ICSFuzz: Manipulating I/Os and Repurposing Binary Code to Enable Instrumented Fuzzing in ICS Control Applications

Dimitrios Tychalas[1], Hadjer Benkraouda[2] and Michail Maniatakos[2]

[1]NYU Tandon School of Engineering, Brooklyn, NY, USA
[2]New York University Abu Dhabi, Abu Dhabi, UAE

Abstract

Industrial Control Systems (ICS) have seen a rapid proliferation in the last decade amplified by the advent of the 4th Industrial Revolution. At the same time, several notable cybersecurity incidents in industrial environments have underlined the lack of depth in security evaluation of industrial devices such as Programmable Logic Controllers (PLC). Modern PLCs are based on widely used microprocessors and deploy commodity operating systems (e.g., ARM on Linux). Thus, threats from the information technology domain can be readily ported to industrial environments. PLC application binaries in particular have never been considered as regular programs able to introduce traditional security threats, such as buffer overflows. In this work, we investigate the feasibility of exploiting PLC binaries as well as their surrounding PLC-specific environment. We examine binaries produced by all available IEC 61131-3 control system programming languages for compilation-based differences and introduced vulnerabilities. Driven by this analysis, we develop a fuzzing framework to perform security evaluation of the PLC binaries along with the host functions they interact with. Fuzzing such non-executable binaries is non-trivial, as they operate with real-time constraints and receive their inputs from peripherals. To prove the correctness of our fuzzing tool, we use a database of in-house developed binaries in addition to functional control applications collected from online repositories. We showcase the efficacy of our technique by demonstrating uncovered vulnerabilities in both control application binaries and their runtime system. Furthermore, we demonstrate an exploitation methodology for an in-house as well as a regular control binary, based on the uncovered vulnerabilities.

1 Introduction

Industrial Control Systems are an integral part of modern society, witnessing a rapid expansion in the era of Industry 4.0 [31]. Computerized control systems are solely responsible for moderating a wide range of industrial sectors, including critical infrastructures such as power grids, oil and gas industries, transportation and water treatment facilities. Possible suspension of their operation may cause severe complications which translate to significant loss of revenue, environmental disasters or even human casualties [22, 47].

Since safety and security are tightly correlated in ICS environments, the recent cyberattacks that targeted industrial settings have showcased tangible threats to contemporary industrial environments. The most prominent of these incidents, Stuxnet [30], drew widespread attention when discovered, as the first potential case of cyberwarfare. In the following years more incidents including the 2015/2016 cyberattacks on the Ukrainian power grid [33, 62] and the 2017 attack on petrochemical facilities in Saudi Arabia [42] have helped piece together a pattern of industrial exploitation through cybersecurity, not a simple observation of isolated incidents. From a financial standpoint, cyberattacks average an estimated $350,000 in damages per attack, reaching $500,000 for larger companies [40]. Consequently, the ICS cybersecurity market is rapidly expanding, from a moderate market size of $1.5 billion in 2018 to a projected $7 billion by 2024 [49].

A primary contributor to this increase in ICS cybersecurity incidents is the ongoing merge of traditional Operational Technology (OT) with modern Information Technology (IT). Opting for cost reduction, increased flexibility and adaptability of ICS devices, manufacturers turn to established software solutions contrary to the traditional in-house firmware development model. As a result, general-purpose user and system software deployment in ICS has experienced rapid expansion, with solutions such as Embedded Linux and Nucleus OS becoming a popular alternative to aged monolithic firmware designs [21]. This integration does not come without consequences though, since traditional threats to IT systems leak over to ICS environments [54]. Such threats have been reported at an increasing rate during the past years [16] exposing this pattern to the ICS community and at the same time attracting attention to an alarming situation.

ICS perform functions on physical processes through Programmable Logic Controllers (PLC), dedicated computing

platforms whose main purpose revolves around receiving, processing and transmitting arithmetic and logic values pertinent to the process itself, such as temperature, pressure etc. The process engineer develops this *control logic* in an assortment of specialized programming languages, standardized under IEC 61131-3 [51], which in turn is compiled into the *control application binary* [1] that performs the control function. The binaries themselves have been the target of numerous research efforts, initially targeting the correct function of the binary through verification techniques [14, 15, 41] and evolving into security evaluation the last few years [19, 38, 50].

All efforts though focus on the logic process, detecting whether it performs correctly or if it introduces threats to the host industrial setting through modification or exploitation. These binaries, however, are merely a collection of assembly instructions subject to programming or compile-induced errors [6, 12, 59]. To the best of our knowledge, security evaluation of control applications *as typical applications* and the potential effect on their host system remains an open problem. Since the host in many cases can be a general-purpose OS, the exploitation of a locally executed binary can lead to system compromise, with situations varying from denial of service to full system seizure.

PLC application building tools have traditionally been restricted to each manufacturer, with industry leaders such as Allen-Bradley and Siemens providing full-stack development frameworks for their specific products with the exception of Codesys which is compatible with multiple platforms for various vendors. These tools are typically closed-source with limited documentation which inherently increases the difficulty in evaluating their function from the security research community. This includes the control application compilation process, which is a similarly unknown factor, where research and practice have showcased tangible threats induced to programs during compile-time [34, 36].

Similarly, the PLC application binaries themselves are built in unique formats based on the vendor. As such, dynamic analysis of these binaries must be performed in a case-by-case basis, without the possibility of a universal approach or automation. Symbolic execution has been proposed as a potent evaluation methodology [23], however due to the unique binary format researchers followed an indirect way to enable such an analysis by translating bytecode to a high-level general-purpose program language. Fuzzing is equally insufficient in handling control applications, with no published or recorded use cases in academic literature or practice: Control binaries follow a specialized execution process handled by their runtime environment and their input delivery is bound by real-time constraints.

In this paper we showcase a novel approach for security evaluation of PLC control applications through fuzzing. Our approach is initiated by a manual research step for a comprehensive assessment of the various IEC 61131-3 languages and the effect their unique features and compilation techniques impose on the produced binaries. Driven by this assessment, we develop a fuzzing framework for discovering potential vulnerabilities in PLC applications and their host software. Our work is a first effort on assessing a unique class of binaries with a contemporary evaluation method such as fuzzing.

In summary, our contributions are as follows:

- We analyze the composition of control applications based on all available IEC 61131-3 languages, highlighting the unique characteristics and intricacies introduced by different languages and compiling tools.
- We develop a fuzzing framework for PLC applications to uncover existing binary vulnerabilities that would lead to crash or exploitation.
- We extend our fuzzing framework to include system functions that belong to the host software of the PLC application.
- We develop and consolidate a collection of vulnerable PLC binaries which can be used by future researchers in industrial control systems security.
- We demonstrate the usefulness of our methodology by uncovering vulnerabilities in both synthetic and regular control binaries and we demonstrate exploitation methodologies to compromise the host system and/or the industrial process.

2 Preliminaries

2.1 Problem Formulation

The principal questions we answer are:

- Given that PLC binaries are compiled from high-level PLC programming languages using proprietary compilers, can security vulnerabilities be introduced?
- PLC binaries are loaded by proprietary runtimes executing as a process of the PLC operating system. Given that these runtimes are compiled from regular C/C++ source code, how vulnerable are they?
- Given that the PLC binaries execute bounded by real-time constraints and with heavy use of GPIO, can fuzzing be leveraged for uncovering potential vulnerabilities?

Since PLCs are increasingly evolving to common general-purpose computing systems, the existence of a traditional vulnerability, e.g. a stack derived buffer overflow in an otherwise correctly functioning code sequence, introduces additional threats that could lead to system compromise or termination of operation. These situations are added on top of possible operations derived and network induced threats that have been a popular topic of ICS security research in the past few years [17, 50, 57].

[1] The terms *control application*, *control binary* and *PLC binary* will be used interchangeably throughout the paper.

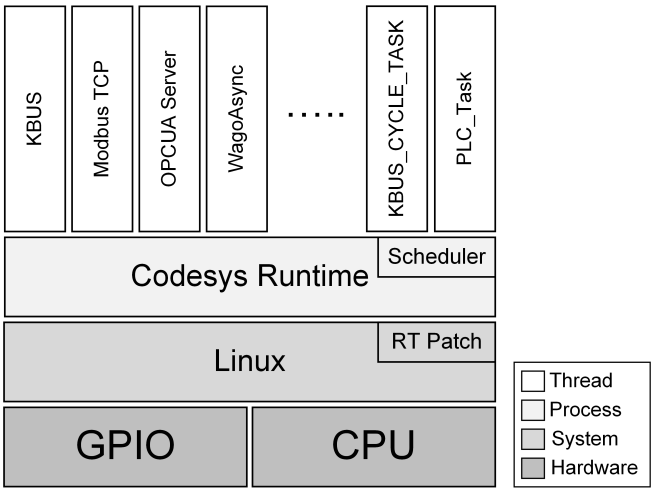


Figure 1: Codesys-based PLC software stack.

