Save the Bruised Striver: A Reliable Live Patching Framework for Protecting Real-World PLCs

Ming Zhou*
mingzhou@njust.edu.cn
School of Cyber Science and Engineering
Nanjing University of Science and Technology

Haining Wang
hnw@vt.edu
Department of Electrical and Computer Engineering
Virginia Tech

Ke Li
like3@sgepri.sgcc.com.cn
NARI Group Corporation State Grid
Electric Power Research Institute

Hongsong Zhu, Limin Sun†
{zhuhongsong,sunlimin}@iie.ac.cn
School of Cyber Security, University of Chinese Academy of Sciences; Institute of Information Engineering, CAS

Abstract

Industrial Control Systems (ICS), particularly programmable logic controllers (PLCs) responsible for managing underlying physical infrastructures, often operate for extended periods without interruption. Thus, it is challenging to patch security vulnerabilities of ICS in a timely manner after disclosure because it often necessitates waiting for a rare downtime window. While live patching has been introduced to avoid downtime and maintenance costs, conventional live patching methods are not viable for closed-source PLCs. Without the source code, it is difficult to understand the system behaviors and determine binary patch equivalence. To address these challenges, we present a Reliable Live Patching framework called RLPatch for applying live patches to third-party binary without source code. We design RLPatch to capture real-time conditions and dynamic behaviors of PLCs, which enables DevOps engineers to identify major non-recoverable fault (MNRF) vulnerabilities and generate hot patches. The core of RLPatch is an update agent that inserts breakpoints over the original MNRF code and then directs execution to the patches. To ensure system reliability, we use the unique constraints of PLCs to integrate the update processes with the scan cycle. We leverage RLPatch to patch 20 real vulnerabilities in three widely used Rockwell PLCs. We evaluate RLPatch in a real-world gas pipeline, demonstrating its reliability and effectiveness in practice.

*The work was done while visiting Virginia Tech.
†The corresponding author.

EuroSys'24, April 22–25, 2024, Athens, Greece

ACM ISBN 979-8-4007-0437-6/24/04.
https://doi.org/10.1145/3627703.3650068

CCS Concepts: • **Computer systems organization → Embedded and cyber-physical systems**; • **Security and privacy → Vulnerability management**.

Keywords: Live Patching, Firmware Security, Programmable Logic Controller, Binary Patch, Vulnerability Analysis.

ACM Reference Format:
Ming Zhou, Haining Wang, Ke Li, and Hongsong Zhu, Limin Sun. 2024. Save the Bruised Striver: A Reliable Live Patching Framework for Protecting Real-World PLCs. In *Nineteenth European Conference on Computer Systems (EuroSys '24), April 22–25, 2024, Athens, Greece.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3627703.3650068

1 Introduction

Programmable Logic Controllers (PLCs) are one of the most frequently used automatic control components within industrial control systems (ICS). They bridge the digital and physical worlds by tracking, controlling, and interconnecting industrial processes, such as gas pipelines and water treatment systems. In contrast to regular desktop computers with a four-year design life-cycle, it is common that PLCs installed even thirty years ago are still in use and play a crucial role in the stability and continuity of the ICS [23, 36]. It is often difficult and sometimes infeasible to replace these outdated controllers with modern ones [23]. For instance, these outdated strivers may have been used before the communications network standards. Therefore, updating control systems entails more than just reconfiguring the new controllers; it mandates the complete reconstruction of the communication infrastructure within the ICS. Some new controllers, such as SIMATIC ET 200M [41] and ABB Pluto B20 [7], use hot swapping to provide run-time device replacement. However, hot swapping raises the PLCs' cost and does not cover the outdated PLCs. PLCs were originally devised to replace conventional hard-wired relays and timers in manufacturing processes, with little or almost no security considerations in a highly isolated environment [18, 23]. With the increase of connectivity to enterprise networks and

the Internet, these bruised strivers have become susceptible to cyber-attacks [33, 35, 46, 47].

PLC vendors often apply patches late or overlook them, especially when dealing with outdated controllers. Prominent PLC vendors like Rockwell Automation [4], Schneider Electric [15], and Siemens Automatic [40] have formed security teams to investigate severe vulnerabilities and issue security advisories. However, official PLC patches have traditionally focused on resolving functionality and stability issues within the original code rather than bolstering security. Moreover, there is a crucial requirement for operators: they must pause the operation of target controllers before installing the new firmware that contains the necessary patches. This temporary disruption potentially leads to service interruptions, including loss of network connectivity or potential damage to field devices. Thus, the tasks of fixing high-risk vulnerabilities have to wait until the next maintenance period reaches. However, the time span between two maintenance periods for control systems is considerably longer than that for traditional IT systems and can extend over several months or even years. This opens a larger window of opportunity for a hacking group to exploit a not-yet-updated controller.

Live patching can upgrade a running system or software without rebooting or restarting any process. Researchers have spent significant efforts to keep Linux servers, Android systems, and embedded systems updated to the highest security level [3, 9, 10, 21, 30, 34]. These solutions typically rely on the availability of source code for the target device and pre-existing vulnerability information. Unfortunately, such an assumption is invalid for outdated controllers with exceedingly rare documentation. Due to the lack of access to the code, developers often resort to black-box testing and reverse engineering to understand the intricate behaviors and internal configurations of PLC systems, which can be time-consuming and may not always yield accurate or comprehensive results. The absence of source code also hinders direct binary modification, and thus complicating the verification process to ensure that patched binaries behave consistently with the original system except for the intended alterations. Furthermore, the development of efficient test cases and debugging procedures is impeded by the limited error reporting and logging capabilities inherent in PLCs.

In this paper, we propose a novel Reliable Live Patching framework called `RLPatch` that allows for prompt firmware updates at the hardware level. RLPatch is the first practical live patching framework that targets closed-source PLCs. More specifically, we first propose a precise vulnerability detection method to capture PLCs' dynamic behaviors by using the interactions between the control programs and the logic instruction library of a controller, which can identify and locate vulnerable code from the firmware with no meta-information about the source code. After vulnerability detection, we develop an update agent to patch firmware running on real PLCs by utilizing the exception-handling process of a processor. To ensure consistency across updated code versions, we provide an integration mechanism that facilitates the application of patches by inserting the update point at the rear of its scan cycle.

To demonstrate the reliability and effectiveness of RLPatch, we implement a prototype and deploy it on three widely used Rockwell PLCs. We leverage RLPatch to successfully repair 20 real-world vulnerabilities during the controller run with constant and negligible overhead. We also evaluate the system impact that RLPatch introduces on a real-world gas pipeline by launching three attack scenarios. The evaluation results show that RLPatch can quickly recover and shield the entire control system from attacks.

Note that the main technical innovations of RLPatch lie in three aspects: (1) its precise vulnerability detection method, (2) its non-intrusive update service, and (3) its adaptive integration mechanism. The vulnerability detection is sufficiently flexible because the debug logic unit it uses to capture PLCs' dynamic behaviors widely exists in embedded processors. The update service is sufficiently flexible because the exception handling mechanism exists in industrial controllers. The service is also safe because the update point sits at the idle time of each scan cycle. The essential technical points of the integration process, such as the exception vectors and update validation algorithm, are thoroughly analyzed to guarantee seamless patching services.

The rest of this paper is structured as follows. Section 2 provides a background on the programmable logic controller and ARM exception handling and presents our threat model. Section 3 details three principle components and how RLPatch uses them to hot-patch controllers at the object code level. Section 4 evaluates RLPatch on three individual Rockwell PLCs and presents an application of RLPatch on a real-world industrial control system. Section 5 discusses some limitations and future extensions of RLPatch. Section 6 surveys related works, and finally, Section 7 concludes this paper.

2 Background and Threat Model

In this section, we first provide basic knowledge of PLC and ARM exception handling. Then, we present our threat model.

2.1 Programmable Logic Controller (PLC)

PLCs are hard real-time systems. The core hardware component of PLC is the controller module that comprises several elements, such as processor and memory. The processor interprets input conditions and outputs the control actions according to the control logic programs. ARM processors are quite common in the PLC world to reduce costs and power consumption [38]. The data memory stores the firmware, and the control programs are typically stored in battery-backed memory such as static random access memory (SRAM). Besides the data and program memory types, PLC often furnishes runtime memory to speed up the interaction between

Table 1. Summary of PLCs with published JTAG ports. RA for Rockwell Automation, SA for Siemens Automatic, and SE for Schneider Electronic.

PLC Brand	Mfr.	Processor	Source
ControlLogix 1756-L61	RA	5561	[6]
ControlLogix 1756-L61	RA	5555	[27]
CompactLogix 1769-L18ER	RA	L1	[19]
CompactLogix 1769-L16ER	RA	-	[26]
MicroLogix 1763-L16AWA	RA	-	[31]
SIMATIC S7-1200	SA	1211C	[45]
SIMATIC S7-1200	SA	1212C	[45]
SIMATIC S7-1200	SA	1214C	[45]
SIMATIC S7-1200	SA	1215C	[45]
SIMATIC S7-1200	SA	1217C	[45]
SIMATIC S7-1200	SA	1212FC	[45]
SIMATIC S7-1200	SA	1214FC	[45]
SIMATIC S7-1200	SA	1215FC	[45]
Modicon M221	SE	-	[31]

firmware and control programs. The runtime memory enables us to change the firmware directly rather than rely on dedicated Flash Patch and Breakpoint (FPB) hardware.

PLC has two types of code: the control programs and the firmware. The control programs defined by operators directly manage the sensors and actuators in the physical world. A control program usually comprises a set of tasks, and each task contains a sequence of logic instructions, such as timer and alarm. Typically, operators adopt vendor-specific engineering software to create control programs. One example is the RSLogix 5000 developed by Rockwell Automation. Each input or output parameter of the control program is called a "controller tag" that points to a location in memory where its data is stored. Thus, we can locate the vulnerable code in the running memory through their logic instructions and associated tags. To ensure high real-time and reliable performance, both the control programs and the firmware adopt static linking rather than run-time linking.

