ICSPatch: Automated Vulnerability Localization and Non-Intrusive Hotpatching in Industrial Control Systems using Data Dependence Graphs

Prashant Hari Narayan Rajput, *NYU Tandon School of Engineering;*
Constantine Doumanidis and Michail Maniatakos, *New York University Abu Dhabi*

https://www.usenix.org/conference/usenixsecurity23/presentation/rajput

ICSPatch: Automated Vulnerability Localization and Non-Intrusive Hotpatching in Industrial Control Systems using Data Dependence Graphs

Prashant Hari Narayan Rajput[1], Constantine Doumanidis[2], Michail Maniatakos[2]

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

Abstract

The paradigm shift of enabling extensive intercommunication between the Operational Technology (OT) and Information Technology (IT) devices allows vulnerabilities typical to the IT world to propagate to the OT side. Therefore, the security layer offered in the past by air gapping is removed, making security patching for OT devices a hard requirement. Conventional patching involves a device reboot to load the patched code in the main memory, which does not apply to OT devices controlling critical processes due to downtime, necessitating in-memory vulnerability patching. Furthermore, these control binaries are often compiled by in-house proprietary compilers, further hindering the patching process and placing reliance on OT vendors for rapid vulnerability discovery and patch development. The current state-of-the-art hotpatching approaches only focus on firmware and/or RTOS. Therefore, in this work, we develop ICSPatch, a framework to automate control logic vulnerability localization using Data Dependence Graphs (DDGs). With the help of DDGs, ICSPatch pinpoints the vulnerability in the control application. As an independent second step, ICSPatch can non-intrusively hotpatch vulnerabilities in the control application directly in the main memory of Programmable Logic Controllers while maintaining reliable continuous operation. To evaluate our framework, we test ICSPatch on a synthetic dataset of 24 vulnerable control application binaries from diverse critical infrastructure sectors. Results show that ICSPatch could successfully localize all vulnerabilities and generate patches accordingly. Furthermore, the patch added negligible latency increase in the execution cycle while maintaining correctness and protection against the vulnerability.

1 Introduction

Critical infrastructure such as water treatment plants, the electric grid, chemical manufacturing, and many more rely on various control systems/components broadly identified as Industrial Control Systems (ICS) for regulating physical processes based on industrial logic, necessary for reliable and uninterrupted operation of deeply interconnected critical infrastructure [52]. Traditionally, these ICS devices executed only the industrial logic, remained confined to the industrial network, and often used proprietary software. However, with the advent of Industry 4.0, the industrial internet of things (IIoT) has gained popularity due to its real-time visibility into the industrial process for robust, affordable asset monitoring and rapid diagnostics while requiring minimal human interaction for data exchange [5]. There is also a trend of shifting away from proprietary software and embracing open source since vendors do not need to reinvent the wheel and develop SSH or web server implementations from scratch. However, these advances in industrial computation also transfer vulnerabilities and attack vectors from the Information Technology (IT) sector to the industrial domain.

As a result, attacks on critical infrastructure have become commonplace in the past few years. These attacks often exploit a vulnerability in the IT infrastructure to reach the Operational Technology (OT) control systems, leading to devastating consequences. One such prominent attack is Stuxnet, which altered the code of the industrial logic running on Siemens PLCs [15]. Some other examples include a self-replicating virus on the computer network of Saudi Aramco, which shut down 5.7 million barrels of output per day - more than 5% of the global oil supply [40]. Another widely known instance is the 2015 Ukraine blackout, which involved an attack on the power grid infrastructure that left approximately 225,000 customers without electricity for several hours [27]. Alternatively, a much more sophisticated attack in December 2016, dubbed Industroyer, with support for four different industrial communication protocols and denial-of-service (DoS) attack implementation for Siemens SIPROTEC family of protection relays, led to another power outage [8]. As a result, organizations have increased their budgets to reduce the risk of cyberattacks. In an investigation carried out by SANS on 340 professionals, 42% of the respondents confirmed an increase in the cybersecurity budget for their organization [16].

A typical ICS setup consists of several control loops utilizing sensors, actuators, and a controller to regulate physi-

cal processes. The control logic developed for managing the industrial process executes inside a runtime (privileged software to manage proper application execution, handling I/O, debugging, and more) and is compiled from IEC 61131-3 programming languages. Recently, vendors producing PLCs have started embracing embedded Linux with a real-time patch [20, 23, 48, 57]. Choosing Linux as the OS in PLCs enables the utilization of open-source libraries such as OpenSSL, Apache HTTP server, and jQuery, to name a few. A prime example of this trend is the Codesys platform, currently utilized by at least 80 industrial device vendors and supports more than 400 devices from a wide variety of manufacturers [20]. Four top PLC vendors, namely Mitsubishi, Schneider Electric, Bosch Rexroth, and ABB, hold ≈28% of the global PLC market share [51] and integrate Codesys into their products. Furthermore, the Codesys ecosystem is growing, with high-profile vendors adopting it in their newer product lines to reduce maintenance efforts for their increasingly complex ecosystems [3, 44], and minor vendors are using it as a turnkey solution to lower the barrier of entry into the market.

Most of the research in the domain focuses on obtaining guarantees about the functionality of the PLC and discovering vulnerabilities: Formal verification of the PLC software [10, 11, 39], security evaluations [17, 33, 50], reverse engineering [25], and fuzzing [55]. These vulnerabilities can result from coding errors (incorrect array indexing) or inherent issues in system libraries (missing bound checks), leading to exploitable vulnerabilities in the control application. Such vulnerabilities can impact not only the control binary itself but also the runtime (i.e., its supervisory software), causing undesired process performance or a complete DoS attack requiring a reboot.

Similar to regular system updates, preventing the exploitation of control application binaries due to vulnerabilities in their code requires patching. However, it remains unexplored in the ICS domain, which has unique constraints. For instance, PLCs regulate critical physical processes and do not allow for any downtime or intrusive modifications to the control application that might affect the continued operation of critical infrastructure, making hotpatching essential. Moreover, control application binaries are non-executable binaries compiled by a proprietary compiler and run on a custom runtime, such as the Codesys runtime system presented earlier. Such nuances prevent the applicability of conventional patching solutions and require a more tailored approach to automated vulnerability localization and hotpatching without interrupting the execution of control logic. In literature, patching ICS devices has predominantly focused only on the firmware, as evident in HERA, which utilizes hardware-based debugging features in Cortex-M microcontrollers to patch vulnerabilities in FreeRTOS [37], and RapidPatch, which employs eBPF virtual machines to execute standard patches released by RTOS developers on resource-constrained devices [21]. On the other hand, further related work on this topic overlooks

the patching element and focuses instead on ensuring continuity of operation, employing hot standby redundant PLCs for various purposes such as resiliency against cyberattacks, validating sensor data, process continuity, and improving reliability [24, 30, 31, 60]. Nevertheless, automated vulnerability hotpatching is a trending topic in other domains like in Android for performing multi-level adaptive patching [7, 58], and in open-source software [13, 54], but it remains unexplored in the ICS domain.

This paper proposes ICSPatch, a novel approach for automated vulnerability localization and non-intrusive in-memory hotpatching. The methodology starts with capturing an execution snapshot of the development/test PLC device, rehosting it in an angr instance running on a host for discovering and localizing vulnerabilities using data dependency graphs. Then, we utilize this information to calculate relevant offsets, create a patch for the vulnerability, and non-intrusively patch them in the main memory of the deployed PLC executing the control application. To our knowledge, this work is the first attempt at automated vulnerability localization and hotpatching of non-executable *ICS control binaries*. In summary, our contributions are the following:

- We develop a methodology for automated vulnerability localization for non-executable control application binaries using custom-built data dependence graphs. Our method does not need access to the source code or binary instrumentation; it only needs a copy of the original unmodified control binary file.

- In parallel, we develop a methodology for control binary hotpatching that can be performed with only remote root access to the device and without any hardware support. Our hotpatching is non-intrusive and decoupled from our automated vulnerability localization. It can be used if there exists no other way of patching the vulnerability.

- We create a diverse synthetic dataset of 24 vulnerable control application binaries from five critical infrastructure sectors using the top entries of the latest Common Weakness Enumeration list.

- We evaluate ICSPatch on: 1) Our synthetic dataset, 2) Other state-of-the-art datasets [55], 3) A hardware-in-the-loop desalination plant model employing a real PLC executing a vulnerable control binary.

2 Preliminaries

We focus on PLCs that employ Linux as the base operating system and runtime for executing control logic. For the current version of ICSPatch, we target the *Codesys* Runtime, a multi-platform execution environment employed for programming controllers in IEC 61131-3 compliant programming languages. It operates predominantly on PLCs running a lightweight version of RT-patched Linux OS and comes packed in a self-contained ELF binary that spawns threads for various components. These components include MainTask,
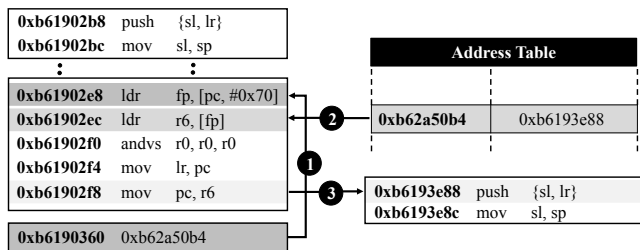
Figure 1: Control application directly modifying the `PC` to branch to the next function in Wago PFC100 PLC.

which handles the execution of the control application binary. The runtime is not a Position Independent Executable (PIE), so its memory regions are mapped to a fixed location. Application binary execution happens in the three stages that comprise the scan cycle: 1) I/O peripheral inputs sampling. 2) Execution of the control logic. 3) Writing outputs to the I/O peripherals. Since `MainTask` executes the control application as a thread, they share process memory space. Thus, changes from the control application can impact the runtime. Codesys compiler compiles the control logic into a proprietary format that is then mapped into memory by the runtime. The runtime also initializes the internal address table that stores absolute addresses of the various control application, runtime, and shared library functions.