2.2 Threat Model

The assumed scenario is as follows: An industrial setting where a PLC is controlling an operational function, receiving inputs from sensors and ensuring correct operation through actuators, and is a host to a program which is responsible for mediating the application of control logic. The device is network connected to provide access to engineers through a Human Machine Interface (HMI). Sensors and actuators also belong to the network of the industrial setting, communicating directly with the device. In our threat model, the attacker can deliver some form of input data to the target PLC. Examples of input data delivery are as follows:

- Man-in-the-middle attack on network packages carrying values delivered to the PLC binary from an HMI terminal. The HMI side of the Codesys framework has the capability to force values on any variable present in the PLC binary in real time [24].
- Man-in-the-middle attack on the sensor that provides the input. This type of attack has been explored in depth in many ICS-related publications [57].
- Firmware trojan that can covertly enable data manipulation [52, 53]. Firmware modification has been identified as one of the principal exploitations in the 2014 attack on the Ukrainian power grid [8].
- Evil maid scenario in which a malicious insider can manipulate values on the PLC binary through an unsupervised HMI terminal [1].

2.3 Codesys Runtime

Codesys is a multi-platform development environment for programming control applications according to the international industrial standard IEC 61131-3. We chose Codesys as our initial target due to the popularity the platform has gained in recent years as well as its multi-platform compatibility. Over 250 manufacturers from diverse industrial sectors include the Codesys platform to their ICS products. The Codesys device directory [11] lists over 400 devices capable of supporting the platform from leading manufacturers. Due to the lack of readily accessible data regarding market shares of these manufacturers, it is difficult to assert a definitive percentage of Codesys devices currently deployed. From empirical data, conversations with experts, and the Shodan search engine [37], we can conservatively approximate a minimum 20% of PLC worldwide utilizing Codesys, although the actual percentage could be much higher.

Thread name	Function	Interaction
Codesys3	Main process	System
KBUS dbus	System monitored comm. bus	System
ModbusSlaveTCP	Modbus TCP comm.	Network
0ms_Watch_Threa	Peripheral event polling	System
WagoIpcMsgComm	Inter-process comm.	System
Schedule	Runtime Scheduler	System
OPCUAServer	Machine to machine comm.	Network
WagoAsyncRt	High priority CAN comm.	Network
WagoAsync	Regular CAN comm.	Network
KBUS_CYCLE_TASK	Scan cycle	ControlApp
PLC_Task	Control Application	ControlApp
VISU_TASK	Visualization module	User

Table 1: List of most active threads attached to the Codesys runtime process, along with their function and the principal entity they interact with.

The Codesys runtime framework[2] is the back-end handler of all functions pertinent to the control binary itself as well as utilities for system and user interaction. Figure 1 illustrates the system stack of a Codesys based PLC. The runtime itself is a self-contained `ELF` binary which resides in the `/usr/bin` folder and is being deployed through a wrapper process as a part of the OS initialization, contained in the `etc/init.d` boot scripts. Following its invocation from the wrapper process, the runtime enters an initialization phase launching an array of communication-related functions, such as network, peripheral, and inter-process. All functions are instantiated as threads, children of the main runtime process, spawned through the `clone()` system call. One of the more interesting functions, `KBUS`, is a lightweight inter-process communication system used to relay data throughout the runtime threads and, more importantly, handles the control application data from GPIO ports to the application itself. A rudimentary scheduler is also instantiated, mainly to facilitate the execution of the control applications. It handles priority assignments, keeps track of used mutexes between the control application-related threads, handles exceptions generated by the control application and resolves watchdog-related exceptions. Table 1 lists a selection of the active threads under the Codesys runtime along with their primary functionality.

The control application loading process begins during the runtime initialization with a file-open system call to a hard-coded folder location where the application binary resides.

[2]Codesys runtime framework will be mentioned simply as runtime for the rest of the paper.

The runtime then begins copying the control application code and data in memory. Following the complete memory loading of application file, execution is initiated, handled by a customized set of functions based on the `pthread` API. The control binary code is pushed to the stack of a newly instantiated thread, forcing execution privileges to be enabled on stack segments across all the main process threads. This action can be a primary enabler for arbitrary code execution, a notion which we explore more in depth in Section 5. Along with the control application loading/execution, the runtime spawns two more utility threads pertinent to the application itself, the `KBUS_CYCLE` which acts as a mediator between the application and `KBUS`, and `VISU` which offers visualization of the control process based on information embedded to its source code.

2.4 Control Application Binaries

Although control binaries share some general similarities with conventional binary file formats, such as ELF and PE, they are ultimately different. Control binaries, like conventional computer program binaries, are composed of a header, a main program, a data section, and linked libraries both statically and dynamically. The main difference between these binary file formats is that control binaries are not independently executable. As mentioned before, this is the biggest challenge in our security analysis.

In [28] the authors offer a concise view of the Codesys compiled control application binary format through reverse engineering.

Here we only describe the sections of the binary file that might have security implications. The file starts with a header section containing critical information for the run-time to enable its execution. Most importantly it contains the program's entry point, stack size and the last dynamic library identifier. The header is followed by a subroutine that sets constant variables and initializes functions used within the global variable section within the IDE. Another important section of the binary is the debugger handler subroutine that enables dynamic debugging from the IDE.

Control binaries also contain calls to Functions or Function Blocks (F/FB) from libraries and user-defined F/FB. Both of these are statically linked and are included in the binary file in the format of two consecutive subroutines. The first contains the instructions that represent the functionality of the F/FB and the second initializes its local memory. Next, the main function of the PLC (`PLC_PRG`) is encapsulated in the next subroutine. This subroutine is the most interesting component, since it includes the control logic. Dynamically linked functions within the control binaries are resolved through a symbol table that is located after the last code subroutine. The symbol table contains two bytes of data that are used by the run-time to calculate the jump offset required for calling the corresponding function.

3 Control Application Analysis

The field of security analysis of PLC control applications has started with the assumption that these binaries are susceptible to attacks, but have not closely investigated as to how. In this section, we aim to establish whether PLC programming languages are secure in terms of memory operations.

3.1 PLC programming languages

Control applications for PLC can be developed in different languages, both graphical and textual. These are high-level domain-specific programming languages for developing control application software. Historically, many PLC companies utilized proprietary programming languages. In recent years however, and in an effort to standardize the programming languages used by PLC vendors, the International Electrotechnical Commission (IEC) has established the IEC 61131-3 standard. This standard outlines the software architecture and programming of PLC by defining programming languages, data types and variable attribution [51]. We exclude Instruction List (IL), since it is an inactive language, and Sequential Function Charts (SFC) language given it is composed of calls to other PLC programming languages and therefore does not have intrinsic characteristics:

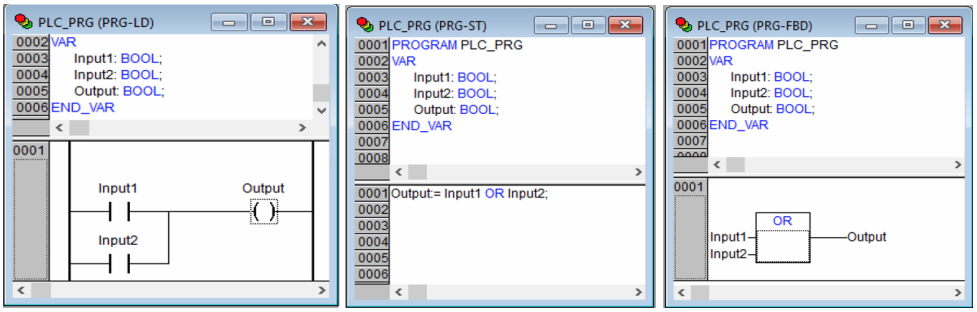
- **Ladder Diagram (LD)**: (Graphical) This language resembles electric circuits and replaces hardwired relay control systems. Since LDs deal with fundamental components (i.e. contacts and coils), representing large contemporary systems and maintaining visual comprehensiveness becomes hard. LD also lacks native support for arithmetic operations and data structures such as arrays and pointers.
- **Function Block Diagram (FBD)**: (Graphical) FBD is also based on a wiring diagram that links Function Blocks (FB). FBs are programming constructs used by PLC programming languages in the same way functions are used in conventional programming languages.
- **Structured Text (ST)**: (Text-based) This language is the closest to high-level computer programming languages and is based on Pascal. It uses conditional statements, loops and similar data structures such as pointers and arrays.

Figure 2 visualizes the differences between the three languages implementing the same logic.

3.2 Comparing programming languages

Comparing languages aims to investigate the necessity for independent analysis of each language. In addition, we have to understand the sources of similarities between the various PLC languages in order to investigate language-based security and the security mechanisms enabled by them.