Firmware is the actual program a PLC uses to execute control programs. The PLC firmware commonly comes with real-time schedulers that call the runtime system to handle the control tasks continuously to ensure that a connected physical process does not halt. In each cycle, PLC gathers data from input devices, runs the control programs, and updates the output data to control a physical process called a scan cycle. The time it takes for the controller to complete one scan cycle is called the scan time. The watchdog timer is a significant term related to the scan time. It monitors whether the controller is executing control logic programs within the specified scan time. If it exceeds a pre-set limit called the watchdog time, the timer will generate a timeout signal to stop the processor, preventing unnecessary damage to the control system. We observe that the scan time is much shorter than the watchdog time. The former

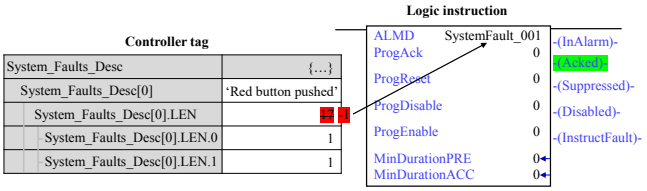


Figure 1. An MNRF attack by modifying the string length of the control program in Rockwell 1756-L61 PLC.

ranges from hundreds to thousands of microseconds, while the latter ranges from tens to hundreds of milliseconds. For instance, the scan time is a function of the logic program and the actual path executed through it. The watchdog time is a configurable component of the logic program. The overall statement of there being some slack probably holds for most PLC applications. We can leverage the slack (several milliseconds) between them to fulfill our live patching. A successful live patch takes several milliseconds or less and can claim to have no downtime.

2.2 Exception Handling

Most embedded processors feature three underlying functions: debug interface, exception generation mechanism, and exception handlers. The debug interface such as the Joint Test Action Group (JTAG) and Universal Asynchronous Receiver/Transmitter (UART), enables product manufacturers to communicate with the processor through a built-in debug channel. We observe that the JTAG ports are popular in PLCs, and PLC owners have no ability to lock them. Table 1 shows that at least 14 widely used PLCs from the top three vendors have functional JTAG ports.

Processors support multiple exception types, and users can use different conditions to trigger them. A `Prefetch Abort` exception can be triggered by configuring hardware breakpoints through a debug logic unit, such as embedded in-circuit emulation (EmbeddedICE) logic [2] and flash patch and breakpoint unit (FPB) [1]. Users can also employ a software breakpoint (processor instruction) to trigger an `Undefined Instruction` exception. Breakpoint instructions exist in most processors, such as the "bkpt #immed8" instruction on the ARM architecture.

For each exception type, device manufacturers provide a default exception handler for processing that exception. ARM processors allow users to define their own exception handlers. Since several exceptions can happen simultaneously, device manufacturers assign a priority to each exception so that the processor can decide which exception is more important. Upon exiting the exception handler, the processor uses the link register (lr) to return the program counter (PC) to an appropriate place in the interrupted program. At this moment, device manufacturers must update the PC with an offset whose value depends on the specific exception type. For instance, the offset is four and zero when

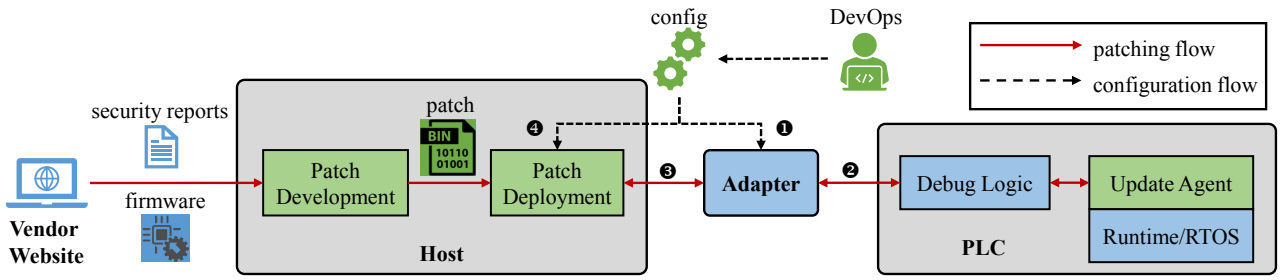


Figure 2. Overview of RLPatch's working flow.

the exception types are the `Prefetch Abort` and `Undefined Instruction`, respectively.

We find that the built-in three features mentioned above can significantly reduce the development complexity of our live patching strategies. Specifically, our patch insertion method relies on the debug channel to transfer update messages. Our update triggering method uses the `Prefetch Abort` and `Undefined Instruction` exceptions to trigger update actions. And we use the two corresponding custom exception handlers to redirect the control flow.

2.3 Threat Model and Assumptions

PLCs have four fault types: major non-recoverable fault (MNRF), major recoverable fault, minor fault, and I/O fault. The operators can directly clear the major recoverable faults and the minor faults by creating fault handlers and resetting errors separately at the control program scope level. Based on the solutions from the maintenance manual, operators can recover the PLCs from I/O faults at the peripheral hardware level. We focus on MNRFs that pertain to memory corruption vulnerabilities. Typically, they suffer from missing out-of-bound checks or accessing a memory location after the memory has been freed or deallocated. Note that MNRFs can only be fully patched at the firmware level. We assume that adversaries identify multiple MNRF vulnerabilities of the controller's firmware through various means, e.g., update notes analysis or firmware fuzzing.

The primary goal of the adversary is to manipulate the control program (ladder logic) and rush the controller into a crashed state. The MNRF vulnerabilities occupy about 24.15% of the total vulnerabilities that Rockwell reports. We use the vulnerability Lgx135333 [5] in the Rockwell 1756-L61 PLC as an example to explain the MNRF attack. Figure 1 shows a logic instruction Digital Alarm (ALMD) in a control program, and this instruction has a string-type controller tag called "System_Faults_Desc[0]". To force the PLC to enter an MNRF state, the adversary manipulates the tag's length parameter from 17 to -1 over the industrial network using engineering software. Our real-world examples in Section 4.3 will show that MNRF attacks have an enduring impact on the entire gas pipeline, not limited to the target controller itself. Note that the alarm logic instructions are not unique to Rockwell PLCs. Similar logic instructions exist in Siemens SIMATIC PLCs (e.g., `Program_Alarm` and `Get_Alarm`) and Schneider Modicon PLCs (e.g., `alarm`).

We make the following assumptions concerning the target controller. First, it has not been updated for years, but it still plays a crucial role in critical settings (e.g., gas pipeline systems and power industries). Second, we assume its processor supports a real-time debug logic unit and exception-handling mechanism. Third, we assume that manufacturers have issued binary patches encapsulated in the firmware updates and some extra related update notes. An update report is a set of version-change information released together with the updated firmware by vendors.

3 RLPatch Design

3.1 Overview

The design goal of RLPatch is to provide a general live patching framework for commodity industrial controllers. It works on ARM architecture-based controllers by leveraging the built-in exception handling mechanism without access to source code. Figure 2 presents the architecture of RLPatch, which has three parts: patch development, patch deployment, and the update agent. To minimize the overhead of RLPatch and its impact on industrial controllers, most of the functionalities including patch development and patch deployment, are implemented on the host side. The device side only executes the update agent.

The patch development takes the firmware updates and updated notes as inputs and outputs a set of binary patches. After that, it is worth briefly examining hardware-relevant configurations. When the adapter (protocol convertor) is properly configured in terms of the target processor specifications at step ❶, the DevOps engineer connects it to the controller board through a suitable debug access interface at step ❷. Then, DevOps installs and configures the corresponding driver on the patcher host before connecting the adapter to the host at step ❸. Once the patching channel is set up, DevOps initiates basic patch deployment commands and tests whether it works well at step ❹. Once the communication link is set up, the patch deployer works closely with

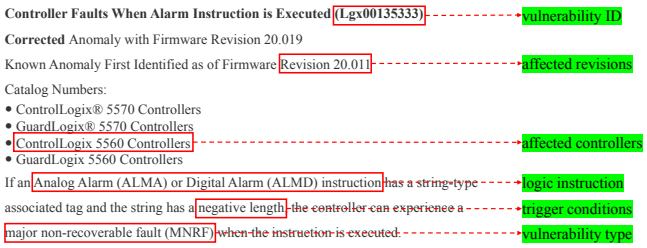


Figure 3. An MNRF vulnerability in official update specifications for Rockwell PLCs.

our update agent to repair critical MNRF vulnerabilities with the implanted hot patches.

3.2 Patch Development

The patch development of RLPatch has four goals: identifying vulnerability information, locating vulnerable code snippets in the firmware, constructing hot patches, and verifying binary patches.

3.2.1 Vulnerability Identification. In the vulnerability identification phase, we first download the update reports from the PLC's vendor website manually. The RLPatch focuses on MNRFs that can trigger memory corruption for PLCs, affecting the entire ICS. Common types of vulnerabilities that can trigger memory corruptions are buffer overflow and use-after-free. Figure 3 shows an official update note of an MNRF vulnerability (Lgx00135333) for Rockwell 1756-L61 PLCs. The update note reveals that this vulnerability (1) exists in the firmware versions ranging from 20.011 to 20.018 of the ControlLogix5560 family controllers; (2) it can be activated by specifying a negative length in the ALMA and ALMD logic instructions.

Then, we use the name entity recognition (NER) technique [13] to extract vulnerability information from the reports. Specifically, we define 13 vulnerability entities, including vulnerability ID, vulnerability type, affected revision, affected controller, logic instruction, and trigger condition. The trigger condition includes multiple sub-conditions such as constraint type and logic type, and the logic type has five sub-types including less than, equal, greater than, equal or greater than, and equal or less than. The output of the vulnerability identification is a series of vulnerability profiles. We manually create profiles for those vulnerabilities that miss necessary entities, such as logic instructions and trigger conditions. To extract the logic instructions for the vulnerability, we query the pertinent controller characteristics from the programming manual supplied by the PLC manufacturer. The trigger conditions for the vulnerability are obtained by focusing on the Arabic numerals and related programming logic keywords, such as 2069 and outside.

