

## Ghost in the PLC vs GhostBuster

***Citation for published version (APA):***

Abbasi, A., & Genuise, A. (2017). Ghost in the PLC vs GhostBuster: on the feasibility of detecting pin control attack in Programmable Logic Controllers. Unpublished. In *Ghost in the PLC vs GhostBuster: On the Feasibility of Detecting Pin Control Attack in Programmable Logic Controllers*

***Document status and date:***

Unpublished: 01/05/2017

***Please check the document version of this publication:***

- A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
- The final author version and the galley proof are versions of the publication after peer review.
- The final published version features the final layout of the paper including the volume, issue and page numbers.

[Link to publication](#)

***General rights***

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

[www.tue.nl/taverne](http://www.tue.nl/taverne)

***Take down policy***

If you believe that this document breaches copyright please contact us at:

[openaccess@tue.nl](mailto:openaccess@tue.nl)

providing details and we will investigate your claim.

Security and Embedded Networked Systems  
Department of Mathematics and Computer Science  
Eindhoven University of Technology

TECHNICAL REPORT

**Ghost in the PLC vs GhostBuster:  
On the Feasibility of Detecting  
Pin Control Attack in  
Programmable Logic Controllers**

*Ali Abbasi, Andrea Genuise*

October 2017

## **Abstract**

Programmable Logic Controllers (PLCs) are a family of embedded devices used for physical process control. Similar to other embedded devices, PLCs are vulnerable to cyber attacks. Because they are used to control the physical processes of critical infrastructures, compromised PLCs constitute a significant security and safety risk. Previously we introduced specific attack against PLCs which can stealthily manipulate the physical process it controls by tampering with the device I/O at a low level. We implemented different variants of the attack in the form of a rootkit and a user-space malicious code over a candidate PLC. We then move forward with a tailored defense which specifically detect modification of PLCs I/O to detect our attack.

# 1 Introduction

Embedded systems are often employed in mission critical systems that have to be both reliable and secure. In particular, it is important that their I/O (Input/Output) is stable and secure [36], as this is the way they interact with the outside world.

Digging into their architecture, we know that the I/O interfaces of embedded systems (e.g., GPIO, SCI, USB, etc.), are usually controlled by a so-called System on a Chip (SoC), an integrated circuit that combines multiple I/O interfaces. In turn, the pins in a SoC are managed by a pin controller, a subsystem of SoC, through which one can configure pin multiplexing or the input or output mode of pins. One of the most peculiar aspects of a pin controller is that its behavior is determined by a set of registers: by altering these registers one can change the behavior of the chip in a dramatic way. This feature is exploitable by attackers, who can tamper with the integrity or the availability of legitimate I/O operations, factually changing how an embedded system interacts with the outside world.

Based on these observations, in this chapter, we introduce a novel attack technique against embedded systems, which we call pin control attack. As we will demonstrate in the chapter, the salient features of this new class of attacks are:

First, it is intrinsically stealth. The alteration of the pin configuration does not generate any interrupt, preventing the OS to react to it. Secondly, it is entirely different in execution from traditional techniques such as manipulation of kernel structures or system call hooking, which are typically monitored by anti-rootkit protection systems. Finally, it is viable. It is possible to build concrete attack using it.

To demonstrate these points, in Section 2 we describe the state of the art in attacks against and defenses for embedded devices. We then discuss the parameters of an applicable host-based defensive solution for PLCs in Section 3. In Section 4, we describe a methodology for bypassing two defensive solutions for embedded devices.

We demonstrate the attack capabilities offered by Pin Control attack, together with the minimal requirements for carrying out the attack in Section 6. We argue that the attack capabilities include blocking the communication with a peripheral, causing physical damage to the peripheral, and manipulating values read or written by legitimate processes. We show how pin control can be exploited both with and without the attacker having kernel-level or root access.