We start by looking at the binary files produced by each language. Initial automated analysis using diffing tools (e.g. `vbindiff`) showed that different languages produced diverse



LD ST FBD

Figure 2: PLC programs written in different PLC programming languages implementing a 2-input OR Gate.

binary files. Further investigation uncovered that the main source of disparity is the fact that the compiler inserts a variable number of No-operation instructions (NOPs) for different PLC languages. It is interesting to note that there were different variations of NOPs, and most of them did not use the reserved opcode for NOPs, but were rather composed of typical instructions performing redundant tasks (e.g. `mov r0, r0`). NOP addition is typically used in embedded systems to introduce intentional delays for timing purposes such as memory load/store in order to avoid potential problems arising from non-deterministic memory access. PLC binaries are optimized for reliability and not for performance.

The evaluation of different languages on both Codesys versions establishes that the languages produce similar machine code, hinting that Codesys produces an intermediate representation before generating the final machine code. Therefore, for the rest of the paper, we focus on one language when performing binary code analysis on compiled PLC control applications. We specifically select ST because it provides extra data structures and functions to the user, such as pointers and loops.

3.3 Potentially vulnerable functions in PLC applications

In this subsection, we compare vulnerable functions from `C/C++` to those found in PLC programming environments [10] to establish whether PLC languages are memory secure. In standard programming languages, memory hazards stem from the ability to directly access memory from the program. PLC programs also allow memory manipulation through pointers. Table 2 shows the list of functions analyzed in our experiments. String operations listed in the second column of Table 2 in standard programming languages like `C++` are well known for potential security vulnerabilities.

Our analysis includes an array of functions including `SysStrCpy` (`SysLibStr` library), `Concat` (`Standard library`), as well as `SysMemCpy`, `SysMemMove`, `SysMemSet`, and `SysMemCmp`, all part of the Codesys `SysLibMem` library.

Our analysis concluded that a subset of the tested functions retain their inherent vulnerabilities and led to crash instances.

	C/C++	Codesys 2.x and 3.x		
	Function Name	Function Name	Bounds Check	Crash
String Operations	strcpy()	SysStrCpy()	✗	✗
	strcat()	Concat()	✓	✗
Memory Operations	memcpy()	SysMemCpy()	✗	✓
	memset()	SysMemSet()	✗	✓
	memmove()	SysMemMove()	✗	✓
	memcmp()	SysMemCmp()	✗	✗

Table 2: Potentially vulnerable functions in conventional and PLC programming languages.

This observation guided us in our development of synthetic PLC binaries in Section 5. The cause behind the crashes is variable based on the function. For example, `SysMemCpy` and `SysMemMove` do not compare the sizes of source and destination buffers and are therefore vulnerable to potential crash leading buffer overflows.

4 Fuzzing Industrial Control Systems

In this section we present the technical details of our methodology for performing security evaluation of control applications by fuzz testing. Fuzz testing, or simply fuzzing, is an automated program testing methodology, initially utilized as a brute force binary testing technique in the early 90's [4]. This primary approach involved repeatedly feeding random data to the target binary observing its behavior through the host system. Fuzzing has largely evolved during the last thirty years, becoming a popular method for software-based vulnerability discovery, becoming "smarter" and more efficient with techniques such as binary instrumentation and input mutation.

Our developed framework creates instances that will carry out automated testing and evaluation of Codesys-compiled control applications through fuzzing. We have deployed symbolic execution to analyze the binary and perform instrumentation to facilitate code coverage extraction during fuzzing. We have also targeted runtime-hosted functions that interact with the binary and added them to our fuzzing framework.

4.1 Fuzzing Control Applications

Control applications as analyzed in previous sections lack binary standardization specifics and follow a different execution mechanism than typical executable files. These factors introduce an array of complications in fuzzing attempts especially in automating the process:

- Execution cannot be directly controlled through typical system calls such as `execve`.
- Execution failure does not produce feedback information for further analysis.
- Input cannot be relayed to the control application through conventional means, e.g. a file or the `stdio`.
- Input delivery cannot be easily synchronized due to the scan cycle execution format of the control binaries.
- Instrumentation cannot be conventionally applied to the binaries given the lack of such an option in the closed-source compilation tools.

Fuzzing setup can be broken down to two major components: Execution control and input generation. The first part handles the execution, communicating with the system to initiate executions and receiving regular/unexpected execution termination signals. The second part moderates the input data which will be fed to the binary and cause execution flow deviations which can lead to unexpected/unrecoverable states.

Execution Control: Control binaries are broadly categorized into two classes, concerning their execution process: Synchronous or asynchronous. Synchronous control binaries follow the scan cycle model in which a binary periodically checks a predetermined memory-mapped address for input updates, performs operations based on the received values and writes to a relevant output address. Asynchronous control binaries can receive external signals for input updates or termination, being able to be executed only once.

Synchronous (fixed cycle) programs are the most straightforward when it comes to execution but offer little flexibility in controlling them. This limits the maximum potential for fuzzing, since controlled execution is a primary contributor to efficiency. Thus, for these binary types, our approach is tightly controlling input delivery to take advantage of every available execution cycle in a specified time frame. Asynchronous programs offer direct execution control by manipulating the input signal which initiates and/or terminates their operation. By manipulating this signal we can jumpstart an execution instance and monitor the corresponding termination signal so the next execution can be initiated, which allows for a more traditional fuzzing process. However, asynchronous programs are specialized cases for PLC programs making up a fraction of available binaries, limiting the applicability of this approach.

The primary objective of any fuzzing instance is the detection of an unexpected termination of the binary execution. In a typical OS, system signals such as `SIGSEGV`, relay an asynchronous exit from a program execution due to a particular fault which is followed by the call of an exception handling function. In our case, the execution termination of the control process is silent as far as the system is concerned, where the scheduler thread is handling the termination of the control process. The sole information visible to the host system is a series of `futex` system calls which suspend the connected threads `VISU` and `KBUS_CYCLE` by essentially leading them to a deadlock. However, the control application thread termination can be monitored from the parent process or an immediate ancestor. The Codesys runtime initialization is handled by a simple wrapper script that launches the runtime by simply invoking it. We have modified the script so the runtime launch is being handled by forking the wrapper process. This ensures that the runtime, and by extension the control process, process ID's (`pid`'s) are in the same family as the wrapper. Then we utilize `wait()` to suspend the wrapper until it receives a termination signal from a child process with a chosen `pid` leading to the termination and reinitialization of the runtime process.

Input Control: Input control comes up as the most critical part of the fuzzing process since execution control is limited. Inputs are typically physical signals, analog or digital, which are received by a specialized peripheral, the I/O module. This device connects to the physical process, e.g. the rotation of a steel mill, and transforms the physical signals to usable information delivered to the PLC through a "GPIO" labeled Linux device which communicates with the the runtime process via the `KBUS` subsystem. `KBUS` then forwards the input data to the control application each scan cycle. In summary, input delivery follows this flow:

1. An I/O module receives a signal from a sensor and relays it to the PLC through GPIO.
2. GPIO receive and store the input data in their memory-mapped space.
3. `KBUS` opens the GPIO device file and performs a read system call, moving the input data to its own memory space, within the runtime process.
4. `KBUS_CYCLE_TASK`, the thread spawned alongside the control process, delivers the input data to the control process memory space through a write system call. This event is repeatable with a period based on the scan cycle length of the control application.

Fig. 3 illustrates the flow of input data from the I/O module to the control application.

The GPIO device, however, is not a simple generic I/O device available to most embedded devices, rather a custom device file responsible for handling the sensor input. Through reverse-engineering and debugging we have approximated its functionality, which mimics a GPIO as far as input data handling. Its interaction with the system however, is not the same as a typical GPIO: Custom system calls are being utilized to relay data instead of typical read/writes. This hinders a possible attempt for replicating its function through techniques such as emulation. In addition, data can be delivered

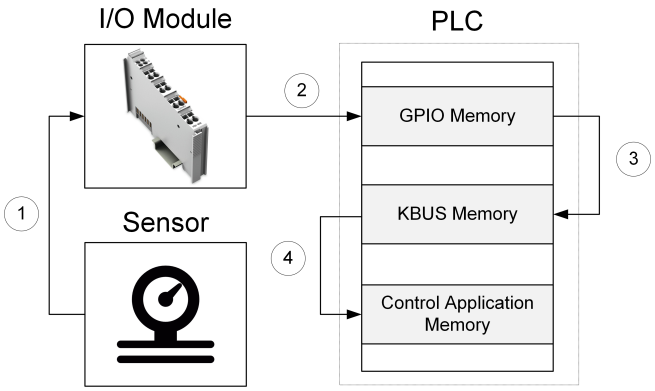


Figure 3: Data flow from sensors to the control application.

to the device bit-by-bit, depending on the I/O module, which must be manually reconstructed, a procedure which is highly unreliable. `KBUS` is a more appealing choice for input interaction, since the input has already been received and stored as a numeric value. The address space of `KBUS` can be extracted through memory-mapping information available at `/proc/maps` in Linux-enabled devices. `KBUS` is also instantiated as a device in the Linux file system, accessible at the `/dev` folder, being offered as a communication channel throughout the system.