3.2.2 Vulnerability Localization. Figure 4 shows the vulnerability localization process of RLPatch. Before starting

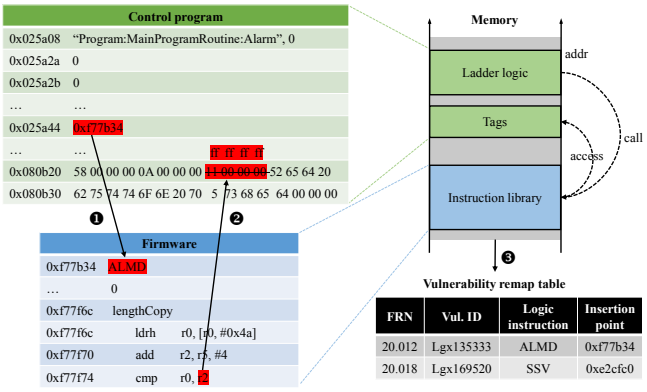


Figure 4. Vulnerability localization process of RLPatch. Step ❶: The ladder logic in the control program calls its library code in the PLC firmware. Step ❷: The library code accesses the associated tags of the ladder logic to get input parameters of the logic instruction. Step ❸: RLPatch outputs a vulnerability remapping table.

the controller, we insert vulnerable logic instructions into a control program with unique-value-tagged controller tags. For instance, we create an `ALMD` instruction with a string-type tag "Red button pushed", and the controller stores `0x11` as the default length of this string in the control program. We then use the built-in debug unit of the PLC to set watchpoints (memory addresses) for the controller tags and track which code snippet in the firmware accesses these tags. We set up two watchpoints in the PLC memory to track the interaction between the control program and the firmware. One watchpoint sits in the ladder logic area, and the other is in the associated tag area. For instance, one watchpoint monitors the logic instruction `ALMD` at address 0x025a44, and the other watchpoint monitors the length parameter of the `ALMD` instruction at address 0x080b2c. To locate the vulnerable library code of the `ALMD` in PLC firmware, we use engineering software to tamper the length parameter of the tag from `0x11` to `0xffffffff`.

After the processor runs into the scan cycle of the firmware, we observe that the control logic program frequently calls a set of firmware's built-in functions through a jump table. The built-in functions are known as the controller's logic instruction library. The jump table has a reference to the instruction library, enabling us to locate all vulnerable logic instructions. As shown in Figure 4, a caller subroutine (alarm instruction `ALMD`) in the control program calls a callee subroutine in the firmware to execute its library code starting from 0xf77b34. During the execution of the library code, we observe that the callee subroutine accesses an input buffer in which the contents are copied from the associated tags. Specifically, the library code of the `ALMD` instruction uses register `r2` to access the length of the associated string. At this moment, memory corruption occurs since PLC writes

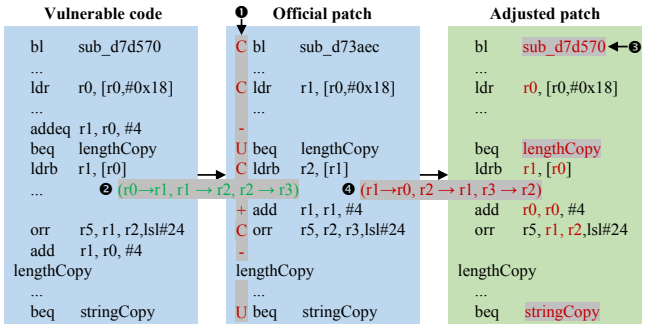


Figure 5. Patch generation process of RLPatch. Step ❶: Compare the vulnerable code and the official patch. Step ❷: Record the state changes of a processor between the vulnerable code and the official patch. Step ❸: Replace the absolute address in the official patch with the matching address in the vulnerable code. ❹: Restore the processor state in the official patch with the matching registers in the vulnerable code.

data beyond the boundaries of a memory allocation (at most 82 characters), overwriting adjacent memory regions.

Finally, we use a vulnerability remap table to record the results of the vulnerability localization. The table contains a set of vulnerable logic instructions of the PLC firmware, and each vulnerability has four items: firmware revision number (FRN), vulnerability ID, involved logic instruction, and insertion point (entry address of the vulnerable instruction).

3.2.3 Patch Generation. Figure 5 shows the patch generation process of RLPatch. To learn what has changed between the vulnerable code and the official patch, we use the binary diffing tool BinDiff [50] to compare them at the function granularity. The comparison results are grouped into four categories by using a support vector machine classifier: added, removed, unchanged, and changed. The added type means that an assembly instruction exists in a patch subroutine but does not exist in its vulnerable subroutine. The removed type is that an assembly instruction exists in a vulnerable subroutine but does not exist in its patch subroutine. If the opcode and the operands are identical between the vulnerable and the patch subroutines, we call them unchanged instructions. We refer to an assembly instruction with the same opcode but a different operand as a changed instruction. Meanwhile, we record the state changes of a processor between the vulnerable code and the official patch, such as the sequence *(r0→r1, r1→r2, r2→r3)*.

To enable the official patch to be placed in any memory location, we adjust the patch to make it suitable for our live patching strategy. There are two cases for the changed instruction: the changed operand wraps absolute address(es) and the changed operand only involves processor registers. In the first case, we replace the address in the official patch with the matching address in the vulnerable code, such as the address *sub_d7d570* marked red in Figure 5. In the second case, we restore the processor state in the official patch with the matching registers in the vulnerable code, such as the sequence *(r1→r0, r2→r1, r3→r2)*. Naturally, the unchanged instructions do not need to be checked. The instructions that are added and removed are essentially patch functions, which should be kept for repairs. To be safe, RLPatch aborts the updates if it detects that the hot patches use different compiler versions.

3.2.4 Binary Patch Verification & Validation. To ensure the patch application is correct, we manually scrutinize the generated executable patches offline. Before the patch deployment, we check whether an official patch can be converted into a hot patch. First, the patch behaviors cannot contain data structure and code logic restructuring changes commonly involved in the function upgrading systems. Second, the patch behaviors cannot contain the update agent and the scan cycle procedures because setting such addresses can cause a recursive trap [22]. RLPatch can identify three vulnerability types: buffer overflow, use-after-free, and function bugs that can be directly observed. RLPatch does not support non-MNRF, such as functional bugs whose operations cannot be directly observed. We verify the patch by checking each field of the binary patch while setting other fields with their original input values. For the integer fields, we use their minimum and maximum values and some out-of-bound random values to check each integer. For the character string fields, we use some random non-zero values to check each byte. For the array fields, we use some random non-zero values to check each element.

To determine if the patch is safe, we also perform validation in PLC's runtime. We first write a control program and use it to measure the timing of the full live patching process after the patch deployment. The light control program incorporates 140 logic instructions that are commonly used in real-world PLC programming. The program controls five bulbs in a field device to flash at different frequencies, and the field device has five inputs and five outputs. Specifically, a blue light flashes every 1,000 milliseconds, a green light flashes every 2,000 milliseconds, a yellow light flashes every 4,000 milliseconds, a red light flashes every 8,000 milliseconds, and a white light flashes every 16,000 milliseconds. Thus, we utilize up to 32 permutation inputs to conduct a performance evaluation. For example, an array *(0,1,1,1,1)* indicates that all inputs are triggered except the first one. We remove those patches from the candidate list if they cause the controller to crash. Second, we test the buggy functions of PLC logic instructions by observing the outputs of the PLC under the given inputs. We remove those patches if they cause the PLC crash or function failure. For instance, the firmware version 20.012 of the ControlLogix 1756-L61 has a function bug Lgx136317; this bug exists in the Motion Axis Jog (MAJ) instruction and it can make the axis move in the wrong direction. We remove this patch if the patched

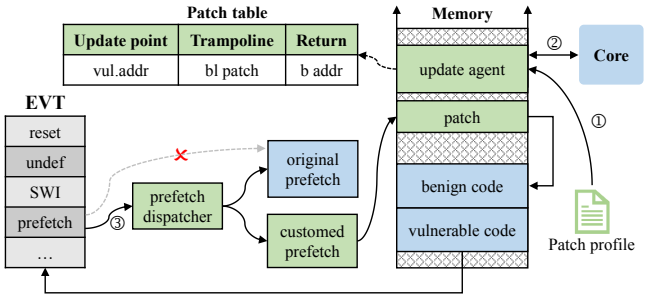


Figure 6. Exception-based hot-patching process. When the processor reaches the entry point of the vulnerable code, it throws an exception (e.g., Prefetch Abort) and turns to execute our customized exception-handling program. Our program redirects the control flow from the vulnerable code to patches corresponding to the vulnerability. After finishing the patches, the processor returns to execute the benign code right after the vulnerable code.

MAJ instruction cannot move the axis in the right direction. Third, we ensure that the patched systems are equivalent to updated systems. More specifically, we perform differential testing to compare the outputs of the patched system and the updated system. This involves executing identical inputs and states on both versions and analyzing their respective outputs to identify any discrepancies.

3.3 Live Patching Strategy

The agent's design utilizes three built-in components to achieve live patching for PLCs with hard time restraints. As shown in Figure 6, they are the debug port that transmits the repair information from the patch deployer (①), the exception generation mechanism that triggers update operations (②), and the exception vector table (EVT) that shares the exceptions between the unmodified runtime system and our update agent (③).

3.3.1 Patch Insertion. Most processors contain a debug port providing a connection between outside the SoC and a program on the SoC. The patch deployer uses the real-time debug interface to transfer the patch profile to our update agent. We lower the priority of the transmission routine so that it cannot interrupt the normal execution of other tasks. To reduce the interrupt latency, we use the debug channel in a polled mode to transfer the repair information asynchronously. Further, this channel helps minimize the impact on execution time and memory usage.

