed the runtime on top of legacy RTOS or use the bare-metal variation of the runtime. Gaining the necessary capabilities for white-box fuzzing requires re-flashing the controller with a modified kernel image and relying on remote debugging.

As future work, we aim to explore the combination of these methodologies to leverage the advantages of both, providing a precise code coverage mechanism while preserving the black-box benefits of the FieldFuzz.

IX. CONCLUSION

In this paper, we present FieldFuzz—a fuzzing framework for control programs and industrial runtime, capable of discovering vulnerabilities in more than 400 known ICS devices from 80 industrial device vendors. FieldFuzz leverages two particular components: i) a Ghost application injected into the target (to provide fuzzing and coverage feedback), and ii) a runtime driver (FRD) to encapsulate the stateful protocol interactions with the target. We successfully fuzz the various instantiations (on different architectures and by different vendors) of the Codesys runtime, reporting three CVE IDs with a speedup of ≈8.3x compared to the state-of-the-art for IEC application binaries, and with increased crash discovery of ≈291x and ≈262x for 32 and 64 bit runtime variants, respectively. We perform fuzzing on both physical and virtualized ICS devices to prove its automation capabilities, reliability, and performance improvements against the current state-of-the-art in this domain. With FieldFuzz, we provide researchers with a powerful open-source tool to enable future research in this direction.

PUBLISHED TOOLS

We release FieldFuzz, FieldFuzz Runtime Driver, Ghost, and the Wireshark dissector for Codesys v3 protocol as open source tools.
source tools with this work.

REPORTED CVE IDs

As a result of this work, our reported vulnerabilities were assigned CVE-2021-34604, CVE-2022-22508, and CVE-2022-22507.

REFERENCES

[1] “CODESYS Runtime (Brochure),” Aug 2021, [Online; accessed 25. Aug. 2021]. [Online]. Available: https://www.codesys.com/products/codesys-runtime-control.html

[2] A. Abbasi, J. Wetzels, T. Holz, and S. Etalle, “Challenges in designing exploit mitigations for deeply embedded systems,” in 2019 IEEE European Symposium on Security and Privacy (EuroS&P). IEEE, 2019, pp. 31–46.

[3] H. J. Abdelnur, R. State, and O. Festor, “Kif: a stateful sip fuzzier,” in Proceedings of the 1st international Conference on Principles, Systems, and Applications of IP Telecommunications, 2007, pp. 47–56.

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

[5] G. Banks, M. Cova, V. Felmetsger, K. Almeroth, R. Kemmerer, and G. Vigna, “Snooze: toward a stateful network protocol fuzzier,” in International conference on information security. Springer, 2006, pp. 343–358.

[6] G. Canet, S. Couffin, J.-J. Lesage, A. Petit, and P. Schnoebele, “Towards the automatic verification of plc programs written in instruction list,” in Smc 2000 conference proceedings. 2000 ieee international conference on systems, man and cybernetics. “cybernetics evolving to systems, humans, organizations, and their complex interactions” (cat. no.0), vol. 4, 2000, pp. 2449–2454 vol.4.

[7] J. H. Castellanos, M. Ochoa, A. A. Cardenas, O. Arden, and J. Zhou, “Attkfinder: Discovering attack vectors in plc programs using information flow analysis,” in 24th International Symposium on Research in Security, Intrusions and Defenses, 2021, pp. 235–250.

[8] S. K. Cha, M. Woo, and D. Brunmley, “Program-adaptive mutational fuzzing,” in 2015 IEEE Symposium on Security and Privacy, 2015, pp. 725–741.