The expected peripheral-received inputs are useful to build an initial test case corpus for fuzzing. Depending on the scan cycle length, a variable number of inputs are acquired within a specified time frame. By monitoring `KBUS` for system calls, we can determine the length of the scan cycle with high precision. Based on this we can tailor a collection rate for any target control application.

Since execution cannot be directly controlled or at least initiated, input relaying must also follow an indirect methodology. Manipulating the actual input of an already executing control process is the only means of input control for fuzzing. As analyzed in a previous section, the `KBUS_CYCLE` thread implements the scan cycle, and repeatedly copies data to the control process. This is facilitated by an `ioctl` system call with the `_IOC_WRITE` macro, which copies the GPIO-received data to the process data space. When intercepted, the write `ioctl` includes all necessary information for the copied data, including the destination address. The address follows a virtualization scheme applied to the process and, by extension, to its hosted threads. With the target virtual address and the process-specific mapping information we can directly write on the control process input data section.

This process, however, cannot be easily synchronized. While the scan cycle duration is known to the programmer and can be approximated by the tester, the exact timing of input arrival to the peripheral is unknown. Thus, the values we force in the designated I/O address space will be overwritten as soon as the regular input arrives. The runtime process hosting the control application does not follow regular communication with the host system through system calls: Instead, the runtime itself is implementing a subset of system-like functions, handled at the process level. The runtime process handles that by being launched at an elevated privilege level, requiring root access. Without information on the communication between the host system and the runtime process, simply manipulating the input values is ineffective.

Synchronization depends on an input-related signal which can be intercepted before the input is relayed to the control application. Given its role in input delivery, `KBUS` can be leveraged to get the necessary signal. The write `ioctl` we used in the previous paragraph was considered for triggering our input delivery, however during testing we understood that the time difference between the write command and data utilization by the control process was too short, resulting in our forced values coming after the input was accessed by the process. However, when traced for system interaction, `KBUS` is also periodically invoking a read `ioctl` system call with the `IOREAD` macro and the 8-bit selector pointing to a specific GPIO port. This can be translated to a string of data copied from the GPIO port to a select local address which will in turn be relayed to the control process itself. This system call is leveraged as a trigger by monitoring the sequence of system calls through a lightweight tracing utility, such as the `audit` subsystem. Making use of `audit` allows a user to gather information for a system call during its entry, resolution, or exit, thus offering high response to a potential system call-based condition such as our `ioctl` trigger. This prompt notification of the input data transfer system call in addition to a functional delay before the next scan cycle is initiated, allows us to successfully overwrite the original input with our chosen data.

Instrumentation: In addition to the main fuzzing functionality we need a feedback mechanism to provide us with execution-related information so we can estimate the efficiency of our methodology. Code instrumentation has been routinely used in contemporary fuzzing tools, added as a modification to regular compilers. In the case of control applications however, with no access to the compilation tools there is no straightforward way to instrument the compiled code. Symbolic execution though, which has been utilized as a facilitator to instrumentation for fuzzing [63], has been explored as a solution for control application analysis [28]. We can leverage knowledge and techniques for symbolic execution of control applications to gain program structure information so we can introduce instrumentation on the compiled binary.

We have utilized `angr` [46] for handling this part of our methodology leveraging its ability to partially execute specific portions of the program without necessarily being aware of the system state. As discussed in prior sections though, the control binaries are not built with a well-defined header section which, in typical files such as `elf`, includes information on file type, target architecture, and entry point. One prerequisite to perform static analysis through `angr` is to determine the

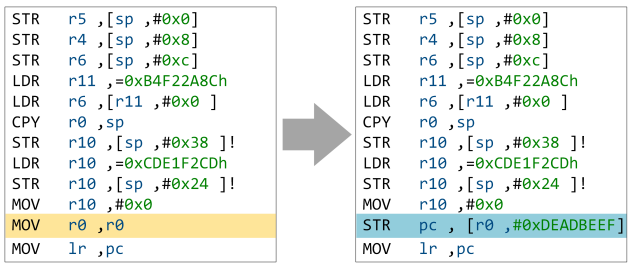


Figure 4: Excerpt from disassembled control application highlighting the NOP instruction substitution.

target platform of the chosen binary, in terms of architecture and endianness. To achieve that, we built on previous work on firmware reverse engineering and file format analysis [13, 60].

The main objective of our instrumentation methodology is to dissect the control binary and divide it into distinct segments in order to statically mark them for gaining execution-time feedback from the binary. We leverage the methodology introduced in [28] to produce a Control Flow Graph (CFG) of the file's functions along with the links between them, including relevant information on the function or function block code composition. More specifically, we perform an instruction count targeting NOPs, such as a `mov $r0, $r0`, logging the number of instantiations for each function block.

As the static analysis part of our research illustrated, NOPs are regularly added to the compiled code. However, since we lack the original compilation tool, we cannot fully understand their purpose. Nevertheless, in our experiments we observed that different languages introduce a variable number of NOPs for same architecture. NOPs have been a staple in older architectures for limiting potential timing errors by providing buffering for long instructions in simple pipelines. The variant number of NOPs for different languages and the capability of ARM [3] processors to remove a NOP from the pipeline before execution, are suggesting that these instructions are placeholders and not execution stallers or performance facilitators. To that end we performed an analysis of NOP placement in the code, relative to its immediate neighbor instructions. Delay intense instructions, such as `lw`, `mul` and `div`, are common choices when introducing pipeline stalls through compilation. We developed an array of control applications of different computation complexity, ranging from simple logical operations to heavy integral/differential calculations, to confirm whether NOPs will be added for functional reasons, in order to enhance execution reliability. Our results showcase that the amount of NOPs introduced is a direct factor of application size rather than operation complexity as initially assumed. The operation-intense programs were larger and had proportionally larger amount of NOPs compared to the simpler ones. Indeed, peering closer to the machine code, NOPs were not added in proximity to time-consuming instructions, such as

[3] ARMv6 and later architectures

	NX Bit	PIE/ ASLR	Stack Gaurd	RELRO	RUNPATH	RPATH	Fortify Source
Codesys 2.x	✗	✗	✗	✗	✗	✓	✗
Codesys 3.x	✓	✗	✗	✗	✗	✗	✗

Table 3: Summary of security mechanisms deployed by different versions of the Codesys runtime.

the ones mentioned above, rather close to simple `move` instructions often before a function call. This leads us to our conclusion that NOPs are introduced in control application mainly for memory alignment rather than hazard prevention. Thus, we can repurpose NOPs for program profiling and instrumentation by statically replacing NOPs with relevant instructions. The low amount of available slots in the compiled code does not offer much flexibility for instrumentation feedback. However, with access to the source code, redundant variable assignments with distinct values, e.g. `0xDEADBEEF` can increase the number of available instrumentation slots and provide more fine-grained feedback.

Typically instrumented binaries produce information through some user-available output, such as `stdout`, in our case however just one instruction slot must be able to relay the necessary information. Since we are aware of the memory layout of our process, we add `STR` instruction which stores the current program counter to predetermined addresses within the process. With the program counter information we can determine which function or function block has been accessed and approximate a code coverage percentage during our fuzzing sessions. Fig. 4 illustrates this substitution on an excerpt from a control application.

4.2 Fuzzing the Runtime

The runtime, as discussed in Section 2, is a traditional ELF binary that loads and executes the control application binary by spawning it as one of its threads. Typically when a thread is spawned through the `clone()` system call with the `CLONE_VM` flag set, any memory-mapping performed with `mmap()` affects both process and thread. This inter-dependency between the two binaries, the runtime and the control application binary, means that their security is intrinsically linked. It is, therefore, important to analyze the security measures employed by both the runtime binary and the control application itself in order to decide whether we should include it in the fuzzing process.

We use `checksec` [48] to investigate security features available in Codesys 2.x and Codesys 3.x runtime binaries. Table 3 summarizes the results. All versions of Codesys runtime implement minimal security mechanisms. As an improvement, Codesys 3.x includes NX bit support protecting against simple buffer overflow attacks, currently stemming from the support subsystems such as the network stack (involved in the CVE-2012-6068 and CVE-2012-6069 vulnerabilities).

The runtime is a complex application, working like a nested firmware inside the host OS. It includes a vast number of util-

ity functions to interact with its environment, perform maintenance, and handle and communicate with the control application. The application cannot be considered a standalone piece of software, as it can only exist in the context of the runtime, sharing the same memory space. Many functions and utilities have direct contact with the control application and are affecting its execution state. It is therefore prudent to consider these functions as an extension of the control application and include them as targets in the fuzzing framework.