Figure 1 shows the steps involved in branching to a called function, 1) The program loads the base address of the address table using PC-relative addressing into `FP` (`R11`), 2) it then loads the absolute address of the next function in `R6`. 3) The `LR` register is overwritten with the return address, and then the `PC` is updated with the value of the `R6` register, currently holding the address of the next function. Traditional approaches exchange parameters between functions using both registers and memory locations. However, function parameters are passed only through memory locations in control application binaries compiled by the Codesys compiler. Codesys runtime also utilizes shared functions provided by standard C libraries (e.g., `Libc`, `Libm`), using a wrapper function to move the parameters from memory to appropriate registers.

Unlike GCC, which employs a Stack Smashing protector, the proprietary Codesys compiler does not implement such protections, enabling malicious writing on the stack. This means that an adversary can control the state of the runtime (the supervisory software for the control application) by exploiting vulnerabilities in the control application itself. To illustrate this, as a motivating example, consider the loader function for the control application, depicted in Figure 2, that passes control to `PLC_PRG` (the equivalent of *main* in C) if the byte extracted from the address `0xb62beb82` is zero. By exploiting vulnerabilities in the control logic, an adversary can manipulate this byte to skip the execution and output phases of the scan cycle, essentially replaying the unprocessed previous written values in the memory-mapped I/O registers.
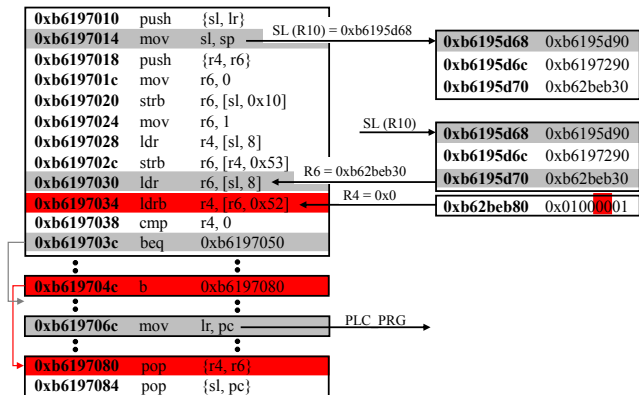
Figure 2: Runtime function executed just before the execution of the entry point of the control application.

3 Problem Formulation

In this work, we explore the security implications to the PLC due to vulnerabilities in control applications and the methodology to locate and patch them. More specifically, we endeavor to solve these research problems:

Automating vulnerability localization in the non-executable ICS binary. ICS devices operate in the field continuously without being taken offline, which only occurs during the scheduled maintenance periods, making hotpatching the only feasible option to patch critical vulnerabilities. This technique patches directly in the main memory during the execution of a program, requiring *precise* knowledge of the location of the vulnerability. The current state-of-the-art patching techniques, such as HERA [37] and RapidPatch [21], assume access to the patched binary or the source code to identify the precise location of the patch, eliminating the localization issue. In our scenario, we assume only access to the binary currently controlling the industrial process, making localization more challenging.

At the same time, our focus is to patch *only* the control application while excluding any of its surrounding logic to avoid unintended system consequences. Vulnerabilities in the control application can manifest from runtime imported functions. For instance, a control application utilizing the `MemSet` function (to initialize memory for a variable) without implementing proper bound checks will crash the runtime while implementing a store (`str`) instruction inside the `MemSet` function. So, the application does not crash the runtime but sends unexpected parameters to the `MemSet` function that leads to the crash. However, these shared runtime functions may also be used by other parts of the code with benign impact. Therefore, we aim to patch the vulnerability *in the control application* rather than the imported functions; thus, localizing the vulnerability in the control binary is critical.

Software-only remote hotpatching of vulnerable control applications. In general, hotpatching involves modifying the

execution flow in the control logic towards the implemented trampoline function by implementing a hook in the original assembly code. Codesys does not provide hotpatching functionality. The development environment (IDE) allows forcing variable value, which writes the variable value in each cycle, permanently holding the variable at the forced value. However, any changes to the code recompiles and redownloads the updated control application binary to the PLC, which is not desirable during continuous operation. Current state-of-the-art solutions such as HERA [37] and RapidPatch [21] patch RTOS compiled like a traditional binary and cannot directly patch non-executable control applications. Furthermore, HERA utilizes Flash Patch and Breakpoint Unit (FPB) on Cortex M3/M4 for dynamically patching embedded devices. Such features limit the applicability of these patching techniques on devices with other microcontrollers and require hardware support. In our scenario, the PLC cannot be decommissioned to add hardware support. Therefore, we aim to apply the patch by remotely connecting to the PLC with admin privileges, a realistic and prevalent way to connect to a PLC in the field.

3.1 Threat Model and Assumptions

We consider a field device, a PLC that regulates a physical industrial process, connected to the industrial control network to enable loading, monitoring, and managing the control logic through a Human Machine Interface (HMI). This PLC also receives sensor inputs processed by the currently executing control logic to produce appropriate outputs relayed to the actuators, allowing the PLC to impact the physical state of the industrial process based on the implemented logic. An intelligent adversary can perform man-in-the-middle (MITM) attacks by intercepting and modifying communication with the critical device in the following ways to deliver malicious inputs to the target PLC: 1) Manipulating the sensor inputs to the PLC by compromising the sensor [56] or by changing the values in the memory-mapped I/O registers [55]. 2) Intercepting and modifying the network packets sent from the HMI to the currently executing control logic, enabling data modification in the program while injecting false data to the HMI [25].

We assume that the adversary is limited to data injection/modification attacks by manipulating the sensors or the network traffic. Because if the adversary can modify the control logic, the patch can be removed/overwritten. Such scenarios require orthogonal control logic modification protection solutions, for instance, using checksums, digital signatures [53], control logic comparison while uploading [46], and formal verification employing behavior [2, 33, 35, 59], state [10, 19, 29], specification verification [4, 22, 62].

Since ICSPatch targets hotpatching of control applications instead of the firmware, we cannot assume the existence of patches from a trusted source (an assumption by HERA [37]

and RapidPatch [21]). Instead, ICSPatch creates its patches by populating simple skeleton patches with concrete values. To identify and locate the vulnerability, ICSPatch assumes the existence of at least one exploit input, a typical assumption made in literature [47]. For our target Codesys platform control binaries, we use ICSFuzz [55] to identify a single exploit input. However, similar to existing approaches in literature [6, 7, 32], ICSPatch cannot guarantee the correctness of the generated patch, which should be decided by the process engineers on a development PLC. However, it ascertains the safety of the generated patch similar to RapidPatch [21], elaborated further in Section 4.3.

4 ICSPatch Methodology

Figure 3 displays the overall workflow involved with patching control application binaries. The process begins with assuming access to an exploit input that can crash the control application executing on the development PLC, an assumption already made in the literature for automated vulnerability localization [47]. Specialized tools such as ICSFuzz [55] enable the fuzzing of control application binaries, providing the exploit input if available. ICSPatch extracts hexdumps of the runtime process memory space, the `MainTask` thread executing the control application, and any other required shared libraries. For example, some of the vulnerable control logic in our experiments use shared functions from `Libc` and `Libm`. We ensure that the exploit input is stored in the memory-mapped I/O registers before extracting the hexdumps. ICSPatch also extracts other information, such as critical addresses, to enable rehosting of the control application.

4.1 Overview

As shown in Algorithm 1, having extracted all the relevant information and memory snapshots from the development PLC, ICSPatch starts by initiating an `angr` simulation (an open-source binary analysis platform combining static and dynamic analysis techniques [1]). The simulation instance is initialized with the memory hexdumps by `INIT_SIM()`, which also sets the `PC` to the start address of the function executed just before the beginning of the PLC control process: "`PLC_PRG`", (also shown in Figure 2). After the end address of the simulation is calculated using `CALC_END_ADDR()`, ICSPatch calls `ENABLE_DETECTION_RULES()` to enable memory violation detection rules based on attack observations. This functionality leverages breakpoint support in the `angr` execution engine. Starting at line 7, we step through the simulation performing *concolic* execution (concrete input combined with symbolic execution) of the loaded control application. As ICSPatch executes the control logic in `angr`, it iteratively builds a custom Data Dependence Graph (DDG) using `ADD_INSTR_NODE()`, `ADD_MEM_NODE()`, and `ADD_EDGE()` to add instruction, memory and transition type nodes, respectively (discussed further
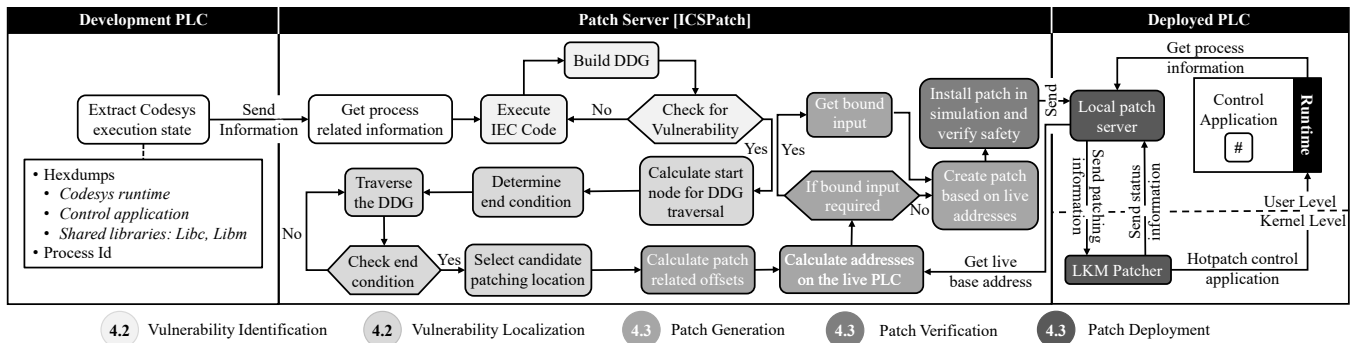
Figure 3: ICSPatch system design.