The patch deployer polls the control register of the debug channel. It writes the repair information of the patch profile into the debug channel for the update agent to read. At a safe update point, the update agent polls the control register to read repair information through a coprocessor and write them into a dedicated memory area. The agent manages a patch table that records all update points, trampolines, and return locations. Once receiving the binary patches, the agent writes them into the patch table. Each update point specifies an entry point of the buggy code, each trampoline points to a piece of patch code of the buggy code, and each return location points to a memory address right after the corresponding buggy code. To avoid affecting the controller runtime during the patch insertion, we choose the idle time between the watchdog time and the scan time as a safe update point to insert the patch information. Specifically, based on the baseline time of each read or write operation, the agent receives and inserts a fixed number of words of the patch information into the memory at the rear of each scan cycle.

3.3.2 Triggering Update Operations. We use the built-in exception-handling mechanism of the processor to trigger update operations. The update agent uses the entry point of the buggy code to program breakpoint registers through the memory-mapped advanced peripheral bus (APB) interface. Once the basic configuration is completed, the debug unit continuously monitors the values currently appearing on the address bus. Once the patch insertion condition `core.addr` $\in \{i_1, i_2, ..., i_n\}$ is present, that is, the breakpoint address enters the execution stage of the instruction pipeline, which causes the processor to halt temporarily and the debug component generates a `Prefetch Abort` exception.

Our update agent can also employ software breakpoints to trigger update operations. A breakpoint instruction is architecture dependent, such as the "bkpt #immed8" instruction on ARM, "int $n" instruction on x86, and "break n" instruction on MIPS. On the ARM architecture, the agent writes a breakpoint instruction over the first instruction of the buggy code. In the ARM mode, it is an `UND` opcode (e,g. 0xfedeffe7) occupying four bytes of the specified instruction. In the Thumb mode, it is the `bkpt` (e.g., 0xbebe) occupying two bytes of the target instruction. The ARM core causes an exception whenever execution reaches the instruction on which the agent sets breakpoints. When this exception happens, the control is transferred to the agent, and then the agent uses our exception handler to conduct update operations.

The specific trigger method that the agent uses depends on the idle time of the controller's one scan cycle. The agent will choose the software breakpoint when it detects that the idle time exceeds a predetermined threshold value. This is because by default the debug logic can only have 2 to 8 comparators, which limits its ability to patch multiple vulnerabilities within one scan cycle. Otherwise, the agent chooses the hardware breakpoint because hardware breakpoints have dedicated registers and thus incur less overhead than software breakpoints. The update agent can trap most locations of the firmware code except itself because setting such addresses can cause a recursive trap. The agent manages such addresses as a denylist. It checks the given addresses against the list and rejects setting them if they exist in the denylist.

Algorithm 1: Control flow redirection

```
1  exception_type ← get exception type;
2  update_point ← get update points;
3  Save_Context ();
4  Set_Mode (exception_type);
5  trampoline ← T_Search (update_point, patch_table);
6  Replace (update_point, trampoline);
7  patch_code ← Get_Patch (trampoline);
8  while patch_code ≠ ∅ do
9  |   execute next instruction of patch_code;
10 end
11 return_location ← R_Search (update_point, patch_table);
12 LR ← return_location;
13 Recovery_Context ();
```

3.3.3 Update Handlers. To achieve update operations, we design a custom update handler for the `Prefetch Abort` and `Undefined Instruction` exceptions, separately. Algorithm 1 depicts the control flow redirection of the update handlers. Each handler first saves the current core context and switches to the appropriate processor mode. Here we take the `Prefetch Abort` handler as an illustrative example. The handler first saves the current core context and switches to `ABT` mode. Then, the handler utilizes the recorded value of the PC, i.e., the entry point of the buggy code, to look up the trampoline from the patch table (line 5). Finally, the handler replaces the fetched instruction with the retrieved trampoline by changing the PC value, which redirects the control flow to an appropriate patch code (line 6). When the patch code completes its execution, the handler returns to the next instruction right after the buggy code (lines 11-12). The main difference between the `Undefined Instruction` and `Prefetch Abort` handlers is that the former works in the `UND` mode while the latter works in the `ABT` mode.

To avoid exception-sharing violations on the update agent and the runtime system, we provide an exception dispatcher for each of the two exceptions. If the exception belongs to the runtime application, the core will execute the exception handler for the runtime itself; otherwise, it will execute the agent handler. Furthermore, we give priority to the original runtime exceptions over agent exceptions. Therefore, the agent handler can be called only after all runtime exceptions have been serviced.

3.4 Integrating Agent into Controller Runtime

We propose an integration method to make our update agent co-work with the original runtime of the controller. Though industrial controllers cover a wide range of applications (from elevator operation to gas pipeline), their operation flows are wrapped in three essential firmware components: boot loader, executive loader, and function firmware.

Boot loader performs processor and peripherals initialization, which covers an exception vector table and several hardware monitors. The EVT defines a group of consecutive exceptions and each exception has the form "ldr pc, [pc,#xxx]," the xxx means the entry point of the exception handler relative to the current program counter. Therefore, we use this form to search for exception handlers in the disassembled boot loader. After finding the EVT, we install our custom update handlers on it through the exception-sharing method mentioned above. We refer to the hardware monitors as the addresses in the firmware for monitoring the hardware locations. The hardware monitors can be discovered by searching for immediate values, and their specific meanings can be inferred by analyzing the procedures after the immediate values.

Executive loader contains a validation algorithm to confirm that the newly uploaded firmware is not corrupted before loading it. We use dynamic analysis techniques to understand the validation process in the executive loader. Cyclic redundancy check (CRC) and modular summation (MSUM) algorithms incorporate the XOR and accumulation operations, respectively. Therefore, we use the keywords `eor` and `add/adc` to search for all potential CRC and MSUM algorithms in the assembly code. Both the CRC and the MSUM algorithms incorporate the loop structure, so we use the expressions `cmp Rx,Ry` and `B(cond) label` to search them. The `Rx` and `Ry` represent two different processor registers, the `cond` represents a processor condition code, and the `label` represents the start or end position of the loop structure. If the firmware adopts cryptography functions, we use the JTAG interface to implant our RLPatch into the controller [19]. After comprehending the validation algorithm, we use Schuett's method [37] to uncover the header structure and recalculate its checksum data of the firmware.

We examine real-time scheduler and scan cycle parts inside the function firmware to integrate four functions of the update agent: stack setup, agent initialization, patch insertion, and patch update. The agent requires a stack space for two processor modes: `UND` and `ABT`. So we initialize the stack pointer for each processor mode with a different size of words. To fix all potential firmware-level vulnerabilities, we install the update agent within the startup sequence of the function firmware, followed by initializing the state machine of the agent to start the communication link. Our update point ensures that PLC always executes the same code version when processing shared arguments such as I/O values. To locate the scan cycle, we debug the processor step-by-step until the controller enters an uninterruptible state. At this moment, we obtain the mapping interval of the peripherals in the memory by querying the memory mapping table from the processor specifications, and then use the target mapping address in memory-mapped I/O (MMIO) operations to identify the scan cycle. Based on the MMIO ports in the memory mapping table of the processor, we use the representation `ldr Rx,=addr` to locate the input and output operations, where `addr` is any memory address in the MMIO ports.

Table 2. Defensive effectiveness of hot patches for the corresponding vulnerabilities in different firmware revisions of three Rockwell PLCs.

Device	FRN	Vul. ID	Vul. type	Before patch	After patch
ControlLogix 1756-L61	19.015	IN25781	Buffer overflow	PLC crash	Safe
	20.012	Lgx135333	Buffer overflow	PLC crash	Safe
	20.012	Lgx136317	function bug	PLC function failure	Safe
	20.013	Lgx150458	Function bug	PLC function failure	Safe
	20.013	Lgx179778	Function bug	PLC function failure	Safe
	20.014	Lgx131221	Function bug	PLC crash	Safe
	20.014	Lgx168703	Function bug	PLC crash	Safe
	20.018	Lgx135333	Do not exist	Not exploited	Normal
	20.018	Lgx140975	Function bug	PLC crash	Safe
	20.018	Lgx169520	Buffer overflow	PLC function failure	Safe
CompactLogix 1769-L18ER-BB1B	20.012	Lgx136059	Function bug	PLC crash	Safe
	20.012	Lgx149169	Function bug	PLC crash	Safe
	20.019	Lgx138238	Function bug	PLC crash	Safe
	20.019	Lgx220553	Function bug	PLC crash	Safe
	20.019	Lgx223326	Buffer overflow	PLC crash	Safe
	20.019	Lgx227157	Buffer overflow	PLC crash	Safe
	20.019	Lgx227905	Use-after-free	PLC crash	Safe
CompactLogix 1769-L24ER-QB1B	20.012	Lgx200735	Function bug	PLC crash	Safe
	20.013	Lgx200734	Function bug	PLC crash	Safe
	20.014	Lgx228810	Use-after-free	PLC crash	Safe
	20.018	Lgx219873	Function bug	PLC crash	Safe

4 Evaluation

We conducted extensive experiments to validate the efficiency of the RLPatch and hot patches. On one hand, we evaluated the effectiveness of RLPatch on three individual PLCs, in terms of resource consumption and reliability. On the other hand, we evaluated the practical application of RLPatch in a real-world natural gas pipeline. In the pipeline, we implemented three attack scenarios to demonstrate how a single controller affects the gas delivery productivity and how RLPatch mitigates these adverse effects without impacting the pipeline.