The runtime exists in the OS as an `ELF` binary, leveraging various dynamic libraries, both system provided as well as in-house developed/modified. The main binary itself as a standalone target is not a good choice for any type of dynamic analysis. The complexity of its functionality, with more than 1000 functions being engaged just for maintenance and communications and nearly 5000 functions in total, renders any type of static analysis infeasible. In addition, the lack of source code deters any attempt at compile-time binary assessment supporting techniques such as instrumentation and address sanitization. An active PIE flag could offer the possibility of code migration enabling library extraction and execution outside the original binary, which does not hold for our versions of the runtime. Therefore, our only possibility for fuzzing anything related to the runtime is through the dynamically linked libraries, shared objects (`.so`) in Linux, and their included functions.

```
1
2 #include LIBDKBUSCOMMON_H
3
4 int main(int argc, char **argv){
5
6     kbus_ksock_t ksock = fopen("/dev/kbus0", "r");
7     [...]
8     kbus_ksock_write_data(ksock, &argv, 32);
9     [...]
10 }
```

Listing 1: Sample code of Codesys function fuzzing harness.

Fuzzing `.so`'s can be very challenging, considering the lack of information available for them. Since the library itself is not a valid target for execution, it must be hosted in an external program, a test harness, which dynamically loads it and declares some included function. The code has to be short and concise to maximize performance and simplify debugging when analyzing crash instances. With the help of `ghidra` [39] we are also able to extract some decompiled intro and outro code from the runtime compiled code so we can emulate the behavior of the chosen function in the context of the runtime. Regarding input arguments, we could find documentation for some functions, at least in terms of their calling convention. For the missing functions we once more deployed `ghidra` to reverse engineer function input arguments and derive their type. For `undefined` input types in functions without documentation, which can be a pointer to a struct or a `typedef`'ed variable, we could not proceed with fuzzing. Listing 1 presents a sample of a test harness for a `KBUS` send message function.

5 Experimental Evaluation

For the experimental evaluation of our project we targeted a WAGO PFC-100 PLC featuring a TI AM335x chipset with a Cortex-A8 ARM processor at 600MHz and 256Mb RAM as the Device Under Test (DUT), and a laptop with an i7 processor, 16GB RAM and 512 GB SSD as our main computer. The computer is connected through SSH to the DUT over a local network. Symbolic execution/instrumentation as well as the runtime function test harness building are performed on the computer, since we do not need device specific hardware/-software for these steps and the superior performance offered by a laptop setup speeds up the process. Control application and runtime function fuzzing are handled locally on the DUT. We have collected control binaries from github repositories as well as the Codesys project website, for a total of 184 binaries. In addition, we have developed an assortment of binaries, utilizing various functions and utilities, so we could thoroughly test our fuzzing engine and observe its bug-finding capabilities in a controlled manner[4]. We have also performed regular fuzzing to the control application-related functions we discussed in the fuzzing section. For these functions, we deployed mature fuzzing tools, namely the American Fuzzy Lop (`AFL`) [61], which offers both instrumentation and sanitization capabilities, with select input seeds for each function. We performed four distinct evaluations, testing correctness, fuzzing control applications with code coverage feedback and fuzzing runtime functions.

Correctness: For the correctness evaluation of our control application fuzzing engine, we focused on input control and output observation. We chose in-house developed binaries for this section, since we need to be aware of their functionality to predict the correct output and cross-validate with the observed value. We utilize the `E-Cockpit` development suite for its convenient graphical representation of the control process, which includes live tracking of process output values. The process we have targeted is a part of the Multi Stage Flash (MSF) Desalination process [2] handled by the target PLC. The MSF Desalination model is an academically developed testbed for desalination plant research which has been used as a target process in recent publications. The process input is brine temperature, the output controls brine density involving two Proportional Integral Derivative (PID) functions. We modified input values directly from the process memory space and observed the altered outcome through the `E-Cockpit`. We forced the same input value throughout the experiment, a value which we collected from the real control input transmitted through the I/O module, which was a temperature of 360°C. Fig. 5 depicts the output brine density fluctuation through time before and during the fuzzing process. It is evident that the forced inputs have a direct impact on the process which validates the input control of our fuzzing scheme. It is

[4]The database of house-developed and online-collected control applications will be available online.

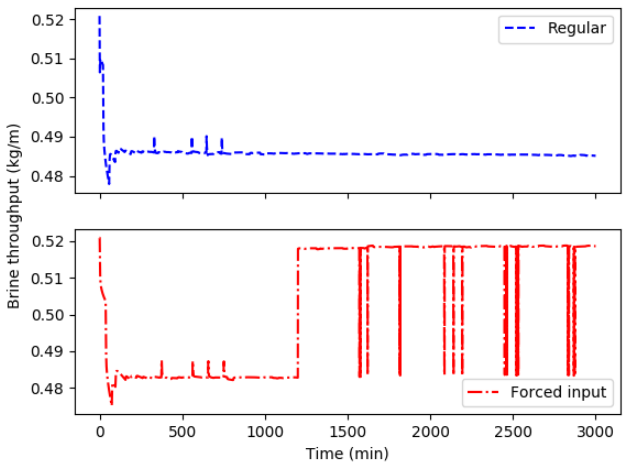


Figure 5: Output observation of forced input on industrial process.

also clear that the timing of our force input methodology is not perfectly accurate. An approximate 6% of the reported outputs correlated with the original input, meaning that for these cases our forced input was not overwritten by the real input from the I/O module. From a performance standpoint, this translates to a 6% decrease in maximum inputs processed during a time span. The original inputs could be considered as introducing unexpected behavior to our engine, in case they cause a program crash while the fuzzer reporting its own forced input as the cause. However, these original inputs are used as seeds to our fuzzer, leading to redundant rather than faulty results. Additionally, each crash-causing input is considered and validated in a case-by-case basis, eliminating mistakes in crash-causing input recognition.

Fuzzing Control Applications: For this part of the experimental evaluation, we have created a collection of potentially vulnerable control binaries. We have considered three distinct scenarios for the introduced vulnerability, with regards to the popularity of their existence in vulnerability assessment practice. These are 1) buffer overflow, 2) out of bounds writes, and 3) divide by zero. Buffer overflow binaries feature memory manipulating instructions, such as `memcpy`, which lack bounds check and the amount of data is a program variable. Out-of-bounds write binaries involve an instantiated array with a variable index, while divide-by-zero binaries have a division operation with an input influenced denominator. The vulnerable binaries have a similar composition:

- The instantiation of the vulnerable section. This can be one of the vulnerable functions, an array assignment, or a division operation.
- One or multiple function calls are included in the main body of the application that may or may not include the vulnerable part. We include function calls to determine whether return addresses can be overwritten.
- Various conditional expressions or loops were introduced to increase the execution complexity of the synthetic binary. This has been done mostly to evaluate our instrumentation-based feedback methodology. Depending on the binary, the vulnerable part is included under a condition making it harder for our fuzzing engine to uncover the vulnerability.

Fuzzing session duration was set to 1 hour, a time period which was proven enough to witness at least one crash instance in our preliminary experiments. Therefore, we followed a black-box approach, mutating an initial zero input vector. Table 4 illustrates the results for this part of the experiment. The control application names correlate to their complexity, i.e. the larger the number, the more decision based flows exist in the program. The naming convention follows the type of vulnerability the test was targeting and the vulnerable function that was instantiated in the program, e.g. `bf_mcpy_1` is the simplest program instantiating a `memcpy` function built to uncover potential buffer overflows and `oob_2_arr_10` is the 10th program instantiating two arrays targeting out-of-bounds access on both. The results in Table 4 are justifying our initial hypothesis that, much like regular binaries, control applications can include binary-type vulnerabilities due to programming errors, lack of compiler security improvements, and vulnerable utility functions. We can also distinguish a lack of correlation between number of inputs processed, which matches to the number of executions, and the execution time until a first crash occurred. Time to execution ratio is clearly dependent to the scan cycle of each control application. The results also validate our fuzzing scheme in terms of performance, forcing multiple crashes in a concise time period, proving to be fast enough to be considered an effective assessment method.

An interesting case category was the divide by zero program which never lead to a crash, no matter the composition of the binary itself. From further research we found out that hardware division is not supported by our platform, with the assembly code produced by the compilation tool replacing regular `idiv` instructions with calls to division functions precompiled for ARMv7, included in a standardized library. Thus, any potential divide by zero error results in a zero output.