Algorithm 1 DDG-based vulnerability localization and patch generation.

Input: *hexdumps*
```
 1: // Initialize simulation, get simulation state and instruction program counter
 2: sim, state, pc ← INIT_SIM(hexdumps)
 3: end ← CALC_END_ADDR(start, state)
 4: // Enable vulnerability detection rules in simulation
 5: sim.ENABLE_DETECTION_RULES()
 6: ddg ← INIT_DDG()
 7: while pc ≠ end do
 8:     // Add instruction node for current instruction, containing its operands
 9:     ddg.ADD_INSTR_NODE(pc, state.pc.oprnds)
10:     if state.op = 'mem_write' then
11:         // If sim writes to memory, add mem node and connect it to the instr node
12:         ddg.ADD_MEM_NODE(state.mem_write_addr)
13:         ddg.ADD_EDGE(pc, state.mem_write_addr, 'stores')
14:     else if state.op = 'mem_read' then
15:         // If sim reads from memory, add mem node and connect it to instr node
16:         ddg.ADD_MEM_NODE(state.mem_read_addr)
17:         ddg.ADD_EDGE(state.mem_read_addr, pc, 'loads')
18:     else if state.op = 'reg_write' then
19:         // If write to reg, connect instr node to previous reg state (transition node)
20:         ddg.ADD_EDGE(PRV_REG_STATE(state.pc.oprnd2), pc, 'next')
21:     end if
22:     // Detect vulnerability using memory violation rules
23:     if DETECT_VULNERABILITY(state) then
24:         // Locate DDG traversal starting point
25:         start_addr ← GET_COMPARISON_INSTRUCTION(state.block)
26:         // Get code block bounds for DDG traversal
27:         block_start, block_end ← GET_NEAREST_APP_BLOCK_ADR()
28:         // Traverse DDG using DFS algorithm to get patch address
29:         sim_p_addr ← DFS(ddg, start_addr, block_start, block_end)
30:         // Check if patch address is valid
31:         if CHECK_RANGE(sim_p_addr, block_start, block_end) is false then
32:             FAIL()
33:         end if
34:         b_addr ← GET_BASE_ADDR()
35:         // Create patch based on simulation and deployed PLC information
36:         patch, hook, liv_p_addr ← BUILD_PATCH(state, sim_p_addr, b_addr)
37:         // Deploy patch by sending it to the local patch server on the PLC
38:         DEPLOY_PATCH(patch, hook, liv_p_addr)
39:         EXIT()
40:     end if
41:     state, pc ← sim.SIM_STEP()
42: end while
```

in Section 4.2). Upon detecting a vulnerability in the current control application, ICSPatch performs localization by traversing the DDG. Traversal happens in DFS() starting from the node calculated by GET_COMPARISON_INSTRUCTION() and is bounded by the addresses of the control application function block closest to the vulnerability manifestation, as calculated by GET_NEAREST_APP_BLOCK_AD(). After completing this step, ICSPatch prompts the user to select the appropri-

ate patching candidate if multiple options are available. In our dataset, this occurs only for MemUtils.BitCpy as it can be exploited by manipulating either the size or offset parameters, having two viable starting nodes for vulnerability localization. However, ICSPatch provides the user with the malicious input at the identified memory location, which can be effortlessly verified with the exploit input to choose the correct candidate. Once the patching candidate (memory location) is selected, ICSPatch calls GET_BASE_ADDR() to gather important base address information of the runtime from the deployed PLC. Consequently, it invokes BUILD_PATCH() combining the offsets calculated from the angr instance with the live base address information to build the patch and calculate all appropriate offsets, such as the offset for the memory location of the patch, the hook, and much more. Finally, in DEPLOY_PATCH(), ICSPatch communicates the patching information to the local patch server running on the deployed PLC, which relays it to the Loadable Kernel Module (LKM) patcher, patching the control application.

4.2 Step 1: Vulnerability Identification & Localization

Vulnerability identification. ICSPatch performs vulnerability identification in DETECT_VULNERABILITY() by utilizing the breakpoint functionality in the execution engine of angr. For executing control logic, the runtime passes execution to the control application, whose initial cycle sets up data structures, such as the address table, utilized in consecutive executions. So, in the case of out-of-bounds write and read for our target control applications, the impact of the vulnerability can remain isolated to the control application or even impact the runtime. ICSPatch handles these two cases of impacts differently in ENABLE_DETECTION_RULES(). For malicious reads and writes, ICSPatch reviews all the store and load operations while executing the control application in angr to verify if any of them write below the highest stack address of the control application. On the other hand, malicious out-of-bounds write/read can also have a limited impact on the control application. For detecting such conditions, ICSPatch

Name	Action	Patch Identifier	Definition	Message
		OOB_READ	READ_ADDRESS > RUNTIME_STACK AND READ_ADDRESS < RUNTIME_DATA	
		OS_CMD_INJ	WRITE_ADDRESS > RUNTIME_STACK AND WRITE_ADDRESS > RUNTIME_ADDRESS_TABLE	

Figure 4: Ruleset format for ICSPatch with an example.

executes the control application with legitimate inputs to capture the offset of stack writes for each function inside their corresponding stack frames, later comparing it with the writes of the control application with exploit input. Since we assume that the adversary only has access to the input of the control application, the changes in the memory locations written by the currently executing control logic are only the result of the exploit input. However, the absolute write/read addresses can change; therefore, ICSPatch captures the offset of a memory location written or read per stack frame for each function.

Finally, OS command injection is a unique case of out-of-bounds write involving writing to the address table with the address of the malicious payload, effectively redirecting execution flow toward the payload. ICSPatch monitors store operations on the address table while executing the control application in `angr`. The address table is initialized once (in the first scan cycle) and only changes when loading a new control application or restarting the runtime.

It should be noted that vulnerability identification in ICSPatch only recognizes known categories of vulnerabilities in unknown compiled application binaries because it relies on a core set of rules for detecting these vulnerabilities (defined in `ENABLE_DETECTION_RULES()`). So, expanding the ruleset enables the detection of more vulnerabilities.

Vulnerability ruleset example. Memory operations on the control application stack should remain isolated from the runtime stack so that the runtime can continue operating uncorrupted. Furthermore, considering Figure 8, the control application stack is followed by the runtime stack, control and runtime code, the address table, and the data section. The control application, during its execution, does not need to manipulate any of these memory regions, so a vulnerability ruleset to capture malicious out-of-bounds write requires checking for memory write operations at an address greater than the runtime stack address as shown in Figure 4. Other components of the ICSPatch ruleset include a *name*, a *message* printed upon detecting a particular vulnerability, and the corresponding *action* (`WARN` or `ALERT`). The *patch identifier* specifies the corresponding patch for utilization. Finally, the *rule definition* specifies a logical relationship for vulnerability detection. It can utilize hardcoded values or special addresses dynamically identified by ICSPatch, such as `WRITE/READ_ADDRESS`, `RUNTIME_STACK/DATA`, and `RUNTIME_ADDRESS_TABLE`.

Vulnerability localization. ICSPatch creates a DDG while executing the control application in `angr` as shown in Algorithm 1. In some cases, memory violation vulnerabilities in the control applications result from missing bound checks

I Instruction Node   M Memory Node   T Transition Node

Figure 5: Vulnerability localization in ICSPatch using Data Dependence Graph. The memory node (M), represented in blue, shows the patching memory location of the malicious input just before the execution leaves the control application.

in imported functions rather than the original control logic. ICSPatch detects the exploit location, i.e., the instruction that triggers vulnerability identification. In the out-of-bounds write example shown in Figure 5, the `stmhs` instruction at `0xb6bd78b8` writes into the runtime stack and is therefore detected (represented in red). However, this location is not the same as the patch location. Patching at such locations modifies the shared runtime function, which is also utilizable at other parts of the program. Moreover, the runtime may also call some of the available `Libc` functions instead of implementing its own; patching at this location implies hotpatching `Libc`. So, ICSPatch must identify a patching location close to the boundary between the control application and the runtime, as illustrated in Figure 5.