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

[10] A. A. Clements, E. Gustafson, T. Scharnowski, P. Grosen, D. Fritz, C. Kruegel, G. Vigna, S. Bagchi, and M. Payer, “HALucinator: Firmware re-hosting through abstraction layer emulation,” in 29th USENIX Security Symposium (USENIX Security 20). USENIX Association, Aug. 2020, pp. 1201–1218. [Online]. Available: https://www.usenix.org/conference/usenixsecurity20/presentation/clements

[11] CODESYS. “CmplcVarAccess Interface Description,” [Online; accessed 20. Aug. 2021]. [Online]. Available: https://help.codesys.com/webapp/ids-CmplcVarAccess_Ifs-lib;product=CmplcVarAccess_Interface;version=3.5.15.0

[12] D. Fang, Z. Song, L. Guan, P. Liu, A. Peng, K. Cheng, Y. Zheng, P. Liu, H. Zhu, and L. Sun, “I3csfuzzer: A framework for discovering protocol implementation bugs in ics supervisory software by fuzzing,” in Annual Computer Security Applications Conference, ser. ACSAC. New York, NY, USA: Association for Computing Machinery, 2021, p. 849–860. [Online]. Available: https://doi.org/10.1145/3485832.3488028

[13] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse, “AFL++: Combining incremental steps of fuzzing research,” in 14th USENIX Workshop on Offensive Technologies (WOOT 20). USENIX Association, Aug. 2020.

[14] L. Garcia, S. Zonouz, D. Wei, and L. P. De Aguiar, “Detecting plc control corruption via on-device runtime verification,” in 2016 Resilience Week (RWS). IEEE, 2016, pp. 67–72.

[15] L. A. Garcia, F. Brasser, M. H. Cinguga, A.-R. Sadeghi, and S. A. Zonouz, “Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit,” ResearchGate, Jan 2017.

[16] H. Gascon, C. Wressnigg, F. Yamaguchi, D. Apr, and K. Reck, “Pulsar: Stateful black-box fuzzing of proprietary network protocols,” in International Conference on Security and Privacy in Communication Systems. Springer, 2015, pp. 330–347.

[17] C. Group, “Codesys device directory,” [Online; Accessed September 2021]. [Online]. Available: https://www.codesys.com/download/download-center.html

[18] ——, “Integrating c modules,” [Online; Accessed: September 2021]. [Online]. Available: https://help.codesys.com/api-content/2/codesys/5.5.10/en/cds_integrating_c_code/

[19] S. Guo, M. Wu, and C. Wang, “Symbolic execution of programmable logic controller code,” in Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, ser. ESEC/FSE 2017. New York, NY, USA: Association for Computing Machinery, 2017, p. 326–336. [Online]. Available: https://doi.org/10.1145/3106257.3106245

[20] J. T. Hagen and B. E. Mullins, “Tep veto: A novel network attack and its application to scada protocols,” in 2013 IEEE PES Innovative Smart Grid Technologies Conference (ISGT), 2013, pp. 1–6.

[21] A. Helin, “Python bindings for libradamsa,” [Online ; Accessed: January 2022]. [Online]. Available: https://github.com/tsundokul/pyradamsa

[22] J. Homan, S. McBride, and R. Caldwell, “Irtogic ICS malware: Nothing to see here... masking malicious activity on SCADA systems,” FireEye threat research blog, 2016.

[23] B. Huang, A. A. Cardenas, and R. Baldick, “Not everything is dark and gloomy: Power grid protections against iot demand attacks,” in 28th USENIX Security Symposium (USENIX Security 19). Santa Clara, CA: USENIX Association, Aug. 2019, pp. 1115–1132. [Online]. Available: https://www.usenix.org/conference/usenixsecurity19/presentation/huang

[24] H. Janicke, A. Nicholson, S. Webber, and A. Cau, “Runtime-monitoring for industrial control systems,” Electronics, vol. 4, no. 4, pp. 995–1017, 2015.

[25] E. Johnson, M. Bland, Y. Zhu, J. Mason, S. Checkoway, S. Savage, and K. Levchenko, “Jetset: Targeted firmware rehosting for embedded systems,” in 30th USENIX Security Symposium (USENIX Security 21). USENIX Association, Aug. 2021, pp. 321–338. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/johnson

[26] JSOF Tech, “Ripple 20 – 19 day-zero vulnerabilities amplified by the supply chain,” 2020.