The control applications used in this experimental evaluation were all developed in-house with the sole purpose of having a collection of applications that were vulnerable from a typical software application standpoint and we were confident in uncovering the underlying vulnerabilities through fuzzing. During our research, we have also collected and consolidated a list of 187 control applications found in open repositories on github and the official project database hosted by Codesys. We have put these applications through our fuzzing setup and we observed that most of them did not produce any crash or even get in a hung state, regardless of time under test, since many of them are very simple programs. The much more complex desalination process, however, produced crashes that we further analyze in order to demonstrate the usefulness of our approach. The analysis highlighted an out-of-bounds write

Control Application	Execution Speed (inputs/sec)	First Crash (time mm:ss)	First crash (inputs)	Crashes (1hr)
bf_mcpy_1	70.88	3:54	15270	32
bf_mcpy_6	64.2	3:08	12172	21
bf_mcpy_8	66.06	4:39	18216	17
bf_mcpy_12	62.11	7:06	26645	9
bf_mset_1	64.56	3:28	13441	21
bf_mset_3	62.68	2:54	10906	24
bf_mset_5	68.8	4:14	17554	16
bf_mset_9	69.76	10:23	43530	7
bf_mmove_1	64.63	2:56	11245	28
bf_mmove_4	63.1	2:39	10070	24
bf_mmove_7	66.31	3:49	15317	15
bf_mmove_12	64.53	13:03	50643	6
oob_1_arr_1	71.86	0:55	3880	39
oob_1_arr_6	77.03	1:43	8085	28
oob_1_arr_9	69.78	1:45	7326	27
oob_1_arr_13	75.2	3:27	27241	19
oob_2_arr_1	73.53	1:57	8558	35
oob_2_arr_5	71.1	2:45	22759	27
oob_2_arr_8	69.8	3:08	13366	22
oob_2_arr_13	70.95	3:12	13401	19
divby0_1	73.68	N/A	N/A	0

Table 4: Fuzzing results for our in-house developed synthetic control applications.

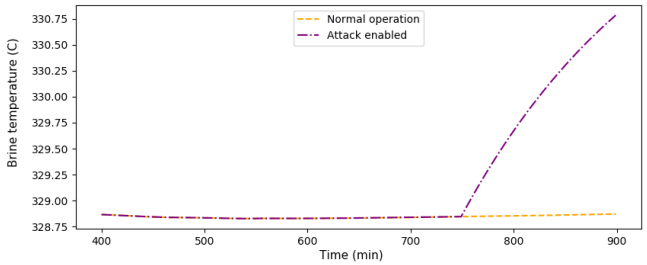


Figure 6: Output observation between normal operation and attack enabled on desalination process.

that corrupted the framework memory resulting in its termination. Given this fact, a DoS-type attack can be a reality, where a spoofed sensor value can directly stop an industrial process such as the desalination plant, resulting in production delay and costly damages as analyzed in [43]. Figure 6 illustrates this attack and its effect on the industrial process. In a desalination plant, an increase of just 1°C in brine temperature causes a substantial decrease in steam flow rate which results in a 12%, or 1 ton/min, decrease in the produced output, translating to a loss of several thousand dollars per [43].

Code Coverage: This part of the experiment was performed to validate the efficiency of our fuzzing scheme through examining the amount of code our engine has explored testing different inputs to the control binary. We have replaced NOPs from distinct function blocks throughout the code with `STR` instructions logging the current program counter (PC) to tag the blocks the program has explored through execution. As discussed earlier, the inclusion of NOPs in control binaries serve as an extra robustness mechanism to protect execution against non-detereministic memory accesses and do not add

[5]Function engaged in the CVE-2017-6025 reported at [16].

Function	Description	Crashes (1hr)
KbusRegisterRequestWrite	Kbus write function	2
KbusRegisterRequestRead	Kbus read function	1
kbus_ksock_write_data	Kbus write function	4
kbus_ksock_read_data	read function	7
XMLParse	XML Parsing Function	8
SysSockRecv[5]	TCP receive data	6
CMAddCoomponentKbus	Kbus instantiation	4
pthread_create	Creates runtime thread	8
pthread_rwlock_unlock	Updates thread privileges	2
pthread_join	Joins PLC task threads	1
pthread_setschedparam	Sets scheduler thread policy	1
GetLoginName	Receives input login name	7
SysLibStrcpy	String copy custom function	2
SysLibStrcmp	String compare custom function	5
SysComWrite	System communication output	7
SysComRead	System communication input	2
GetHookName	Get name of hooked function	6
CopyRtsMetrics	Copies PLC data	8
getspnam	Returns info from shadow file	5

Table 5: Fuzzing results for the runtime functions.

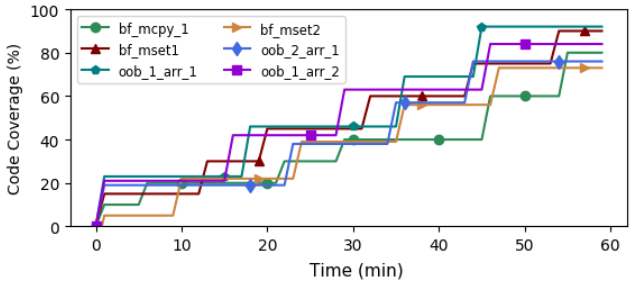


Figure 7: Code coverage results for a subset of our tested control applications.

any hidden/obscure functionality. We sampled the logged PC values every minute for 1-hour fuzzing sessions and present the resulted code coverage percentages in Fig. 7

We have targeted six relatively low complexity control applications as test subjects for this experiment, to have a more accurate approximation of the percentage of code coverage. `oob_1_arr_1` was the simplest one, containing at a small number of execution flows (<10) from which our fuzzing engine managed to discover 90%. `bf_mcpy_1` was a slightly more complex but still relatively simple application, containing less than 20 execution flows, with our fuzzer discovering again approximately 90%. `bf_mset2` yielded the least coverage out of the the tested functions, being the most complex one, with 69% functions traversed.

Fuzzing Control Applications Analysis: Following the fuzzing sessions for our control binaries, we performed a manual analysis to determine the nature of the observed crashes. There were two principal causes for the crash instances, a stack buffer overflow or an out-of-bounds write. In either case, a function instantiated in the PLC binary has its return address overwritten by either an overflow of data or a misaddressed array assignment. Given the analysis at Section

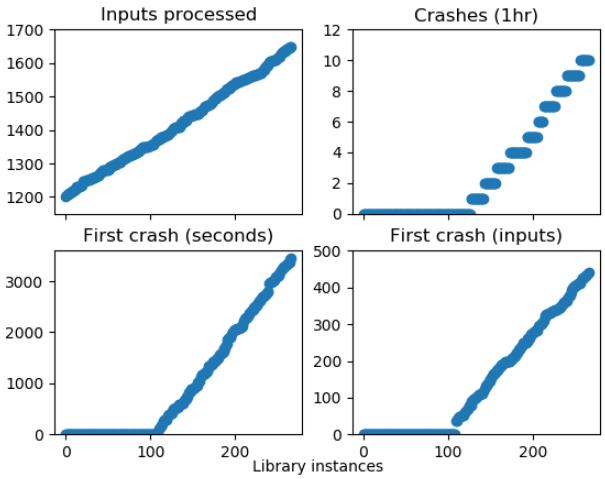


Figure 8: Fuzzing results of runtime functions.

3, we identified a number of functions vulnerable to overflow bugs based on their composition, i.e. lack of checks on bounds or input/buffer sizes. Regarding array value assignments, we observed the inability of the compiler to flag a restricted memory write/read when the array index is influenced by an input variable, such as a sensor value. Thus, in both cases, the possibility to overwrite the return address of a called function validates the potential for execution flow altering attacks. For the next step, we considered the stack hosting the called vulnerable function. We reverse engineered the control application loading process observing its placement in the memory space of the Codesys framework. The code and data sections of the control application are pushed into a stack of a temporary thread and execution of the application itself happens locally in this stack. This fact led to the assumption, and following validation, of the executable status of all stacks instantiated for each running thread, including the control application handling thread. While the Codesys framework binary has the no-execute bit enabled, code can still be executed in all instantiated stacks. Thus, with the combination of overwriting a return address and confirmed arbitrary execution in the process stack, we can formulate an attack vector that can lead to system compromise. We built a control binary with one of the vulnerabilities highlighted earlier, where we also statically included shellcode consisting of a NOP sled followed by assembly code that executes a simple Linux system call. Following the loading and crash of the binary, we were able to distinguish the rogue system call on the kernel log. An arbitrary system call through binary exploitation can enable much more complex payloads, such as rootkit insertion. A rootkit can be easily placed on the PLC with simple user-level access. Then a simple `insmod` with the rootkit name as an argument can be used as a payload in the vulnerable control application. Since the application, as part of the Codesys runtime, runs on root-level privilege, the `insmod` can be executed and the rootkit covertly inserted.

Fuzzing Runtime Functions: Our framework can successfully fuzz non-executable control binaries compiled from high-level PLC programming languages. To have a complete understanding of PLC security, we also need to investigate potential vulnerabilities in the runtime environment. Fuzzing functions in the Codesys runtime was performed on the DUT with seed inputs based on the input argument types of each function and an approximate understanding of the argument purpose, e.g. a size variable, a text message or an address. For more complex inputs, such as structs, we manually initialized the struct variable values and used the mutated inputs alternating between the struct variables, concatenating or repeating based on variable type. We cross-compiled `AFL` for the ARM architecture and utilized the included `afl-gcc`, an enhancement of the latest `gcc` compiler with added instrumentation capabilities, for compiling the test harnesses. Function selection was an important part of this evaluation, since not all available runtime function can be targeted for fuzzing or are not influenced by an external input. As mentioned in Section 4, the principal way of fuzzing the runtime functions is building a test harness which instantiates the function itself. In addition, we chose functions whose execution flow can be controlled by a given input and thus are prime targets for exploitation. Functions such as the thread scheduler are independent of external inputs and thus there is no user-based means for exploitation. We performed fuzzing on all chosen functions based on our selection process. The results for this part of the experiment are illustrated in Fig. 8 and a subset is listed in Table 5. Out of over 250 tested functions we isolated 133 which produced crash instances. The number of crashes found was predictably small, due to the relatively low execution count, but enough to validate our initial assumption of vulnerabilities in the functions interacting with the control application. Through observation of the first crash instance for the affected functions, we can also assert the efficiency of this approach in discovering vulnerabilities in these utility functions: An average of 14 minutes was enough to crash the target function at least once.