DDG structure and methodology. Figure 5 shows three main categories of DDG nodes, the instruction I, memory M, and transition nodes T. Control applications utilize memory locations to pass parameters between functions, thus requiring store (`str`) and load (`ldr`) instructions, represented by instruction nodes. Store operations write to a memory location represented by memory nodes in the DDG. We need to include these nodes in our DDG to locate the input in memory that leads to the manifestation of the vulnerability (represented by M). Finally, the control logic might modify the values loaded into the register or move them into a different register. Transition nodes represent such operations on the loaded memory value.

In the example displayed in Figure 5, the `stmhs` instruction at `0xb6bd78b8` leads to the violation of the out-of-bounds write rule. However, such bound checks are implemented by checking the value of the bound, in this case, stored in `R2`, decremented at `0xb6bd789c` with `subs` and verified at

`0xb6bd78bc` with `bhs`. ICSPatch recognizes `0xb6bd789c` as the start node and traverses backward along the path represented by dashed lines reaching the instruction node at `0xb61b3240`. This `str` instruction stores the bound value at the memory location represented by M, which is the value manipulated by an adversary, leading to the manifestation of the vulnerability: Out-of-bounds write in this case. Using the DDG and traversing it with its `DFS()` algorithm, ICSPatch can successfully localize vulnerabilities for control applications and select an appropriate patching location.

Why does ICSPatch utilize DDG? Vulnerability localization can be performed in various ways. For instance, VULNLOC utilizes statistics for identifying correct patching locations for 88% of the considered 43 CVEs while considering Top-5 outputs [47]. On the other hand, the machine learning-based approach DeepVL [26] achieves an accuracy of 96.9% with low precision of 70.1%, while Devign [61] reaches accuracy of 80.24%. ICSPatch captures memory snapshots from the development PLC, allowing simulated execution of the control application with concrete values. At the same time, control binaries typically lack optimization, so data/control flow is easier to understand/extract with precision compared to regular optimized binaries. Dynamic analysis techniques with concrete values of registers explore a single path while providing high fidelity (accurate intermediate states) and produce no false positives (every data dependency is valid) [18,36]. Since ICS devices often control and regulate physical processes in critical infrastructure, a deterministic approach with no false positives is preferable over the aforementioned probabilistic methods with false positives. Therefore, DDGs provide an accurate answer, if a vulnerability exists; if not, ICSPatch fails to patch, which is preferred to the alternative of allowing false positives, i.e., patching at incorrect locations.

4.3 Step 2: Patch Generation & Deployment

Patch generation. In contrast to state-of-the-art embedded systems hotpatching solutions that target the RTOS [21,37], ICSPatch focuses on a much more exotic target: the industrial control application. So, unlike state-of-the-art, it does not have an upstream source for official patches. To automate the process of hotpatching, we restrict our focus to a select category of weaknesses. In `BUILD_PATCH()`, we create skeleton patches for all selected weaknesses and populate the patches with live values from the deployed PLC by communicating with the local patch server running on it. For instance, the out-of-bounds write/read patch compares the value of the bound with the highest permissible value to prevent illegitimate write/read to the stack. ICSPatch populates the skeleton patch with concrete values, which include the higher upper bound for the size and the live memory address of vulnerable input on the deployed PLC (shown by M in Figure 5). On the other hand, the patch for OS command injection requires the live address of the overwritten location in the address table

and the corresponding live expected value in that location, which is captured automatically from the deployed PLC.

Patch verification. ICSPatch injects the hotpatch and the hook in the `angr` simulation state to rerun the scan cycle of the control application with the exploit input to check for unbounded loops, dangerous instructions (for instance, involving stack state manipulation), memory modifications outside the range of the control application stack, and the vulnerability ruleset. If the scan cycle terminates without triggering any safety checks, the patch is deemed safe for deployment. To note, ICSPatch, like other state-of-the-art solutions, cannot verify the correctness of the patch automatically, so instead, it checks for its safety [21].

Patch deployment. Patch deployment with ICSPatch requires two main components operating on the target vulnerable PLC: 1) The local patch server that communicates with the ICSPatch server over the network, which upon request from `GET_BASE_ADDR()` sends information such as specific base addresses, and then through `DEPLOY_PATCH()` receives commands to verify memory location content, and write into the process memory. The local patch server also preprocesses the patch-related information received from ICSPatch before transferring it to the LKM patcher. 2) The LKM patcher is the main patching component. It executes the commands received from the ICSPatch server that are relayed via the local patch server to verify the content of a memory location or to write new content. To ensure the safety of operations, the PLC runtime must not be able to execute during the patching process. As will be discussed later in the experimental results, the critical part of patching (i.e., writing the hook) takes a negligible amount of time (ranging from 0.00022 ms to 0.00046 ms), thus running without interruption. Nevertheless, for extra caution, we introduce two more safety measures: 1) Just before the critical code, we adjust the `nice` process scheduling priority for the runtime by setting it to 19, which ensures that the runtime will only run when no other system process with higher priority wishes to execute. 2) Additionally, before entering the critical section of the patching process, our LKM temporarily disables kernel preemption and interrupt handling for the local processor using `preempt_disable()` and `local_irq_disable()` respectively [9,28], and re-enables them after writing the hook, ensuring an atomic operation.

At the same time, to avoid writing at inappropriate locations on the deployed PLC, our LKM verifies the memory content at said locations with the content obtained from the development PLC. Specifically, the patch and the address of the patch to be written in the address table expect empty memory locations, and only the hook that modifies the execution flow requires an exact memory content match to proceed with its installation at that specific location. The LKM patcher can write into the memory space of any process and at locations as specified by the ICSPatch server. However, patching is aborted in case of a mismatch in the memory location content during verification. The two components mentioned above op-
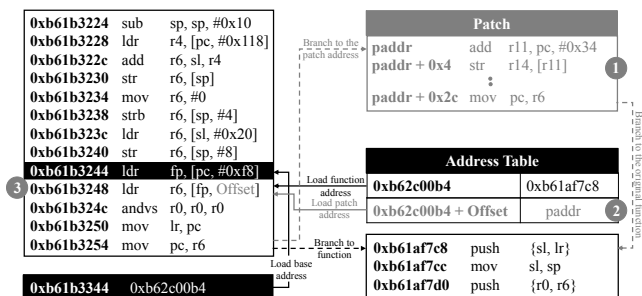
Figure 6: Steps involved in modifying the control flow for patching control applications with ICSPatch are shown in gray.

erate on the deployed PLC and allow non-intrusive patching of the live control application.

The patches generated by ICSPatch require a change in the execution flow for patching the vulnerability before returning it to normal flow. As shown in Figure 6, three primary steps are involved. 1) ICSPatch verifies an empty memory location between the control application and the address table, writing the patch at this location (*paddr*). 2) It then verifies an empty location in the address table and writes the patch address while noting its offset. To find an empty location in the address table, ICSPatch utilizes the base address of the address table loaded in an `ldr` instruction which remains unaltered during patching (for instance, at address `0xb61b3244` in Figure 6) and traverses the address table for `0x7fe` memory locations (highest value of permitted immediate offset for an `ldr` instruction) attempting to find an empty location of 32 bits. This process is performed entirely on the rehosted `angr` instance of the control application binary, not impacting the deployed PLC. 3) Finally, it modifies the `ldr` instruction at `0xb61b3248`, which was initially in this example `ldr r6, [fp]`, to `ldr r6, [fp, Offset]` for adding an offset to the original address into the address table. As a consequence of this offset, during execution, the `R6` register loads the address of the patch instead of the next function at `0xb61af7c8`. The execution flow changes to the patch, which fixes the bound value at the automatically detected memory location during vulnerability localization and passes the control back to the intended function after restoring the values of the registers employed in the patch. Figure 6 displays the changes committed by ICSPatch in gray.

5 ICSPatch Usage Scenario

Figure 7 shows three primary chronological steps involved with the real-world deployment of ICSPatch. The scenario assumes remote (SSH) access to the deployment PLC and also access to a development PLC to perform intrusive operations.
1) Preparation (if needed). If the user does not already have a copy of the control binary, they need to connect to the deployed PLC and extract it (①) over the network with a simple
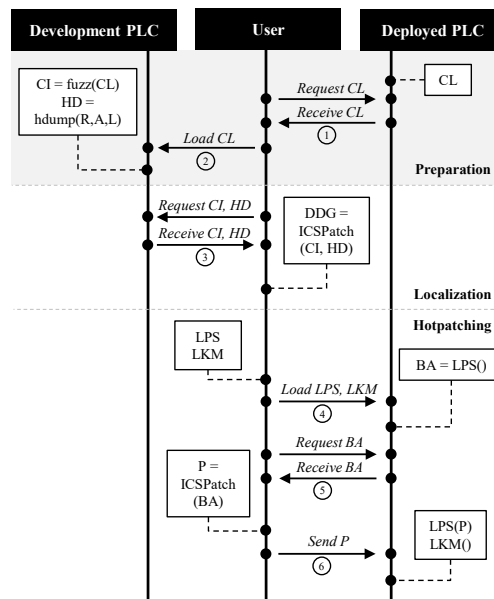
CL: Control Logic   CI: Crash Inputs   HD: Hexdumps   R: Runtime
A: Application   L: Libraries   DDG: Data Dependency Graph
LPS: Local Patch Server   LKM: Loadable Kernel Module
BA: Base Address   P: Patch

Figure 7: A chronological overview of using ICSPatch in the field. All arrows are SSH connections.