4.1 Preparation

To uncover the internal processor information and the update access port of real-world industrial controllers, we reverse-engineered three off-the-shelf PLCs: ControlLogix 1756-L61, CompactLogix 1769-L18ER-BB1B, and CompactLogix 1769-L24ER-QB1B. It took us about four months to understand the first controller, but subsequent controllers took only one day each. The consistent design and functionality across PLCs significantly reduce the analysis time required. To ensure that PLCs can work reliably in an intense electromagnetic environment, their debug port signals must have default HIGH and LOW states. Therefore, we analyzed the physical laws of the JTAG circuit and used the pull-up and pull-down resistors to identify the JTAG ports of the three PLCs.

We disassembled 13 PLC firmware images to uncover their control flow. It took us two hours to understand the three components of every firmware image. Specifically, we located the exception handlers of the boot loaders stored in the shipped three PLCs. We also figured out the firmware validation algorithms in the executive loaders. In addition, we located the input and output operations of the scan cycle in the function firmware images.

We chose these PLCs to build an online update channel for the following experiments. Specifically, we selected a Segger J-Link adapter and used it to connect the patching host and the three PLCs. Based on their processors' datasheets, we wrote three configuration scripts and used them to instantiate the corresponding PLCs. Once the patch profile is transferred to the update agent through the update channel, RLPatch starts to automatically execute the live patching tasks. Note that the hot patches are added to the PLC memory at runtime, while the update handler is integrated into the PLC firmware offline. The three steps, reverse engineering of the hardware and software architectures of PLCs and the construction of the update channel, are a one-time effort.

4.2 Individual PLC Case Study

4.2.1 Experimental Design. To exclude the influence of adverse factors like network traffic in the complex ICS, we utilized the light control program described in Section 3.2 to evaluate RLPatch in an individual PLC testing environment. The program has real-time requirements that automatically control bulbs with different colors in the field device to flash at different frequencies. The physical facilities incorporate an engineering workstation running Windows 7, a

Table 3. Vulnerabilities and patches for three Rockwell PLCs.

Device	FRN	Vul. ID	Logic instr.	SoP
ControlLogix 1756-L61	20.018	Lgx131221	MSG	0x180
		Lgx138238	GSV/SSV	0x44
		Lgx136317	MAJ	0x9F0
		Lgx140975	SQI/SQO/SQL	0x1418
		Lgx150458	MAJ	0x160
		Lgx169520	GSV/SSV	0x3D0
	20.014	Lgx138238	GSV/SSV	0x44
		Lgx136317	MAJ	0x9F0
		Lgx150458	MAJ	0x160
		Lgx152701	GSV/SSV	0x1B4
		Lgx179778	GSV/SSV	0x1B4
		Lgx131221	MSG	0x180
		Lgx168703	MSG	0x60
	20.013	Lgx138238	GSV/SSV	0x44
		Lgx136317	MAJ	0x9F0
		Lgx150458	MAJ	0x160
		Lgx152701	GSV/SSV	0x1B4
		Lgx179778	GSV/SSV	0x1B4
		Lgx131221	MSG	0x180
		Lgx168703	MSG	0x60
		Lgx152820	GSV/SSV	0xC8
	20.012	Lgx169520	GSV/SSV	0x3D0
		Lgx135333	ALMD/ALMA	0x4D4
	19.015	Lgx169520	GSV/SSV	0x3D0
		IN25781	ALMD/ALMA	0x4C8
	17.004	Lgx169520	GSV/SSV	0x3D0
	16.023	Lgx169520	GSV/SSV	0x3D0
CompactLogix 1769-L18ER-BB1B	20.019	Lgx138238	GSV/SSV	0x44
		Lgx220553	GSV/SSV	0x1EC
		Lgx227905	MOV	0x6D4
		Lgx223326	EQU/.../NEQ	0xF8
		Lgx227157	SRT	0x6BC
	20.012	Lgx149169	UID/UIE	0x38
		Lgx227905	MOV	0x6D4
		Lgx223326	EQU/.../NEQ	0xF8
		Lgx227157	SRT	0x6BC
		Lgx136059	IOT	0x280
		Lgx131221	MSG	0x180
		Lgx135333	ALMD/ALM	0x4D4
CompactLogix 1769-L24ER-QB1B	20.018 20.014 20.013 20.012	Lgx220553	GSV/SSV	0x1EC
		Lgx219873	GSV/SSV	0x17C
		Lgx227905	MOV	0x6D4
		Lgx228810	MOV	0x15C
		Lgx200734	GSV/SSV	0x108
		Lgx200735	GSV/SSV	0x178
		Lgx227157	SRT	0x6BC

Segger J-Link adapter, an I/O field device, and three Rockwell PLCs. The industrial software used in the experiment includes RSLogix 5000/Studio 5000, ControlFlash V15.03, and Ozone V3.22. Specifically, the RSLogix 5000 is used to design and configure control logic programs. The ControlFlash is used to change the firmware revision on a controller, and the Ozone is used to dump the runtime contents of the controller memory. The communication module collects the most recent execution time of the controller module. The input and output modules provide 16 input points and 16 output points for the controller module, respectively.

We use Dunlap's time collection tool [14] to measure the execution time. The tool uses the Common Industrial Protocol (CIP) to poll the controller every 250ms, which can automatically request the most recent execution time of our control application. Since the memory of the Rockwell controller is divided into several areas, we manually measured the memory usage of the RLPatch and hot patches. Specifically, we calculated the memory size occupied by the RLPatch and hot patches by dumping the flash memory and dynamic RAM from the running memory.

4.2.2 Correctness Evaluation. We performed a correctness evaluation of the vulnerability detection phase of RLPatch by using 13 firmware versions collected from the Rockwell website. We first analyzed 13 update reports of the three Rockwell PLCs and found 68 MNRF vulnerabilities. Then, we employed 20 logic instructions on the MNRF vulnerabilities to activate these vulnerabilities, leading to the successful identification of 67 vulnerabilities. Finally, we implemented the localization step to identify these vulnerabilities in the operational memory, identifying 67 locations of vulnerabilities. In the experiments, the discovered vulnerabilities have no false positives and only one false negative case. Release notes indicate that version 20.018 of the firmware has a bug Lgx135333 (Table 2), while our testing shows that it has actually been fixed. These results demonstrate the precision of our vulnerability detection method, encompassing both vulnerability identification and localization.

After identifying the vulnerable library code, we employed the patch development approach to produce 67 hot patches. Table 3 provides a comprehensive overview of the 67 vulnerabilities and their corresponding hot patches. Note that a single vulnerability may exist in multiple firmware revisions. The last column presents the size of patch (SoP) for all logic instructions. For instance, the first item in Table 3 reveals that the logic instruction `MSG` in the firmware revision 20.018 of the ControlLogix 1756-L61 controller has a vulnerability Lgx131221, and the patch size for this vulnerability is 0x180 (384 bytes). To assess the defensive efficacy of RLPatch and the generated patches, we conducted 20 proof-of-concept (POC) attacks on three fixed PLCs. The results presented in Table 2 demonstrate that RLPatch effectively fixes all 20 distinct vulnerabilities without causing any conflicts with coexisting applications on the PLCs. Additionally, we verified that the applied patches can be uninstalled without affecting the operations of the PLCs.

4.2.3 Runtime Evaluation. To evaluate the performance of RLPatch on runtime, we measured the time that the firmware takes to execute one iteration of the control application on three Rockwell PLCs. Note that both the update agent and the patch can affect the scan time of RLPatch, and their effects cannot be measured separately. We repeated ten executions of each firmware, and each execution started 60

Table 4. The mean runtime overheads for three PLCs executing different firmware versions. Columns are the baseline before repairing vulnerabilities, hot-patching after applying the hardware breakpoint (HWB) based and the software breakpoint (SWB) based RLPatch, and percent increase relative to the baseline scan time.

Device	FRN	Vul. ID	Scan time (μs)		
			Baseline	Increase	
				HWB	SWB
ControlLogix 1756-L61	20.018	Lgx169520 Lgx140975	298.64	0.56%	3.23%
	20.014	Lgx131221 Lgx168703	238.42	0.64%	3.45%
	20.013	Lgx150458 Lgx179778	294.09	0.54%	2.72%
	20.012	Lgx135333 Lgx136317	245.36	0.60%	3.13%
	19.015	IN25781	143.89	0.54%	4.63%
	17.004	Lgx169520	204.74	0.41%	3.16%
	16.023	Lgx169520	187.45	0.43%	3.87%
CompactLogix 1769-L18ER-BB1B	20.019	Lgx227905 Lgx227157	263.81	0.64%	4.51%
	20.012	Lgx131221 Lgx135333	251.38	0.63%	4.70%
CompactLogix 1769-L24ER-QB1B	20.018	Lgx220553 Lgx219873	270.82	**0.69%**	**4.73%**
	20.014	Lgx227905 Lgx228810	258.65	0.61%	4.55%
	20.013	Lgx200734 Lgx200735	253.25	0.59%	4.62%
	20.012	Lgx227157	225.17	0.32%	2.73%

seconds after the controller reached a steady state of operation. Table 4 shows changes in scan time overheads before and after applying RLPatch.

The hot-patching increase part of Table 4 shows the average runtime overhead for the controller executing different firmware versions. We first compared the difference in average runtime increase for repairing each vulnerability using the HWB-based and the SWB-based protection mechanisms. The results are between 0.73-0.93μs for HWB-based protection and between 3.85-7.25μs for SWB-based protection. In the worst case, it takes approximately 75 (0.93μs · 80MHz) and 508 (7.25μs · 70MHz) clock cycles for HWB-based RLPatch and SWB-based RLPatch to fix the vulnerability Lgx220553/Lgx169520, respectively. Then, we used the HWB-based and the SWB-based protection mechanisms to measure the average scan time increase for repairing each of the seven firmware revisions. We compared their repair performance of two vulnerabilities at most because there are only two hardware breakpoints in the processor of ControlLogix 1756-L61 PLC. The results show that the SWB-based protection mechanism is about 5-10 times slower than the HWB-based mechanism. Finally, we only used the SWB-based RLPatch to measure the average scan time increase for repairing the firmware revisions with more than two vulnerabilities. In the worst case, it takes about 44.80μs for RLPatch to repair seven vulnerabilities in the FRN 20.018 of the CompactLogix 1769-L24ER-QB1B PLC. Note that HERA cannot patch all vulnerabilities of the three PLCs since they come with up to only six comparators for hardware breakpoints.