Out of 850 total crash instances examined, we have a better understanding of the cause of the crash for 97 of them, all related to buffer or heap overflows. These functions were instantiated in the test harness with related preceding instructions and structures, adding conditions and loops as well as initializing-related variables we observed in the decompiled instances of the runtime we received through `ghidra`.

While providing with convenient abstraction, decompilation is still not fully accurate in reproducing the original code. Thus, while our results uncover potential vulnerabilities in certain parts of the runtime, the original source code could have taken measures against them.

Summarizing, fuzzing the runtime produced concerning results indicating that immediate action is needed on improving the security posture of these pieces of software.

6 Discussion and Related Work

In recent years ICS security has gained a lot of attention. Most of the research focused on securing the system at the network level though intrusion detection systems (IDSs) [56] or anomaly detection [20] while, control applications have not received the same attention. Traditionally, PLC control applications/software have been researched as a means of deploying malicious payloads to the device itself to compromise the attached industrial process, a prominent example being Stuxnet. As such, a control application has either been considered as malware in its entirety or as a malware infected legitimate application, with most efforts focusing on detecting and uncovering the malicious payload [44, 45]. Fuzzing has been a popular topic in embedded device security analysis as well, targeting devices such as smart meters [3], smartphones [58], automotive [27], and a variety of other devices [32]. An interesting work introduces input-generation fuzzing to evaluate Robotic Vehicles (RV) control code and uncover cases that would lead to incorrect control decisions [29]

In practice and in research PLC control software has not been subject to security assessments with very few publications related to the presented work. Instead, many prior efforts target the field of safety verification of PLC control applications/software. In both [7] and [25] the authors perform language specific PLC control application verification. Other efforts focus on detecting potential corruption of PLC control applications through runtime monitoring and verification [18, 26]. VETPLC [64] aims to verify real-world PLC code, by taking into account the sequence of events and their time constraints. Many of these solutions mainly focus on the detection of safety violations.

SymPLC [23] leverages the OpenPLC [5] framework and Cloud9 engine [9] to conduct dynamic analysis on control applications. Symbolic execution was used to evaluate control application binary code considering the application as a piece of software and not just a control process facilitator. The introduced SYMPLC framework abstracts the control application byte code to a high-level C-based representation. Deploying mature symbolic analysis tools the authors succeeded in achieving high function coverage for tested binaries.

The authors in [28] tackled reverse engineering Codesys derived control applications combining manual exploration of the target binary and automated analysis through symbolic execution. The authors succeeded in producing a control flow graph covering every function, statically or dynamically called covering the entirety of the application. Leveraging this information, they showcased an automated on-the-fly attack formulation based on a regular control application.

Emulation is a popular methodology for assisting fuzzing as showcased in [35, 65]. In our platform, however, the implementation of emulation is not as straightforward as in typical Linux-based systems [55]. Either partial emulation, full binary emulation, or full system emulation would be challenging to incorporate to our framework:

- Partial emulation, e.g. I/O module emulation, which can essentially replace our input forcing method can be an alternative, but it is not a simple endeavor. The I/O modules are proprietary peripherals with no software specifications given to the public. A simplistic emulation of an I/O module, based on its known functionality, could be viable but it will fail the I/O check done by the Codesys framework for valid connected I/O peripherals.
- Full emulation of the binary itself is an extremely challenging task, given its unique loading process. Outside the context of the framework, the PLC binary is just a collection of assembly instructions packed in a file. Line-by-line execution is an option, but it would fail at the first instance of input delivery requiring a system call which is routed and handled by the framework.
- Full system emulation would provide full control over the emulated instance and the ability to manipulate conditions, such as the aforementioned I/O check. A full system emulation with the Codesys framework, however, is a very challenging project on its own. While there are broad firmware emulation frameworks published recently, they lack however nuances native to Industrial Control Systems, e.g. the handling of non-generic peripheral such as the I/O modules, or a real-time scheduler.

7 Conclusion

In conclusion, we summarize the answers to the questions that appear in the problem statement section 2.1:

- PLC binaries are inherently robust due to the high-level nature of the PLC programming languages as well as the very well-defined problems they are addressing. As the binaries get more complex in terms of size or function, however, exploitable vulnerabilities that can compromise the host system or industrial process can be introduced.
- PLC runtimes suffer from the same problems that plague regular C/C++ developed software, and this compromises the whole industrial control system computation stack.
- Fuzzing is a great tool for uncovering vulnerabilities in industrial control systems, even in the presence of heavy I/O and scan cycles.

Acknowledgments

This project was supported partly by the U.S. Office of Naval Research under Award N00014-15-1-2182, and by the NYU Abu Dhabi Global PhD Fellowship program.

Resources

ICSFuzz will be available at the following github repository: https://github.com/momalab/icsfuzz.

References

[1] Adam Pilkey. F-secure's guide to evil maid attacks. https://blog.f-secure.com/f-secures-guide-to-evil-maid-attacks/, 2018.

[2] Hala Faisal Al-Fulaij. *Dynamic modeling of multi stage flash (MSF) desalination plant*. PhD thesis, UCL (University College London), 2011.

[3] Vincent Alimi, Sylvain Vernois, and Christophe Rosenberger. Analysis of embedded applications by evolutionary fuzzing. In *2014 International Conference on High Performance Computing & Simulation (HPCS)*, pages 551–557. IEEE, 2014.

[4] Magnus Almgren, Davide Balzarotti, Jan Stijohann, and Emmanuele Zambon. D5. 3 report on automated vulnerability discovery techniques.

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

[6] Neelesh Bhattacharya, Abdelilah Sakti, Giuliano Antoniol, Yann-Gaël Guéhéneuc, and Gilles Pesant. Divide-by-zero exception raising via branch coverage. In *International Symposium on Search Based Software Engineering*, pages 204–218. Springer, 2011.

[7] G. Canet, S. Couffin, J. . Lesage, A. Petit, and P. Schnoebelen. Towards the automatic verification of plc programs written in instruction list. In *International Conference on Systems, Man and Cybernetics. IEEE, 2000*, volume 4, pages 2449–2454 vol.4, Oct 2000.

[8] Defense Use Case. Analysis of the cyber attack on the ukrainian power grid. *Electricity Information Sharing and Analysis Center (E-ISAC)*, 388, 2016.

[9] Liviu Ciortea, Cristian Zamfir, Stefan Bucur, Vitaly Chipounov, and George Candea. Cloud9: a software testing service. *Operating Systems Review*, 43:5–10, 2009.

[10] CODESYS. Codesys control v3 manual, 2019.

[11] CODESYS. CODESYS Device Directory. https://devices.codesys.com/device-directory/, 2019. [Online ; Accessed January 2020].

[12] Crispin Cowan, F Wagle, Calton Pu, Steve Beattie, and Jonathan Walpole. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00*, volume 2, pages 119–129. IEEE, 2000.

[13] Weidong Cui, Marcus Peinado, Karl Chen, Helen J Wang, and Luis Irun-Briz. Tupni: Automatic reverse engineering of input formats. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 391–402. ACM, 2008.

[14] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Vinuela, and Víctor M González Suárez. Formal verification of complex properties on plc programs. In *International Conference on Formal Techniques for Distributed Objects, Components, and Systems*, pages 284–299. Springer, 2014.

[15] Dániel Darvas, István Majzik, and Enrique Blanco Viñuela. Formal verification of safety plc based control software. In *International Conference on Integrated Formal Methods*, pages 508–522. Springer, 2016.

[16] CVE Details. Codesys Runtime System Security Vulnerabilities. https://www.cvedetails.com/vulnerability-list/vendor_id-12574/product_id-23853/version_id-164054/3s-software-Codesys-Runtime-System--.html, 2018. [Online ; Accessed January 2020].

[17] Alexey G Finogeev and Anton A Finogeev. Information attacks and security in wireless sensor networks of industrial scada systems. *Journal of Industrial Information Integration*, 5:6–16, 2017.

[18] L. Garcia, S. Zonouz, Dong Wei, and L. P. de Aguiar. Detecting plc control corruption via on-device runtime verification. In *2016 Resilience Week (RWS)*, pages 67–72, Aug 2016.