SSH connection. Once they have the binary, they load it to a second, offline PLC ("development PLC", ②), and a fuzzer is employed (e.g., ICSFuzz [55]) to uncover potential crash inputs in the control logic. When fuzzing yields a set of crash inputs, these are recorded, along with hexdumps of the runtime, control logic application, and memory.
2) Localization. The crash files described before are retrieved by the user ③, and ICSPatch is executed, localizing the vulnerability using DDGs.
3) Hotpatching. To hotpatch the vulnerability, the local patch server and LKM are loaded onto the deployed PLC ④. The local patch server extracts the runtime base address ⑤, which is provided to ICSPatch to create a patch. Consequently, the user sends the patch to the deployed PLC ⑥, where the local patch server interfaces with the LKM to patch the control application in memory.

6 Dataset and Vulnerabilities

Table 1 presents the top 5 software weaknesses for 2021 [34]. We observe that out-of-bounds write, read, improper input validation, and OS command injection apply to control binaries and can affect PLCs. Cross-site scripting, on the other hand, is not an applicable threat to the control application and, therefore, is not considered. Software weaknesses can affect PLCs since the control binary runs in the runtime context, inherits its privileges, and shares its memory spaces.

To evaluate the ability of ICSPatch to localize and patch

Table 1: Top 5 Common Weakness Enumeration (CWEs) software weaknesses for 2021 as reported by MITRE [34].

Rank	CWE	Description
1	CWE-787	Out-of-bounds write
2	CWE-79	Cross-site scripting
3	CWE-125	Out-of-bounds read
4	CWE-20	Improper input validation
5	CWE-78	OS command injection

Table 2: A diverse synthetic control application dataset for testing ICSPatch.

Shared Library	Imported Functions	Aircraft Flight Control				Anaerobic Digestion Reactor				Chemical Plant				Desalination Plant				Smart Grid			
		CWE-787	CWE-125	CWE-78	CWE-20	CWE-787	CWE-125	CWE-78	CWE-20	CWE-787	CWE-125	CWE-78	CWE-20	CWE-787	CWE-125	CWE-78	CWE-20	CWE-787	CWE-125	CWE-78	CWE-20
SysMem23	SysMemSet	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
	SysMemMove	○	○	○	○	●	○	○	○	○	○	○	○	○	○	○	○	○	○	○	○
SysMem	SysMemSet	○	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	●
	SysMemMove	○	○	○	○	○	○	○	○	●	○	○	○	○	○	●	○	○	○	○	○
	SysMemCpy	○	○	○	○	○	○	○	○	○	○	●	○	○	○	○	○	○	○	○	○
MemUtils	MemSet	○	○	○	●	●	○	○	○	○	○	○	○	○	○	○	○	●	○	○	○
	MemCpy	○	○	○	○	●	●	○	○	●	○	○	○	○	○	○	○	●	○	○	○
	BitCpy	●	○	○	○	○	●	○	○	●	○	○	○	○	○	○	○	●	○	○	○
IEC 61131-3 code																					
Out-of-bounds array index		○	○	●	○	○	○	●	○	○	○	●	○	○	○	●	○	○	○	●	○

CWE-787/CWE-125: Out-of-Bounds Write/Read   CWE-78: OS Command Injection
CWE-20: Improper Input Validation

Figure 8: Typical operation and the manifestation of out-of-bounds write, read, and OS command injection vulnerabilities in PLCs. Here, the control code represents the currently executing control application.

vulnerabilities, we created a synthetic dataset of control application binaries from five critical infrastructure sectors, including aircraft flight control, anaerobic digestion reactor, smart grid, chemical, and desalination plants, with a total of 24 vulnerable binaries. Table 2 provides a summary of the synthetic dataset, including the shared library, the imported functions utilized for each example control application, and the corresponding critical infrastructure.

CWE-787 Out-of-bounds write. An adversary can manipulate the size of the destination array by setting it to a size smaller than that of the source array in functions such as SysMemSet, SysMemMove, MemCpy and BitCpy, overwriting memory locations on the stack. These overwritten memory locations can be confined to the stack of the control application or can also cross over to the stack region of the runtime functions, crashing the MainTask. As a result, the control application or the runtime might crash (will not execute) due to overwritten variables or return addresses, as shown in Figure 8.

CWE-125 Out-of-bounds read. These vulnerabilities can manifest in functions missing bound checks when reading from a larger destination array compared to the source, enabling an adversary to read from any of the selected memory regions, including the stack, the code and data sections of the runtime, and the control application as shown in Figure 8. Out-of-bounds read samples do not lead to runtime or control application crashes.

CWE-78 OS command injection. In these examples, an adversary modifies the execution flow to execute the injected payload or point to the beginning of a ROP chain. The examples achieve this by using an array and a malicious index

manipulated by the adversary to modify the address of the following function in the address table to redirect the control flow to the malicious code rather than the expected function, as shown in Figure 8.

CWE-20 Improper input validation. An input to the control application can be from the connected sensors or other industrial communication protocols such as Open Industry Standard Unified Architecture (OPC UA). An adversary can manipulate inputs sent via the network during transit, requiring validation checks in the control application. Abstractly, all the other presented weaknesses could result from improper input validation, and we have already explored them individually. Therefore, for our dataset, we enrich it with more out-of-bounds write weaknesses.

7 Performance Evaluation

To evaluate the performance of ICSPatch, we test it against the 24 control applications presented in Section 6. We use two PLCs with different computational capabilities: A Wago PFC100 with a single-core Cortex A8 processor operating at 600 MHz and a Wago PFC200 operating at 1 GHz.

Vulnerability localization accuracy. To test the accuracy of ICSPatch, we utilize our synthetic application binary dataset with 24 binaries and the 20 vulnerable binaries from the ICS-Fuzz dataset [55]. We loaded these control applications on Wago PFC 100 and successfully detected all the vulnerabilities in the 44 vulnerable binaries (100% localization accuracy). However, it should be noted that ICSPatch detects vulnerabilities based on specific violation rules, and any function that does not follow this pattern might go undetected. The ICSFuzz dataset consisted primarily of out-of-bounds write vulnerability manifesting due to various imported functions and, in some examples, incorrect index to an array, enabling successful detection.

Table 3: ICSPatch execution timings and overheads for the 24 vulnerable binaries.

Critical Infrastructure	Vulnera-bility	Time (ms)					Mean Execution Time (µs)			Achieved Scan Cycle (µs)	Memory (Bytes)
		Vulnerability Localization	Patch Generation	Patch Verification (s)	Patch Deployment		Pre-patch	Post-patch	Difference		
					Total	Critical (µs)					
Aircraft Flight Control	CWE-20	3.06	178.57	7.71	252.13	0.22	20.09	21.06	0.97	69.59	64
	CWE-787	6.72	166.39	67.18	332.73	0.3	18.05	20.04	1.99	77.2	64
	CWE-787	4.42	143.68	33.43	232.2	0.26	25.4	27.19	1.8	74.94	64
	CWE-125	4.85	178.59	28.93	252.24	0.46	21.6	21.78	0.17	74.54	64
	CWE-78	1.54	203.03	9	230.17	0.3	16.61	20.5	3.89	72.87	56
Anaerobic Reactor	CWE-20	5.05	134.98	10	234.42	0.31	20.04	21.04	0.99	134.46	64
	CWE-787	3.78	126.87	2.86	232.07	0.35	17.02	17.07	0.05	71.46	64
	CWE-787	4.82	130.18	7.17	246.06	0.24	19.99	20.24	0.25	147.74	64
	CWE-787	4.44	124.95	2.16	223.7	0.22	21.32	21.37	0.05	74.49	64
	CWE-125	5.91	125.24	4.62	234.49	0.23	16.89	18.94	2.04	77.3	64
	CWE-125	4.96	221.38	169.1	236.26	0.28	23.98	28.08	4.09	152.96	64
	CWE-78	1.44	171.11	6.16	298.04	0.3	15.10	19.3	4.19	71.7	56
Chemical Plant	CWE-20	5	126	1.85	254.16	0.3	14.82	17.45	2.63	67.75	64
	CWE-787	3.76	183.18	424.23	236.42	0.31	36.62	54.03	17.4	148.54	64
	CWE-125	6.61	170.95	17.83	253.94	0.23	21.37	25.8	4.42	71.41	64
	CWE-78	1.64	127.89	20.26	252.72	0.26	24.09	26.64	2.54	72.38	56
Desalination Plant	CWE-20	5.44	134.94	7.4	244	0.24	18.6348	19.88	1.25	75.82	64
	CWE-787	4.81	127.11	3.32	238.35	0.23	15.717	18.07	2.35	75.79	64
	CWE-125	4.94	139.36	11.25	241.02	0.25	19.3252	19.99	0.66	73.08	64
	CWE-78	1.52	133.81	5.5	230.13	0.26	17.483	20.52	3.04	80.5	56
Smart Grid	CWE-20	3.95	133.9	3	264.83	0.27	15.1286	17.15	2.02	65.71	64
	CWE-787	3.64	134.14	9.2	247.91	0.22	26.0833	27.02	0.94	83.69	64
	CWE-125	5.73	126.27	4.6	227.34	0.23	20.0931	22.98	2.89	97.48	64
	CWE-78	1.46	222.22	6.6	232.86	0.34	25.2406	27.31	2.07	77.12	56

CWE-20: Improper Input Validation   CWE-787: Out-of-Bounds Write   CWE-125: Out-of-Bounds Read   CWE-78: OS Command Injection

Table 4: Detailed breakdown of ICSPatch used on the Aircraft Flight Control CWE-20 vulnerable binary.