[27] S. Kalle, N. Ameen, H. Yoo, and I. Ahmed, “Clik on PLCs! attacking control logic with decompilation and virtual PLC,” in Binary Analysis Research (BAR) Workshop, Network and Distributed System Security Symposium (NDSS), 2019.

[28] A. Kelifir and M. Maniatakos, “Isref: A framework for automatic reverse engineering of industrial control systems binaries,” arXiv preprint arXiv:1812.03478, 2018.

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

[30] Z. Luo, F. Zuo, Y. Jiang, J. Gao, X. Jiao, and J. Sun, “Polar: Function code aware fuzz testing of ics protocol,” ACM Transactions on Embedded Computing Systems (TECS), vol. 18, no. 5s, pp. 1–22, 2019.

[31] Z. Luo, F. Zuo, Y. Shen, X. Jiao, W. Chang, and Y. Jiang, “Ics protocol fuzzing: coverage guided packet crack and generation,” in 2020 57th ACM/IEEE Design Automation Conference (DAC). IEEE, 2020, pp. 1–6.

[32] MITRE, “Cve list,” [Online ; Accessed: January 2022]. [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Codesys

[33] M. Niedermaier, F. Fischer, and A. von Bodisco, “Propfuzz—an industry-focused method for automated reverse engineering of industrial control systems binaries,” in 2017 International Conference on Applied Electronics (AE). IEEE, 2017, pp. 1–4.

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

[35] ——, “Security research: CODESYS Runtime, a PLC control framework. Part 2,” Sep 2019, [Online; accessed 20. Aug. 2021]. [Online]. Available: https://ics-cert.kaspersky.com/reports/2019/09/18/security-research-codesys-runtime-a-plc-control-framework-part-2

[36] V.-T. Pham, M. Böhme, and A. Roychoudhury, “Aflnet: a greybox fuzzer for network protocols,” in 2020 IEEE 15th International Conference on
[37] O. A. V. Ravnás, “Frida: Dynamic instrumentation toolkit for developers, reverse-engineers, and security researchers.” [Online; accessed 26. Jul. 2022]. [Online]. Available: https://frida.re/

[38] A. Serhane, M. Raad, R. Raad, and W. Susilo, “Plc code-level vulnerabilities,” in 2018 International Conference on Computer and Applications (ICCA), 2018, pp. 348–352.

[39] W. Systems, “CodeMeter from Wibu-Systems,” [Online; accessed 20. Aug. 2021]. [Online]. Available: https://www.wibu.com/products/codemeter.html

[40] M. Tiegelkamp and K.-H. John, IEC 61131-3: Programming industrial automation systems. Springer, 2010.

[41] D. Tychalas, H. Benkraouda, and M. 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). USENIX Association, Aug. 2021, pp. 2847–2862. [Online]. Available: https://www.usenix.org/conference/usenixsecurity21/presentation/tychalas

[42] D. Urbina, J. Giraldo, N. O. Tippenhauer, and A. Cardenas, “Attacking fieldbus communications in ics: Applications to the swat testbed,” in Proceedings of the Singapore Cyber-Security Conference (SG-CRC) 2016. IOS Press, 2016, pp. 75–89.

[43] M. Zhang, C.-Y. Chen, B.-C. Kao, Y. Qamsane, Y. Shao, Y. Lin, E. Shi, S. Mohan, K. Barton, J. Moyne, and Z. M. Mao, “Towards automated safety vetting of plc code in real-world plants,” in 2019 IEEE Symposium on Security and Privacy (SP), 2019, pp. 522–538.

[44] F. Zuo, Z. Luo, J. Yu, Z. Liu, and Y. Jiang, “Pavfuzz: State-sensitive fuzz testing of protocols in autonomous vehicles,” in 2021 58th ACM/IEEE Design Automation Conference (DAC), 2021, pp. 823–828.