[19] Luis Garcia, Ferdinand Brasser, Mehmet Hazar Cintuglu, Ahmad-Reza Sadeghi, Osama A Mohammed, and Saman A Zonouz. Hey, my malware knows physics! attacking plcs with physical model aware rootkit. In *NDSS*, 2017.

[20] Pedro García-Teodoro, Jesús E. Díaz-Verdejo, Gabriel Maciá-Fernández, and Enrique Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28:18–28, 2009.

[21] David Greenfield. Why is Linux Trending? https://www.automationworld.com/why-linux-trending, 2018. [Online ; Accessed January 2020].

[22] The Guardian. Robot kills worker at Volkswagen plant in Germany. https://www.theguardian.com/world/2015/jul/02/robot-kills-worker-at-volkswagen-plant-in-germany, 2015. [Online; accessed July 2019].

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

[24] Peter Huitsing, Rodrigo Chandia, Mauricio Papa, and Sujeet Shenoi. Attack taxonomies for the modbus protocols. *International Journal of Critical Infrastructure Protection*, 1:37–44, 2008.

[25] Ralf Huuck. Semantics and analysis of instruction list programs. *Electron. Notes Theor. Comput. Sci.*, 115(C):3–18, January 2005.

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

[27] Markus Kammerstetter, Christian Platzer, and Wolfgang Kastner. Prospect: peripheral proxying supported embedded code testing. In *Proceedings of the 9th ACM symposium on Information, computer and communications security*, pages 329–340. ACM, 2014.

[28] Anastasis Keliris and Michail Maniatakos. ICSREF: A framework for automated reverse engineering of industrial control systems binaries. In *NDSS*, 2019.

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

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

[31] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, 2014.

[32] Hyeryun Lee, Kyunghee Choi, Kihyun Chung, Jaein Kim, and Kangbin Yim. Fuzzing can packets into automobiles. In *2015 IEEE 29th International Conference on Advanced Information Networking and Applications*, pages 817–821. IEEE, 2015.

[33] Gaoqi Liang, Steven R Weller, Junhua Zhao, Fengji Luo, and Zhao Yang Dong. The 2015 ukraine blackout: Implications for false data injection attacks. *IEEE Transactions on Power Systems*, 32(4):3317–3318, 2017.

[34] Hal Lonas. Introduction to GCC Compiler Induced Vulnerability. https://www.openwall.com/lists/oss-security/2018/10/22/3, 2018. [Online ; Accessed January 2020].

[35] Dominik Maier, Benedikt Radtke, and Bastian Harren. Unicorefuzz: On the viability of emulation for kernelspace fuzzing. In *13th {USENIX} Workshop on Offensive Technologies ({WOOT} 19)*, 2019.

[36] Michaël Marcozzi, Qiyi Tang, Alastair Donaldson, and Cristian Cadar. A systematic impact study for fuzzer-found compiler bugs. *arXiv preprint arXiv:1902.09334*, 2019.

[37] John Matherly. Complete guide to shodan. *Shodan, LLC (2016-02-25)*, 2015.

[38] Stephen E McLaughlin, Saman A Zonouz, Devin J Pohly, and Patrick D McDaniel. A trusted safety verifier for process controller code. In *NDSS*, volume 14, 2014.

[39] National Security Agency. A software reverse engineering (sre) suite of tools developed by nsa's research directorate in support of the cybersecurity mission. https://ghidra-sre.org/, 2019.

[40] SecureWorld News Team. Industrial Control Systems: Suffer a Breach and Lose Big. https://www.secureworldexpo.com/industry-news/industrial-control-systems-suffer-a-breach-and-lose-big, 2017. [Online ; Accessed January 2020].

[41] Tolga Ovatman, Atakan Aral, Davut Polat, and Ali Osman Ünver. An overview of model checking practices on verification of plc software. *Software & Systems Modeling*, 15(4):937–960, 2016.

[42] Nicole Perlroth and Clifford Krauss. A Cyberattack in Saudi Arabia Had a Deadly Goal. Experts Fear Another Try. https://www.nytimes.com/2018/03/15/technology/saudi-arabia-hacks-cyberattacks.html, 2018. [Online ; Accessed January 2020].

[43] Prashant Hari Narayan Rajput, Pankaj Rajput, Marios Sazos, and Michail Maniatakos. Process-aware cyberattacks for thermal desalination plants. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security*, pages 441–452, 2019.

[44] Julian L Rrushi. Timing performance profiling of substation control code for ied malware detection. In *Proceedings of the 3rd Annual Industrial Control System Security Workshop*, pages 15–23. ACM, 2017.

[45] Abraham Serhane, Mohamad Raad, Raad Raad, and Willy Susilo. Plc code-level vulnerabilities. In *2018 International Conference on Computer and Applications (ICCA)*, pages 348–352. IEEE, 2018.

[46] Yan Shoshitaishvili, Ruoyu Wang, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. Firmalice-automatic detection of authentication bypass vulnerabilities in binary firmware. In *NDSS*, 2015.

[47] Ryan Singel. Industrial Control Systems Killed Once and Will Again, Experts Warn. https://www.wired.com/2008/04/industrial-cont/, 2008. [Online; accessed July 2019].

[48] Slimm609. Checkseck. https://github.com/slimm609/checksec.sh, 2011.

[49] Rob Sobers. 60 Must-Know Cybersecurity Statistics for 2019. https://www.varonis.com/blog/cybersecurity-statistics/, 2019. [Online ; Accessed January 2020].

[50] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. Plc-blaster: A worm living solely in the plc. *Black Hat Asia*, 16, 2016.

[51] Michael Tiegelkamp and Karl-Heinz John. *IEC 61131-3: Programming industrial automation systems*. Springer, 1995.

[52] Dimitrios Tychalas, Anastasis Keliris, and Michail Maniatakos. LED Alert: Supply Chain Threats for Stealthy Data Exfiltration in Industrial Control Systems. In *2019 IEEE 25th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 194–199. IEEE, 2019.

[53] Dimitrios Tychalas, Anastasis Keliris, and Michail Maniatakos. Stealthy information leakage through peripheral exploitation in modern embedded systems. *IEEE Transactions on Device and Materials Reliability*, 20(2):308–318, 2020.

[54] Dimitrios Tychalas and Michail Maniatakos. Open platform systems under scrutiny: A cybersecurity analysis of the device tree. In *2018 25th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pages 477–480. IEEE, 2018.

[55] Dimitrios Tychalas and Michail Maniatakos. IFFSET: In-Field Fuzzing of Industrial Control Systems using System Emulation. In *2020 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 662–665. IEEE, 2020.

[56] David I. Urbina, Jairo A. Giraldo, Alvaro A. Cardenas, Nils Ole Tippenhauer, Junia Valente, Mustafa Faisal, Justin Ruths, Richard Candell, and Henrik Sandberg. Limiting the impact of stealthy attacks on industrial control systems. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS '16, pages 1092–1105, New York, NY, USA, 2016. ACM.

[57] David I Urbina, Jairo Alonso Giraldo, Nils Ole Tippenhauer, and Alvaro A Cárdenas. Attacking fieldbus communications in ics: Applications to the swat testbed. In *SG-CRC*, pages 75–89, 2016.

[58] Fabian Van Den Broek, Brinio Hond, and Arturo Cedillo Torres. Security testing of gsm implementations. In *International Symposium on Engineering Secure Software and Systems*, pages 179–195. Springer, 2014.

[59] Suan Hsi Yong and Susan Horwitz. Protecting c programs from attacks via invalid pointer dereferences. In *ACM SIGSOFT Software Engineering Notes*, volume 28, pages 307–316. ACM, 2003.

[60] Jonas Zaddach and Andrei Costin. Embedded devices security and firmware reverse engineering. *Black-Hat USA*, 2013.

[61] Michal Zalewski. American fuzzy lop. http://lcamtuf.coredump.cx/afl. [Online; Accessed January 2020].

[62] Kim Zetter. The Ukrainian Power Grid Was Hacked Again. https://motherboard.vice.com/en_us/article/bmvkn4/ukrainian-power-station-hacking-december-2016-report, 2017. [Online; Accessed January 2020].

[63] Li Zhang and Vrizlynn LL Thing. A hybrid symbolic execution assisted fuzzing method. In *Region Ten Conference*, pages 822–825. IEEE, 2017.

[64] Mu Zhang, Chien-Ying Chen, Bin-Chou Kao, Yassine Qamsane, Yuru Shao, Yikai Lin, Elaine Shi, Sibin Mohan, Kira Barton, James R. Moyne, and Z. Morley Mao. Towards automated safety vetting of plc code in real-world plants. In *S&P 2019*. IEEE, 2019.

[65] Yaowen Zheng, Ali Davanian, Heng Yin, Chengyu Song, Hongsong Zhu, and Limin Sun. Firm-afl: high-throughput greybox fuzzing of iot firmware via augmented process emulation. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*, pages 1099–1114, 2019.