If the scan time of the control application is greater than the length of the watchdog time of the controller, the controller would report a major fault and fall into the MNRF state. In three Rockwell PLCs, their watchdog time is 10ms, and its maximum scan time under our control application is less than 500μs. Therefore, the watchdog time does not limit the number of hot patches for our RLPatch.

Table 5. Increase in memory usage for distinct memory of the Rockwell 1756-L61 controller while applying RLPatch.

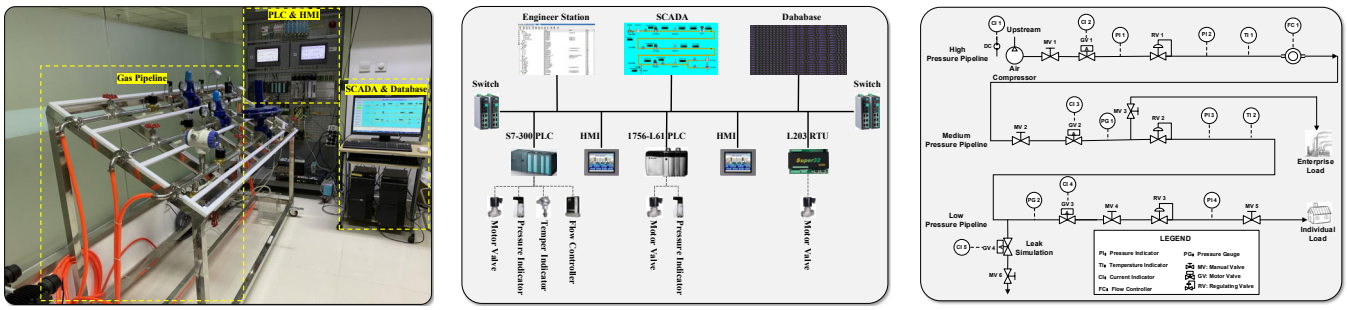
Device	Free Memory	Size	Increase
ControlLogix 1756-L61	Flash Memory	3,782	**0.05%**
	Dynamic RAM	128	**7.15%**
CompactLogix 1769-L18ER-BB1B	Flash Memory	$\geq 10^6$	$\leq$ 0.0002%
	Dynamic RAM	8,192	0.07%
CompactLogix 1769-L24ER-QB1B	Flash Memory	$\geq 10^6$	$\leq$ 0.0002%
	Dynamic RAM	8,192	0.06%

Table 6. Jitters during normal system activity (baseline) and system update (hot-patching) for three Rockwell PLCs.

Device	Criteria	Baseline	Hot-patching
ControlLogix 1756-L61	min	0	0
	avg	0.05%	0.05%
	max	3.14%	4.45%
CompactLogix 1769-L18ER-BB1B	min	0	0
	avg	0.04%	0.04%
	max	4.26%	5.25%
CompactLogix 1769-L24ER-QB1B	min	0	0
	avg	0.03%	0.03%
	max	**4.50%**	**5.29%**

4.2.4 Memory Usage Evaluation. The Rockwell PLC has four types of memory: flash memory, I/O memory, static RAM, and dynamic RAM. The flash memory stores the bootloader and firmware. The I/O memory includes the I/O tags and communications with the workstation. The static RAM stores all the logic routines of the control application. The dynamic RAM executes the firmware and the control application. Our RLPatch only uses the flash memory and dynamic RAM, but the user memory (I/O memory and static RAM) is not affected during the live patching. Of the three controllers, the 1756-L61 PLC has the least memory resources, including 8MB flash memory, 494KB I/O memory, 2MB static RAM, and 24MB dynamic RAM.

Although the original functionalities of the Rockwell PLCs have occupied most parts of the memory, non-trivial parts are never used. Table 5 shows the changes in their memory usage. The FRN 20.014 is the largest firmware in the 1756-L61 PLC, which takes up 2,746 KB of flash memory. Therefore, the flash memory has 3,782 KB of free memory to



(a) Test-bed **(b)** System topology **(c)** Operation flow diagram

Figure 7. The evaluation gas pipeline test-bed.

hold our update agent, excluding the 128KB and 1,536 KB for storing the bootloader and base firmware, respectively. In this experiment, our update agent only occupies about 0.05% (2KB/3,782KB) of the free flash memory. We analyzed the memory contents of the dynamic RAM when the 1756-L61 PLC reached a steady state. The results show a 128KB of memory is free for our hot patches. The examination was conducted by setting watchpoints within their addresses during our online analysis. In the worst case, our six hot patches only occupy about 7.15% (9,372 bytes/128KB) of the free dynamic RAM. Obviously, the spare memory size of the controller cannot limit the number of patches of RLPatch.

4.2.5 Reliability Evaluation. To evaluate the reliability of RLPatch, we measured the jitter of three PLCs (deviation of the actual scan time from the ideal scan time). The watchdog timers of three PLCs are 10ms. The most typical causes for jitter are peripheral interrupts and network connections. We first measured the system jitter during the normal execution of the light control program without updates, and then we measured the jitter while hot-patching.

Table 6 lists all jitter results of three PLCs during their normal system activities (baseline) and hot-patching. The results show that most jitters of these PLCs are zero in baseline and hot-patching, their average jitters are the same, and their maximum jitters are in the same order of magnitude. In the worst case, RLPatch incurs a maximum jitter of around 5.29% (529μs/10ms) in the CompactLogix 1769-L24ER-QB1B PLC. Therefore, RLPatch is reliable because it does not introduce any bumps to system activities.

4.3 Real-world Gas Pipeline Case Study

4.3.1 Test-bed. The test-bed, as shown in Figure 7a, includes three parts: the gas pipeline, PLCs & human-machine interfaces (HMIs), and the supervisory control and data acquisition (SCADA) server & database. The gas pipeline delivers medium-pressure gas to the enterprise load and low-pressure gas to the individual load. PLCs collect data (e.g., pressure and temperature) from sensors, convert analog/digital signals, and send them to the SCADA server, and control valves based on the control program. HMIs enable the management of the system state for the operators through their connection to the PLC. The SCADA server monitors the gas pipeline, acquires and stores sensor values in the database, and controls the PLCs and HMIs. Figure 7b shows the system topology of the test-bed. The monitor/control nodes are connected through industrial switches, and sensor/actuator nodes are connected to the PLCs/RTU through power transmission lines. The S7-300 PLC and L203 RTU control the high and medium pressure pipelines, and the 1756-L61 PLC controls the low-pressure pipeline.

Figure 7c shows the operation flow of the gas pipeline. The safety of gas pipelines, due to natural gas's flammability, requires precise design and operation for efficient transportation and effective leak detection. We employed an air compressor to simulate gas compression for achieving efficient gas transport. The air compressor shuts off if the tank's pressure reaches 800KPa, and it turns on again and re-pressurizes when tank pressure reaches its lower limit. In addition, three regulating valves adjust the pressure of the high pipeline, medium pipeline, and low pipeline to the range of 400-700KPa, 200-300KPa, and 2-10KPa, respectively. The S7-300 PLC measures the gas pressures of the high and medium pressure pipelines, the gas temperatures of the high and medium pipelines, and the gas flow of the high pipeline. Also, it is used to control the motor valves of high, medium, and low pipelines. The L203 RTU collects two feedback signals: the inlet pressure of the low pipeline and the state of the leakage valve. The 1756-L61 Programmable Logic Controller (PLC) measures the inlet and outlet pressures of the low pipeline while also controlling the motor valve associated with that pipeline. Note that we will use the 1756-L61 PLC to implement live patching.

4.3.2 Attack Scenarios. We created three attack scenarios on the unpatched 1756-L61 PLC based on the first three vulnerabilities in Table 2. We used the first attack scenario as an example to demonstrate its impact on the entire gas pipeline. Figure 8 shows the pressure state at the different places of the gas pipeline during the attack. Once the 1756-L61 PLC ran

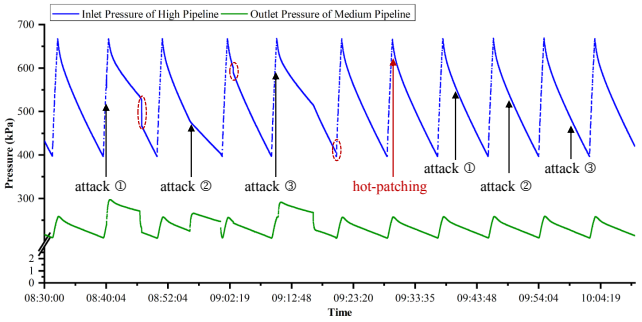


Figure 8. Actual pressure measurements in three attack scenarios and the hot-patching process.

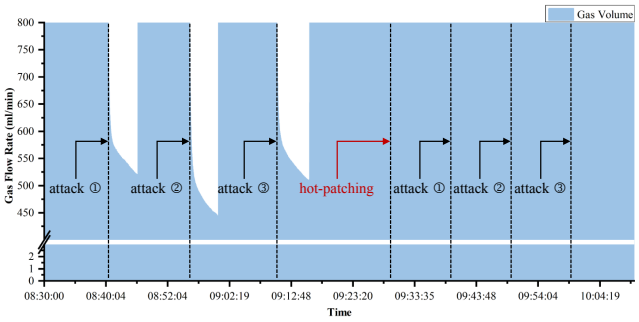


Figure 9. Actual gas flow measurements in three attack scenarios and the hot-patching process.

into the MNRF state at 8:40:08 AM, the SCADA server lost its connection to the 1756-L61 controller and could not gain the inlet and outlet pressures of the low pipeline anymore. As a result, the server continued to insert the same value into the database, i.e., the inlet pressure remained at 204.828 KPa, and the outlet pressure at 2.555 KPa. In this attack scenario, the S7-300 PLC and the L203 RTU still worked in progress, but their measurements, such as pressures, were affected, although not being directly attacked. The SCADA server issued alerts for the operators after multiple re-connection attempts failed. The dashed red cycle in Figure 8 shows that the inlet pressure of the high pipeline plummeted from 526.620 KPa to 467.593 KPa because the pipeline state from 8:45:50 to 8:47:43 was missing during the shutdown of the SCADA server. The other two attacks produced similar effects on the pipeline.