Phases	Preparation	Vulnerability Localization			Patch Generation				Patch Deployment					
Steps	Hexdump Extraction	Load Hexdumps	Control App Execution	DDG Traversal	Locate Live Addresses	Hook Creation	Patch Creation	Patch Verification	MV	MW	MV	MW	MV	MW
									Address Table		Patch		Hook	
Device	Development PLC	ICSPatch (angr)			Deployed PLC	ICSPatch			Deployed PLC					
Time (s)	733.11	52.02	4.73	0.003	0.06	0.09	0.03	7.71	0.016	0.033	0.054	0.055	0.05	0.043

MV: Memory Verification   MW: Memory Write

ICSPatch execution timings. Table 3 shows that ICSPatch can successfully localize vulnerabilities in the control application in under a minute for most cases, except for the out-of-bounds write example for the Chemical Plant infrastructure, mainly due to the execution time of the control application in angr (310.4 s) because of a large number of loops in the control logic. At the same time, patch generation with ICSPatch ranges from ≈124 ms to ≈222 ms, while patch deployment ranges from ≈223 ms to ≈332 ms. This timing includes everything, such as empty memory location verification, writing the patch in memory, verifying the hook location, and the hook writing time. The last step, specifically the LKM part involving hook injection, is the critical step (since it modifies the control flow of the ICS binary) and appears in Table 3 as the "Critical" column. As evident in the Critical column, LKM hook writing time is extremely fast, in the order of µs, significantly reducing the probability of the runtime executing while hook writing takes place. Nevertheless, ICSPatch takes additional precautions for ensuring atomic hook writing operation as outlined in Section 4.3. Table 4 takes one of the binaries (CWE-20 for Aircraft Flight Control) and further breaks down the timings. The time for writing the patch hook in memory, as shown in Table 4, includes the ICSPatch communication with the user system, the local patch server time, and the LKM time. Table 3 presents only the critical LKM time, which is a subset of the total time.

ICSPatch overhead. Table 3 also shows the latency incurred by the control application execution as a result of the patch in µs. The increase in the execution time is negligible (0.05-17.4 µs) compared to typical scan cycles of critical infrastructures, which range from a few milliseconds to a few seconds (exact scan cycle values are process and setup specific), further illustrated in the case study in Section 8. Patching increases latency by only a few µs; however, the out-of-bounds write example for the chemical plant is an exception with an increase of ≈17.4 µs. This increase is independent of the patch and is a consequence of the structure of the control application. Unlike other examples, an exploit input, in this case, triggers the execution of additional for loops for larger array initialization. The patch only fixes the input to the MemUtils.MemCpy that results in the out-of-bounds write. Initializing a larger array due to an exploit input is a legitimate operation as it does not overwrite any memory location, leading to an increase in time.
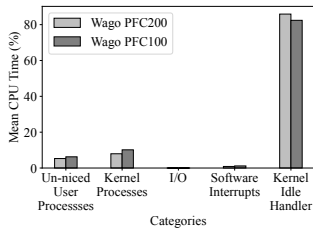
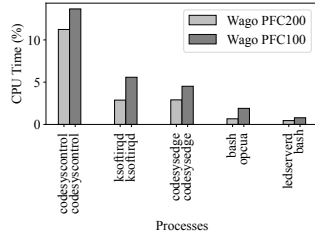Figure 9: An overview of CPU utilization by different operations on a PLC.

Figure 10: CPU utilization of the top five processes on a PLC.
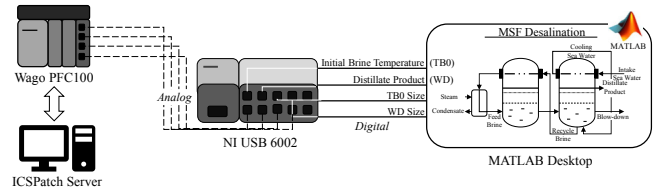

Figure 11: Experimental setup for patching out-of-bound write vulnerability in an MSF desalination plant HIL.

With regards to the extra memory needed for the patch, presented in Table 3, there are three crucial parts to the patches generated by ICSPatch: The main patch, the address of the patch written in an empty location inside the address table, and the hook. The patch address is 32 bits for our target platforms, and the hook size depends on the target compilation mode for the runtime. In our experiments, the hook for our Wago PFC100 (ARM) is 32 bits, as it consists of modifying an `ldr` instruction with an immediate operand. The main patch for out-of-bounds write and read vulnerabilities is based on 13 assembly code lines and the 32 bit bound limit. Combined with the patch address and the hook, this results in a 64-byte patch. On the other hand, the skeleton patch for resolving OS command injection vulnerability does not include a bound limit and also does not require us to load the base address to branch to the patch location, thus saving an additional 4 bytes by the omission of an `ldr` instruction, making the total patch size 56 bytes.

Non-intrusiveness. ICSPatch utilizes an LKM-based patcher for writing the patch at appropriate memory locations. Inserting the hook (modified `ldr` instruction) is the most critical part of the patching process, resulting in a modified execution flow toward the patch. Since PLCs often execute straightforward dedicated process logic, they lack heavy computation power. For instance, Wago PFC100 and PFC200 (Generation 2) utilize single-core Cortex-A8 processors. To execute multiple processes, such processors utilize time slicing. As discussed earlier, the LKM patcher inserts the patch when the runtime process is switched out and sleeping. To further ensure that patching commences safely in an atomic fashion, the runtime process priority is adjusted, and our LKM temporarily disables kernel preemption and interrupt handling.

We write a `bash` script for Wago PFC100 and PFC200 to measure the execution and sleeping time for the runtime. We measure running/sleeping times in an interval of 1 hour (3600 seconds). We observe that, on average, for a basic out-of-the-box installation, the runtime process utilizes ≈492 s (≈13.66%) and ≈404 s (≈11.23%) of total CPU time for Wago PFC100 and PFC200, respectively, while executing the control application binary for regulating the steam flow into the brine heater for thermal desalination plants, explained further in Section 8. While the `codesyscontrol` process still

occupies the majority of the CPU time when compared to any other processes, as shown in Figure 10, the majority of the time is spent in the kernel Idle handler, ≈2966 s (≈82.41%) and ≈3092 s (≈85.6%) for Wago PFC100 and PFC200 respectively, as shown in Figure 9. This affirms that the LKM patcher can utilize this idle time on these single-core PLC devices to non-intrusively patch the currently executing control application binaries.

8 Case Study: Hotpatching Out-of-bounds Write Vulnerability in a Desalination Plant

Hardware-in-the-loop setup. For testing the applicability of ICSPatch on PLCs deployed in the field, monitoring and regulating physical processes in critical infrastructure, we perform a Hardware-in-the-Loop (HIL) experiment. The experiment setup is shown in Figure 11, consisting of a MATLAB Simulink model for a Multi-Stage Flash desalination plant validated against the Khubar II plant in Saudi Arabia and utilized for HIL experiments in literature [12, 41–43]. The model controls the steam flow into the heating section based on the initial brine temperature (TB0) and the distillate output (WD). It runs on a host connected to NI USB 6002, a data acquisition (DAQ) device providing multiple digital and analog I/Os, which converts the digital inputs from the Simulink model to analog outputs and then feeds them to the Wago PFC100 PLC. The PLC sends the output of the control logic as analog values back to the DAQ device, which converts it to digital values that the Simulink model reads. We extract TB0 and WD from the Simulink model and send them to the PLC. ICSPatch server is connected via SSH to the PLC.

Out-of-bounds write. The control logic executing on Wago PFC100 and regulating the critical physical process utilizes `MemUtils.MemCpy`, an imported function for copying memory contents into temporary variables in a function. An adversary can maliciously modify the bound value to trigger an out-of-bounds write on critical runtime variables on the stack, resulting in a crash. In our example, as shown in Figure 12, an adversary triggers an out-of-bounds write attack at the 100th cycle of the model by maliciously modifying the TB0 size and overwriting critical sections of the runtime stack.

Patching. ICSPatch extracts memory segments from the development PLC loaded with the exploit input, rehosts them in `angr`, and performs concolic execution to detect vulnerabil-
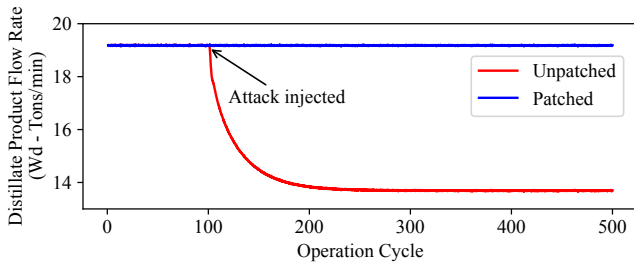
Figure 12: Distillate product flow rate for control application with unpatched and patched out-of-bounds write vulnerability.

Vendor	Firmware Base	Supports Codesys
ABB	Linux	Yes
Beckhoff	FreeBSD	Yes
Bosch Rexroth	Linux	Yes
Mitsubishi	N/A	Yes
Rockwell Automation	VxWorks	No
Schneider Electric	VxWorks	Yes
Siemens	OpenBSD	No
WAGO	Linux	Yes

ities. Once it detects the vulnerability, it localizes it, selects a patch location, connects with the deployment PLC, and patches it on the deployment PLC by adding a bound check to the size of the TB0 parameter, which is the root cause of the vulnerability in the control logic.

Figure 12 shows the consequences of unpatched control logic regulating the MSF process (shown in red) with an out-of-bounds write vulnerability. The output Wd flow rate decreases significantly due to the attack beginning on the 100th operational cycle of the simulation. The adversary crashes the runtime by triggering an out-of-bounds write vulnerability. However, a patched instance of the control application is protected against the attack, and the control logic functions as expected without any output changes.

9   Scalability Discussion

Localizing and patching more vulnerabilities. We include eight different memory-related functions in our dataset and successfully patch the vulnerable control binaries using IC-SPatch. The patches for these memory-related functions implement a bound check and are therefore applicable to all the relevant examples with minor modifications to the live memory addresses of the deployed PLC. In most cases, extending ICSPatch to support other functions requires adding the appropriate node to the DDG for automated vulnerability localization.

Hotpatch deployment. ICSPatch requires a kernel-level component (LKM, in our case, or a dedicated patch driver) for patching the control applications running on Linux operating systems. However, PLCs may run bare-metal firmware. In that case, ICSPatch also provides a JTAG-based patcher out-of-the-box to support such devices, which we have successfully tested on a BeagleBone Black (BBB). Similarly, HERA implements a low-priority task, and RapidPatch assumes a traditional OTA transfer tunnel for its patch deployment.

Targeting more devices. We extend ICSPatch to work with Codesys runtime running on BBB. The main differences appear in the methodology used for branching between functions. The Codesys variant for Wago PFC100 saves the `R14` and explicitly modifies `PC` (Figure 1), whereas, on BBB a `blx` (branch with link exchange) instruction is used for branching to the next function. Due to the discrepancies in branching,

the patch requires minor modifications to accommodate the use of different registers and branching mechanisms. Furthermore, the runtime variant operating across devices might also vary in the mode of its compilation. For instance, the runtime for BBB is compiled in Thumb mode, whereas it is ARM for Wago PFC100. Therefore a branch to the following function on BBB expects the code to be in Thumb, whereas it has to be in ARM on PFC100, requiring the patches to be compiled accordingly.

ICSPatch can be extended to support other vendors, including Siemens, Rockwell, Mitsubishi, and Schneider, who collectively share approximately 75% of the PLC market [51]. While older PLCs primarily used monolithic proprietary firmware containing only in-house developed components, this is no longer the case, as shown in Table 5. Instead, the ICS market has transitioned to a model that utilizes commodity operating systems and third-party components, enabling the deployment of tools like ICSPatch. For instance, our analysis of other PLC firmware reveals that the Siemens SIMATIC S7 line utilizes OpenBSD while Rockwell Automation and Schneider Electric use the VxWorks kernel. Looking closer, we noticed that the firmware bundles had numerous proprietary and third-party open-source components. Such components included JavaScript libraries like jQuery, utilities like OpenSSL, ICS-specific software like the Codesys runtime and Unified Automation's OPC UA SDK, and even more bespoke dependencies like the Dinkumware C++ framework, IBM Rational Rhapsody, and the Electronic Arts standard template library. Beyond the underlying OS and platform support, ICSPatch also depends on the disassembly of the control binary during the parsing phase. Manufacturers like Mitsubishi, Beckhoff, and Schneider Electric can be easily supported since their PLC ecosystems integrate the Codesys runtime. PLCs from other vendors like Siemens can be supported using the corresponding tools in their respective ecosystems, like the JEB S7 PLC Block Decompiler [49] or the Rizin framework [38].

10   Limitations

Codesys specific implementation. As mentioned, ICSPatch has three main components: Vulnerability identification, lo-

calization, and hotpatching. Vulnerability localization based on DDG does not assume Codesys-specific compilation and can work with other platforms. Likewise, modifications to the `angr` rules can also identify vulnerabilities in control applications compiled for other platforms. However, the patches are designed with the Codesys compiler as a target. Therefore, some effort is required to create appropriate skeleton patches to extend applicability to other platforms.

Patching multiple vulnerabilities. A control application may have multiple vulnerabilities; for instance, the same malicious input (bound) utilized by two or more `SysMem.SysMemSet` functions for initializing memory. However, in its current version, ICSPatch only patches the first instance of the vulnerability.

Requiring user input for patching. While, for the most part, ICSPatch is automated to identify, locate, and deploy the patch, it cannot automatically suggest an appropriate bound check for memory-related vulnerabilities (for out-of-bounds read/write), thus requiring user input during patch generation. Nevertheless, patching OS command injection vulnerabilities is entirely automated as it involves fixing the address table.

Patching on multicore devices. ICS devices, for the most part, do not employ powerful multicore processors as they execute simple and definite logic. The non-intrusiveness of ICSPatch relies on the single core execution of all processes. A more elaborate mechanism for ensuring non-intrusiveness would be required in multicore cases.

11 Related Work

Automated vulnerability identification and localization. Machine Learning can be used for vulnerability localization; for instance, Rebecca et al. utilize deep feature representation learning on lexed source code [45]. Devign employs graph-level classification with a graph neural network-based model to extract valuable features learned from rich node representations [61]. On the other hand, for binaries, Runhao et al. propose a gradient-guided vulnerability localization method: DeepVL, utilizing execution traces to filter vulnerable basic blocks [26]. Finally, Shiqi et al. utilize statistical localization technique to discover the root cause of the bug, given just one exploit input [47].

Hotpatching in Android. Yue et al. put forth KARMA, a system that produces patches written in high-level memory-safe language applicable at multiple levels in the kernel to enable malicious input filtering [7]. VULMET utilizes weakest precondition reasoning to modify official patches into semantically correct hotpatches [58]. Yue et al. propose LIB-BANDAID for automatically generating updates for third-party libraries [14]. Finally, bowknots for kernel bugs nullify the side effects of currently executing syscalls triggering bugs [54].

Hardware-based hotpatching. Traditional hotpatching employs trampolines; however, approaches such as InstaGuard use basic debugging primitives supported by ARM CPUs to enable a rule-driven hotpatching mechanism without injecting any code [6]. HERA, supported on ARM Cortex-M3/M4 devices, utilizes its Flash Patch and Breakpoint (FPB) unit to redirect the vulnerable code to the patch [37]. On the other hand, a recently proposed technique, RapidPatch, facilitates a dynamic code replacement technique utilizing eBPF virtual machines for executing patches on an embedded device with resource constraints [21].

Comparison to state-of-the-art. HERA and RapidPatch are the current state-of-the-art hotpatching solutions for RTOSes employed for embedded devices, thus, the closest to our work. HERA utilizes FPB in Cortex-M3/M4, and RapidPatch employs eBPF virtual machines. However, these solutions cannot directly be employed for PLC control binaries because:

- Control applications compiled into nonstandard file format run in the context of a proprietary piece of software, the runtime. So, hotpatching requires understanding control application internals while handling runtime abstractions.
- Vulnerabilities manifest in the imported functions, and due to their shared nature, they require patching within the boundaries of the control application, necessitating localization.
- Unlike the assumption made by the current state-of-the-art techniques, ICSPatch does not have access to any upstream patch source for creating the hotpatch.
- HERA utilizes hardware-specific debugging features for redirecting execution flow to the patch (Cortex-M3/M4), limiting its applicability. For instance, Wago PFC100 and PFC200 employ Cortex A8, lacking the FPB debug unit.
- RapidPatch utilizes ePBF virtual machines, which cannot directly be applied for control applications as they execute inside the runtime with the proprietary file format.

12 Conclusion

This work proposes ICSPatch, a tool for automated vulnerability identification by detecting violations to security specifications, localization by traversing a DDG, and hotpatching using an LKM-based patcher. We implement ICSPatch for the Codesys platform, deployed on over 400 known ICS devices from 80 industrial device vendors. We successfully patch out-of-bounds write/read, OS command injection, and invalid input validation vulnerabilities spread across a dataset of 24 synthetic control application binaries while only incurring negligible execution and memory overheads. We also demonstrate ICSPatch patching a live PLC controlling a HIL simulation of an industrial process.

Acknowledgment

References

[1] Angr. https://angr.io/. [Online; Accessed: May 2022].

[2] Borja Fernández Adiego, Dániel Darvas, Enrique Blanco Viñuela, Jean-Charles Tournier, Simon Bliudze, Jan Olaf Blech, and Víctor Manuel González Suárez. Applying model checking to industrial-sized plc programs. *IEEE Transactions on Industrial Informatics*, 11(6):1400–1410, 2015.

[3] Mitsubishi Automation. Codesys: A leading open programming and control environment that can be deployed directly on mitsubishi electric's c controller (iq platform). https://gb.mitsubishielectric.com/fa/solutions/efactory/alliance/partner/codesys. [Online; Accessed: October 2022].

[4] Sidi Ould Biha. A formal semantics of plc programs in coq. In *2011 IEEE 35th Annual Computer Software and Applications Conference*, pages 118–127. IEEE, 2011.

[5] Hugh Boyes, Bil Hallaq, Joe Cunningham, and Tim Watson. The industrial internet of things (iiot): An analysis framework. *Computers in industry*, 101:1–12, 2018.

[6] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *2018 Network and Distributed System Security Symposium (NDSS'18)*, 2018.

[7] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1253–1270, 2017.

[8] Anton Cherepanov. Win32/industroyer: A new threat for industrial control systems. *White paper, ESET (June 2017)*, 2017.