As shown in Figure 9, the gas flow state of the high pipeline monitored by the S7-300 PLC was also affected. Once the 1756-L61 PLC entered the MNRF state, the gas flow rate behaved as follows. About 30 seconds later, the flow rate began to drop rapidly. After 60 seconds, the descent of the gas flow rate became slower. The reason for this abnormal change is that the motor valve of the low pipeline closes automatically. As the attack cuts off part of the gas supply, the residents face a gas shortage, which may cause social anxiety. Intuitively, the longer the motor valve of the low pipeline is closed, the more the gas flow rate drops. However, we found that the gas flow rate decreased even more (below 500 ml/min) during the second attack scenario in which the closure of the motor valve lasted the shortest amount of time (5 minutes and 7 seconds) among the three. This is because the second attack scenario occurred when the air compressor was off, while the other two attack scenarios happened when the compressor was releasing gas. This finding shows that the timing of the attack has a far greater impact on the gas delivery process than what the motor valve does.

In summary, it took 25 minutes for the operators to eliminate the negative effects of the three attacks. Within 15 minutes and 23 seconds, the productivity of gas delivery was reduced by about 33.42% (240,907.64 *ml*) in total.

4.3.3 Live Patching Results. We used RLPatch to repair the first three vulnerabilities mentioned in Table 2 at 9:30:00 AM. During the live patching, the 1756-L61 controller worked well, and it did not lose connections to other controllers and the SCADA server. Moreover, all data, such as the pressure state, temperature state, and gas transmission state provided by SCADA, did not exceed their safety margins. To validate the defensive effectiveness of RLPatch, we launched the same three attacks (attack ①, attack ②, and attack ③) on the 1756-L61 controller, and the attacks occurred every 10 minutes interval. The results show that the primary business goal of the gas pipeline, gas transmission volume, was not affected by RLPatch. Comparing the productivity efficiency of the gas pipeline before and after hot-patching, we can see that RLPatch has the capability of fully shielding the 33.42% gas volume loss caused by three attacks.

5 Discussions and Future Extensions

The issue of timely ICS security patching is a serious cross-sector issue. We have provided methods for vulnerability localization, patch development, live patching, and self-assessment tools that can be incorporated into existing ICS patch management processes.

Patches beyond MNRF vulnerability. We used taint analysis techniques to identify MNRF vulnerabilities but only generated simple patches with check code insertion. In our evaluation, we did not observe more complex patches in terms of MNRF vulnerabilities. This is because manufacturers avoid causing any new vulnerabilities when releasing new firmware versions. Furthermore, we will explore a technique to disable those vulnerabilities without corresponding official fixes, even though addressing this problem requires the development of original binary patches.

Applications beyond outdated PLCs. To support outdated PLCs, we need to reverse-engineer the debug interface and PLC firmware, which violates manufacturers' terms of service, and thus invalidates warranty contracts. On the other hand, our current RLPatch leverages hardware debug capabilities to deploy patches for outdated PLCs, while exposing

these interfaces increases the attack vectors for adversaries. Thus, we will cooperate with PLC vendors to remove or lock these interfaces and leverage their dedicated network interfaces to deploy patches. Vendors have in-depth knowledge of their own hardware platforms and firmware architectures. The promising usage of RLPatch is that vendors may increase the number of hardware breakpoints in their PLCs and employ our hardware-assisted live patching strategy.

6 Related Work

Live patching. The history of dynamic software upgrades goes back to the late 1970s when telephone networks sought to upgrade their telecommunication switching systems [17]. Early systems and techniques typically update program functions at the source code level, often with explicit support from the target machine [24, 25, 39]. This work focuses on live patching rather than dynamic software upgrades.

Since security patches tend to introduce little semantic differences [29], several live patching techniques [3, 20, 42] have been developed to address the security updating issue on the Linux kernel. Other research systems, such as PatchDroid [28], KARMA [10], and InstaGuard [9], provide live patching approaches to solve the delayed security update problem on Android devices. To safely apply a patch to a running multi-threaded program, WFPATCH [34] applies it gradually to each thread individually at a local quiescence, while all other threads can proceed uninterrupted. However, these kernel tools are unsuitable for hot-patching PLCs, due to the potential overhead and delays associated with their complex update approaches.

Embedded devices directly execute code on flash, and the trampoline insertion requires erasing an entire flash block before updating, resulting in high patching latency. Therefore, HERA employs a dedicated FPB unit on Cortex-M3/M4 microcontroller units (MCUs) to hot-patch real-time embedded systems. It assumes that the source code of the RTOS, the OTA update service, and the FPB unit for instruction remapping are available. Wahler et al. [43, 44] proposed a live patching solution for component-based systems. They assumed that the components of the target system have strict boundaries between each other. However, there is no sufficient memory to load new components for outdated PLCs.

Patch development. Formal analysis cannot capture the dynamic behaviors of the PLCs, including runtime conditions such as input values and system states. The discovRE [16] and FirmUp [12] suffer from high false negative and positive rates, such as losing runtime vulnerabilities and identifying non-vulnerabilities. Inception [11] requires the firmware intensively interacts with external peripherals. The analyst can use our vulnerability testing method to construct taint logic instruction sequences and then use them to model the interaction behaviors of the industrial controllers. VUzzer [32], Avatar2 [26], and IoTFuzzer [8] require a set of initial test inputs compatible with the target system. However, internal functions in the stripped firmware images are unknown to the analyst, and the PLC vendors cannot provide the required input file format.

Among approaches for binary patch generation in stripped images, PatchScope [49], RapidPatch [21], and VULMET [48] are closer to our work. PatchScope excavates program memory objects and correlated input fields, and then it outputs the patch context information for security analysts. However, it causes approximately 15% false positive rates that are unacceptable for safe-critical controllers. RapidPatch uses one Extended Berkeley Packet Filter (eBPF) patch to fix the same vulnerability on different heterogeneous devices. Nevertheless, it necessitates the source code to generate hot patches. VULMET leverages the weakest preconditions obtained by calculating the semantic equivalence of the official patch to produce patches. However, VULMET cannot apply to the PLCs that use static links, instead of dynamic links, with the requirement of absolute address changes.

7 Conclusion

In this work, we proposed RLPatch as a practical solution to automate the live patching process of closed-source PLCs by using their inherent working mechanisms. In comparison to existing live patching approaches, the unique features of RLPatch include that (1) it can automatically patch the industrial controllers below the OS level, (2) it can work well with the PLC's runtime without intrusiveness, (3) it can use the time difference between the watchdog timer and the scan cycle to conduct live patching at a safe update point, and (4) it can efficiently construct hot patches from third-party binary. Since hardware-assisted live patching is 5-10 times faster than the instrumentation-based methods, we recommend vendors increase the number of hardware breakpoints in their PLCs. To validate the efficacy of RLPatch, we implemented a prototype of RLPatch and used it to hot-patch three Rockwell PLCs for fixing 20 MNRF vulnerabilities, and the results show that RLPatch incurs negligible and predictable overhead. A further evaluation of RLPatch on a real-world gas pipeline shows that RLPatch can eliminate the adverse impacts of MNRF attacks.

Acknowledgments

We would like to thank our shepherd Saurabh Bagchi and the anonymous reviewers for their insightful and detailed feedback. This work was supported in part by the U.S. Office of Navy Research (ONR) under Grant No. N00014-23-1-2158, National Science Foundation of China (NSFC) Beijing-Tianjin-Hebei Basic Research Cooperation Special Project under Grant No. V1640354653903), MIIT Special Project for Rebuilding Industrial Base and High-Quality Development of Manufacturing Industry under Grant No. 0747236ISC-CZA193, and CSC scholarship.

References

[1] ARM. 2021. About the Flash Patch and Breakpoint Unit (FPB). https://developer.arm.com/documentation/ddi0337/h/debug/about-the-flash-patch-and-breakpoint-unit--fpb- (visited on 03/01/2022).

[2] ARM. 2021. The EmbeddedICE-RT macrocell. https://developer.arm.com/documentation/ddi0234/b/debugging-your-system/the-embeddedice-rt-macrocell (visited on 03/01/2022).

[3] Jeff Arnold and M. Frans Kaashoek. 2009. Ksplice: automatic rebootless kernel updates. In *4th ACM European Conference on Computer Systems (EuroSys)*. ACM, Nuremberg, Germany, 187–198.

[4] Rockwell Automation. 2021. Industrial Cybersecurity: Security Solutions from Plant to Enterprise. https://www.rockwellautomation.com/en-ua/capabilities/industrial-security.html (visited on 03/01/2022).

[5] Rockwell Automation. 2021. Release Notes for ControlLogix Controllers. https://compatibility.rockwellautomation.com/GeneratedReleaseNote.aspx?v1=54988&v2=55442&o=&pdf=0 (visited on 03/01/2022).

[6] Zachary H. Basnight. 2013. *Firmware Counterfeiting and Modification Attacks on Programmable Logic Controllers.* Master's thesis. Air Force Institute of Technology (USAF).

[7] Paige Beach. 2017. ABB Jokab Safety Pluto safety PLC features hot-swap capabilities for in-the-field replacement. https://www.designworldonline.com/abb-jokab-safety-pluto-safety-plc-features-hot-swap-capabilities-field-replacement/ (visited on 03/01/2022).

[8] Jiongyi Chen, Wenrui Diao, Qingchuan Zhao, Chaoshun Zuo, Zhiqiang Lin, XiaoFeng Wang, Wing Cheong Lau, Menghan Sun, Ronghai Yang, and Kehuan Zhang. 2018. IoTFuzzer: Discovering Memory Corruptions in IoT Through App-based Fuzzing. In *23rd Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA, 1–14.

[9] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. 2018. InstaGuard: Instantly Deployable Hot-patches for Vulnerable System Programs on Android. In *25th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, San Diego, California, USA, 1–15.

[10] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. 2017. Adaptive Android Kernel Live Patching. In *26th USENIX Security Symposium*. USENIX Association, Vancouver, BC, Canada, 1253–1270.

[11] Nassim Corteggiani, Giovanni Camurati, and Aurélien Francillon. 2018. Inception: System-Wide Security Testing of Real-World Embedded Systems Software. In *27th USENIX Security Symposium*. USENIX Association, Baltimore, MD, USA, 309–326.

[12] Yaniv David, Nimrod Partush, and Eran Yahav. 2018. FirmUp: Precise Static Detection of Common Vulnerabilities in Firmware. In *23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. ACM, Williamsburg, VA, USA, 392–404.

[13] Ying Dong, Wenbo Guo, Yueqi Chen, Xinyu Xing, Yuqing Zhang, and Gang Wang. 2019. Towards the Detection of Inconsistencies in Public Security Vulnerability Reports. In *28th USENIX Security Symposium*. USENIX Association, Santa Clara, CA, USA, 869–885.

[14] Stephen J. Dunlap. 2013. *Timing-based Side Channel Analysis for Anomaly Detection in the Industrial Control System Environment.* Master's thesis. Air Force Institute of Technology (USAF).

[15] Schneider Electric. 2021. Cybersecurity Solutions. https://www.se.com/ww/en/work/solutions/cybersecurity (visited on 03/01/2022).

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *21st Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA, 58–79.

[17] Robert S. Fabry. 1976. How to design a system in which modules can be changed on the fly. In *2nd International Conference on Software Engineering (ICSE)*. IEEE Computer Society, San Francisco, CA, USA, 470–476.

[18] Abdullah Al Farooq, Jessica Marquard, Kripa George, and Thomas Moyer. 2019. Detecting Safety and Security Faults in PLC Systems with Data Provenance. In *2019 IEEE International Symposium on Technologies for Homeland Security (HST)*. IEEE, Woburn, MA, USA, 1–6.

[19] Luis A. Garcia, Ferdinand Brasser, Mehmet H. Cintuglu, Ahmad-Reza Sadeghi, Osama Mohammed, and Saman A. Zonouz. 2017. Hey, My Malware Knows Physics! Attacking PLCs with Physical Model Aware Rootkit. In *24th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, San Diego, CA, USA, 1–15.

[20] Red Hat. 2014. Introducing kpatch: Dynamic Kernel Patching. https://www.redhat.com/en/blog/introducing-kpatch-dynamic-kernel-patching (visited on 03/01/2022).

[21] Yi He, Zhenhua Zou, Kun Sun, Zhuotao Liu, Ke Xu, Qian Wang, Chao Shen, Zhi Wang, and Qi Li. 2022. RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices. In *31st USENIX Security Symposium*. USENIX Association, Boston, MA, USA, 2225–2242.

[22] Jim Keniston, Prasanna S Panchamukhi, and Masami Hiramatsu. 2022. Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt (visited on 03/01/2022).

[23] Geir M Køien. 2021. Zero-Trust Principles for Legacy Components. *Wireless Personal Communications* 121, 2 (2021), 1169–1186.

[24] Jeff Kramer and Jeff Magee. 1990. The evolving philosophers problem: Dynamic change management. *IEEE Transactions on software engineering* 16, 11 (1990), 1293–1306.

[25] Stephen McCamant and Michael D. Ernst. 2003. Predicting problems caused by component upgrades. In *9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*. ACM, Helsinki, Finland, 287–296.

[26] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. 2018. Avatar2: A Multi-Target Orchestration Platform. In *2018 Workshop on Binary Analysis Research (BAR) (colocated with NDSS Symposium)*. Internet Society, San Diego, CA, USA, 1–11.

[27] J. Mulder, M. Schwartz, M. Berg, Jonathan Van Houten, J. Urrea, and Alex Pease. 2012. *Reverse engineering industrial control system field devices.* Technical Report. Sandia National Laboratories.

[28] Collin Mulliner, Jon Oberheide, William K. Robertson, and Engin Kirda. 2013. PatchDroid: scalable third-party security patches for Android devices. In *29th Annual Computer Security Applications Conference (ACSAC)*. ACM, Orleans, LA, USA, 259–268.

[29] Hoan Anh Nguyen, Anh Tuan Nguyen, Tung Thanh Nguyen, Tien N Nguyen, and Hridesh Rajan. 2013. A study of repetitiveness of code changes in software evolution. In *28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, Palo Alto, CA, USA, 180–190.

[30] Christian Niesler, Sebastian Surminski, and Lucas Davi. 2021. HERA: Hotpatching of Embedded Real-time Applications. In *28th Annual Network and Distributed System Security Symposium (NDSS)*. Internet Society, Virtual Event, 1–16.

[31] Muhammad Haris Rais, Rima Asmar Awad, Juan Lopez, and Irfan Ahmed. 2021. JTAG-based PLC memory acquisition framework for industrial control systems. *Forensic Science International: Digital Investigation* 37 (2021), 301196.

[32] Sanjay Rawat, Vivek Jain, Ashish Kumar, Lucian Cojocar, Cristiano Giuffrida, and Herbert Bos. 2017. VUzzer: Application-aware Evolutionary Fuzzing. In *22nd Annual Network and Distributed System Security Symposium (NDSS)*. The Internet Society, San Diego, CA, USA, 1–14.

[33] F-Secure Labs Security Response. 2016. *BLACKENERGY and QUEDAGH: the convergence of crimeware and APT attacks*. Technical Report. F-Secure.

[34] Florian Rommel, Christian Dietrich, Peng Huang, Daniel Friesel, Sangeetha Abdu Jyothi, Karan Grover, Marcel Köppen, Nina Narodytska, Muthian Sivathanu, Christoph Borchert, et al. 2020. From Global to Local Quiescence: Wait-Free Code Patching of Multi-Threaded Processes. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. USENIX Association, Virtual Event, 651–666.

[35] Julian Rrushi, Hassan Farhangi, Clay Howey, Kelly Carmichael, and Joey Dabell. 2015. A quantitative evaluation of the target selection of havex ics malware plugin. In *2015 Industrial Control System Security (ICSS) Workshop*. ACM, Los Angeles, CA, USA, 1–5.

[36] Phil Salkie. 2017. *Legacy Industrial Control Systems - Secure / Replace / Ignore?* Technical Report. Jenariah Industrial Automation. https://static1.squarespace.com/static/5047a5a6e4b0dcecada15549/t/5ff8ca2ff96b3c097b38b7ec/1610140229707/Legacy+Industrial+Control+Systems+-+Salkie (visited on 03/01/2022).

[37] Carl D. Schuett. 2014. *Programmable Logic Controller Modification Attacks for Use in Detection Analysis*. Master's thesis. Air Force Institute of Technology (USAF), Wright-Patterson Air Force Base, Ohio.

[38] Moses D. Schwartz, John Mulder, Jason Trent, and William D. Atkins. 2010. *Control system devices: architectures and supply channels overview*. Technical Report. Sandia National Laboratories.

[39] Mark E. Segal and Ophir Frieder. 1993. On-the-fly program modification: systems for dynamic updating. *IEEE Software* 10, 2 (1993), 53–65.

[40] Siemens. 2021. Cybersecurity at Siemens. https://new.siemens.com/global/en/company/topic-areas/cybersecurity.html (visited on 03/01/2022).

[41] Siemens Product Support. 2020. Which modules can you replace with S7-1500 running? https://support.industry.siemens.com/cs/document/109744698/which-modules-can-you-replace-with-s7-1500-running-?dti=0&lc=en-WW (visited on 03/01/2022).

[42] SUSE. 2014. SUSE Linux Enterprise Live Patching. https://www.suse.com/products/live-patching/ (visited on 03/01/2022).

[43] Michael Wahler, Stefan Richter, Sumit Kumar, and Manuel Oriol. 2011. Non-disruptive large-scale component updates for real-time controllers. In *27th International Conference on Data Engineering (ICDE)*. IEEE Computer Society, Hannover, Germany, 174–178.

[44] Michael Wahler, Stefan Richter, and Manuel Oriol. 2009. Dynamic software updates for real-time systems. In *2nd ACM Workshop on Hot Topics in Software Upgrades (HotSWUp)*. ACM, Orlando, FL, USA, 1–6.

[45] Thomas Weber. 2021. Reverse Engineering Architecture And Pinout of Custom Asics. https://sec-consult.com/blog/detail/reverse-engineering-architecture-pinout-plc/ (visited on 03/01/2022).

[46] Sharon Weinberger. 2011. Computer security: Is this the start of cyberwarfare? *Nature News* 474, 7350 (2011), 142–145.

[47] Wikipedia. 2022. Colonial Pipeline Ransomware Attack. https://en.wikipedia.org/wiki/Colonial_Pipeline_ransomware_attack (visited on 03/01/2022).

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

[49] Lei Zhao, Yuncong Zhu, Jiang Ming, Yichen Zhang, Haotian Zhang, and Heng Yin. 2020. PatchScope: Memory Object Centric Patch Diffing. In *2020 ACM SIGSAC Conference on Computer and Communications Security (CCS)*. ACM, Virtual Event, 149–165.

[50] Zynamics. 2021. BinDiff Homepage. https://www.zynamics.com/ (visited on 03/01/2022).