[9] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux device drivers*. " O'Reilly Media, Inc.", 2005.

[10] Dániel Darvas, Borja Fernández Adiego, András Vörös, Tamás Bartha, Enrique Blanco Viñuela, 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.

[11] 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.

[12] Constantine Doumanidis, Prashant Hari Narayan Rajput, and Michail Maniatakos. Icsml: Industrial control systems machine learning inference framework natively executing on iec 61131-3 languages. *arXiv preprint arXiv:2202.10075*, 2022.

[13] Ruian Duan, Ashish Bijlani, Yang Ji, Omar Alrawi, Yiyuan Xiong, Moses Ike, Brendan Saltaformaggio, and Wenke Lee. Automating patching of vulnerable open-source software versions in application binaries. In *NDSS*, 2019.

[14] Yue Duan, Lian Gao, Jie Hu, and Heng Yin. Automatic generation of non-intrusive updates for Third-Party libraries in android applications. In *22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019)*, pages 277–292, Chaoyang District, Beijing, September 2019. USENIX Association.

[15] Nicolas Falliere, Liam O Murchu, and Eric Chien. W32. stuxnet dossier. *White paper, Symantec Corp., Security Response*, 5(6):29, 2011.

[16] Barbara Filkins, Doug Wylie, and AJ Dely. Sans 2019 state of ot/ics cybersecurity survey. *SANS Technology Institute*, 2019.

[17] 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.

[18] Ivan Gotovchits, Rijnard Van Tonder, and David Brumley. Saluki: finding taint-style vulnerabilities with static property checking. In *Proceedings of the NDSS Workshop on Binary Analysis Research*, volume 2018, 2018.

[19] Vincent Gourcuff, Olivier De Smet, and J-M Faure. Efficient representation for formal verification of plc programs. In *2006 8th International Workshop on Discrete Event Systems*, pages 182–187. IEEE, 2006.

[20] CODESYS Group. Codesys device directory. https://www.codesys.com/download/download-center.html. [Online; Accessed: April 2022].

[21] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. Rapidpatch: Firmware hotpatching for real-time embedded devices. In *31th USENIX Security Symposium (USENIX Security 22)*, 2022.

[22] Yanhong Huang, Xiangxing Bu, Gang Zhu, Xin Ye, Xiaoran Zhu, and Jianqi Shi. Kst: Executable formal semantics of iec 61131-3 structured text for verification. *IEEE Access*, 7:14593–14602, 2019.

[23] Janztec. empc linux kernel sources. `https://github.com/janztec/empc-arpi-linux`. [Online; Accessed: June 2022].

[24] Edwin Franco Myloth Josephlal, Sridhar Adepu, Zheng Yang, and Jianying Zhou. Enabling isolation and recovery in plc redundancy framework of metro train systems. *International Journal of Information Security*, pages 1–13, 2021.

[25] Anastasis Keliris and Michail Maniatakos. Icsref: A framework for automated reverse engineering of industrial control systems binaries. *Network and Distributed Systems Security (NDSS)*, 2019.

[26] Runhao Li, Chen Zhang, Chao Feng, Xing Zhang, and Chaojing Tang. Locating vulnerability in binaries using deep neural networks. *Ieee Access*, 7:134660–134676, 2019.

[27] 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, 2016.

[28] Robert Love. *Linux kernel development*. Pearson Education, 2010.

[29] Luca Luccarini, Gianni Luigi Bragadin, Gabriele Colombini, Maurizio Mancini, Paola Mello, Marco Montali, and Davide Sottara. Formal verification of wastewater treatment processes using events detected from continuous signals by means of artificial neural networks. case study: Sbr plant. *Environmental Modelling & Software*, 25(5):648–660, 2010.

[30] Jim Luo, Myong Kang, Emmanuel Bisse, Mike Veldink Dmitriy Okunev, Scott Kolb, Joseph G Tylka, and Arquimedes Canedo. A quad-redundant plc architecture for cyber-resilient industrial control systems. *IEEE Embedded Systems Letters*, 2020.

[31] Yue Ma, Mingshi Li, Zhenyu Yin, and Mengjia Lian. design of safety plc execution unit based on redundancy structure of heterogeneous dual-processor. In *2017 10th International Conference on Intelligent Computation Technology and Automation (ICICTA)*, pages 364–368. IEEE, 2017.

[32] Aravind Machiry, Nilo Redini, Eric Camellini, Christopher Kruegel, and Giovanni Vigna. Spider: Enabling fast patch propagation in related software repositories. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1562–1579. IEEE, 2020.

[33] 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.

[34] MITRE. Weaknesses in the 2021 cwe top 25 most dangerous software weaknesses. `https://cwe.mitre.org/data/definitions/1337.html`. [Online; Accessed: May 2022].

[35] Josh Newell, Linna Pang, David Tremaine, Alan Wassyng, and Mark Lawford. Translation of iec 61131-3 function block diagrams to pvs for formal verification with real-time nuclear application. *Journal of Automated Reasoning*, 60(1):63–84, 2018.

[36] James Newsome and Dawn Xiaodong Song. Dynamic taint analysis for automatic detection, analysis, and signaturegeneration of exploits on commodity software. In *NDSS*, volume 5, pages 3–4. Citeseer, 2005.

[37] Christian Niesler, Sebastian Surminski, and Lucas Davi. Hera: Hotpatching of embedded real-time applications. In *28th Network and Distributed System Security Symposium (NDSS)*, 2021.

[38] Rizin Organization. Rizin: Free and open source reverse engineering framework, 2020. `https://rizin.re/`. [Online; Accessed: October 2022].

[39] 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.

[40] Nicole Perlroth and Clifford Krauss. A cyberattack in saudi arabia had a deadly goal. experts fear another try. *New York Times*, 15, 2018.

[41] Prashant Hari Narayan Rajput and Michail Maniatakos. Towards non-intrusive malware detection for industrial control systems. In *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1703–1706. IEEE, 2021.

[42] 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.

[43] Prashant Hari Narayan Rajput, Esha Sarkar, Dimitrios Tychalas, and Michail Maniatakos. Remote non-intrusive malware detection for plcs based on chain of trust rooted in hardware. In *2021 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 369–384. IEEE, 2021.

[44] Bosch Rexroth. Codesys group partner. `https://apps.boschrexroth.com/microsites/ctrlx-automation/en/ctrlx-world/partner/codesys-en/`. [Online; Accessed: October 2022].

[45] Rebecca Russell, Louis Kim, Lei Hamilton, Tomo Lazovich, Jacob Harer, Onur Ozdemir, Paul Ellingwood, and Marc McConley. Automated vulnerability detection in source code using deep representation learning. In *2018 17th IEEE International Conference on Machine Learning and Applications (ICMLA)*, pages 757–762, 2018.

[46] Saranyan Senthivel, Irfan Ahmed, and Vassil Roussev. Scada network forensics of the pccc protocol. *Digital Investigation*, 22:S57–S65, 2017.

[47] Shiqi Shen, Aashish Kolluri, Zhen Dong, Prateek Saxena, and Abhik Roychoudhury. Localizing vulnerabilities statistically from one exploit. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 537–549, 2021.

[48] SIEMENS. Simatic iot2000 yocto board support package. https://github.com/siemens/meta-iot2000/. [Online; Accessed: June 2022].

[49] PNF Software. Jeb s7 plc block decompiler, 2017. https://www.pnfsoftware.com/jeb/plc. [Online; Accessed: October 2022].

[50] Ralf Spenneberg, Maik Brüggemann, and Hendrik Schwartke. Plc-blaster: A worm living solely in the plc. *Black Hat Asia*, 16:1–16, 2016.

[51] Statista. Global plc market share as of 2017, by manufacturer, 2017. https://www.statista.com/statistics/897201/global-plc-market-share-by-manufacturer/. [Online; Accessed: October 2022].

[52] Keith Stouffer, Joe Falco, and Karen Scarfone. Guide to industrial control systems (ics) security. *NIST special publication*, 800(82):16–16, 2011.

[53] WIBU Systems. Protecting and licensing codesys applications. https://www.wibu.com/us/products/embedded-security-kits/embedded-security-kit-for-codesys.html. [Online; Accessed: June 2022].

[54] Seyed Mohammadjavad Seyed Talebi, Zhihao Yao, Ardalan Amiri Sani, Zhiyun Qian, and Daniel Austin. Undo workarounds for kernel bugs. In *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, August 2021.

[55] Dimitrios Tychalas, Hadjer Benkraouda, and Michail Maniatakos. Icsfuzz: Manipulating i/os and repurposing binary code to enable instrumented fuzzing in {ICS} control applications. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, 2021.

[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*, pages 1092–1105, 2016.

[57] WAGO. Firmware sdk for pfc200-g1 and pfc100 family. https://github.com/WAGO/pfc-firmware-sdk. [Online; Accessed: June 2022].

[58] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 2397–2414. USENIX Association, August 2020.

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

[60] Zhiyuan Zheng and AL Narasimha Reddy. Towards improving data validity of cyber-physical systems through path redundancy. In *Proceedings of the 3rd ACM Workshop on Cyber-Physical System Security*, pages 91–102, 2017.

[61] Yaqin Zhou, Shangqing Liu, Jingkai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

[62] Saman Zonouz, Julian Rrushi, and Stephen McLaughlin. Detecting industrial control malware using automated plc code analytics. *IEEE Security & Privacy*, 12(6):40–47, 2014.