To demonstrate the feasibility of our attack technique, in Section 7 we describe the practical implementation of an attack against a Programmable Logic Controller (PLC) environment by exploiting the runtime configuration of the I/O pins used by the PLC to control a physical process. The attack allows one to reliably take control of the physical process normally managed by the PLC, while remaining stealth to both the PLC runtime and operators monitoring the process through a Human Machine Interface, a goal much more challenging than simply disabling the process control capabilities of the PLC, which would any-

way lead to potentially catastrophic consequences. The attack does not require modification of the PLC logic (as proposed in other publications [43,44]) or traditional kernel tampering or hooking techniques, which are normally monitored by anti-rootkit tools.

We present two variations of the attack implementation. The first implementation allows an extremely reliable manipulation of the process at the cost of requiring root access. The second implementation slightly relaxes the requirement of reliable manipulation while allowing the manipulation to be achieved without root access.

Finally, in Section 7.6 we discuss potential mechanisms to detect/prevent Pin Configuration exploitation. However, because the pin configuration does happen legitimately at runtime and the lack of proper interrupt notifications from the SoC, it seems non-trivial to devise monitoring techniques that are both reliable and sufficiently light-weight to be employed in embedded systems.

## 2 Background

### 2.1 Attack Techniques

The attack techniques used against embedded devices can be divided into three categories: (i) firmware modification attacks, (ii) configuration manipulation attacks and (iii) control-flow attacks.

- **Firmware modification attacks:** in recent years, a number of firmware modification attacks against embedded devices have been researched and discussed. Cui et al. [12] demonstrated how the HP-RFU firmware update protocol can be exploited to allow adversaries to inject malicious firmware into HP printers. Traynor et al. [58] showed how to recursively compromise embedded devices and use them to create a network of malicious devices by manipulating their firmware. Wegner [64] demonstrated how to install a backdoor into Siemens office telephone communication devices by exploiting a vulnerability in their firmware verification system. Basnight et al. [5] illustrated that it is feasible to execute arbitrary code in a PLC by exploiting the firmware update feature, and finally, Peck et al. [48] showed how to exploit the Ethernet module of a PLC by uploading malicious firmware to it.
- **Configuration manipulation attacks:** these attacks allow an adversary to modify critical configuration parameters of an embedded device to force it to misbehave. For example, an anonymous security researcher with the nickname PT [49] demonstrated how to obtain access to a Private Branch Exchange (PBX), an embedded device used for telephone systems, by exploiting a vulnerability in the proprietary authentication protocol used by one vendor. A special case of configuration manipulation attacks concerns programmable devices, such as PLCs. PLCs can be programmed to control a physical process by following the logic specified by the user.

In this case, the attack consists of uploading a malicious logic to alter the manner in which the process is controlled. Falliere et al. [21] reported that the Stuxnet malware was used to manipulate the logic of PLCs from a programming station to subvert part of the uranium enrichment process at Natanz (Iran). In [43,44], McLaughlin et al. introduced two techniques for the dynamic generation of a malicious PLC control logic. To the best of our knowledge, the techniques proposed by McLaughlin et al. are, for the moment, limited in their practical applicability and have never been used in real-world attacks.

- **Control-flow attacks:** in general, this category of attacks consists of manipulating the execution flow of a running process. This is typically achieved by exploiting a stack/heap overflow or use-after-free vulnerability, which allows for the execution of arbitrary code by an adversary. Jump- and return-oriented programming (JOP and ROP) are considered to be control-flow attacks. Recent research has illustrated the possibility of control-flow attacks in embedded devices. For example, Beresford [6] presented multiple protocol vulnerabilities in Siemens PLCs that can allow an adversary to perform a remote code execution attack. Wightman demonstrated that Schneider Electric PLCs are vulnerable to buffer overflow attacks [29,65]. Heffner [46,50,51] presented multiple memory corruption vulnerabilities in home routers.

Although several techniques have been proposed to detect or prevent control-flow attacks on general IT systems, this class of attacks remains one of the most dangerous. Effective countermeasures that are simultaneously applicable in the domain and not circumventable by adversaries have yet to be developed. For example, Schuster et al. [54] evaluated several detection techniques for control-flow attacks [9, 24, 47] and claimed that attackers can bypass them using the code sequence within the executable modules of the target program. Davi et al. [16] introduced several techniques for bypassing detection techniques for control-flow attacks in multiple system security products [24,45,47]. Specifically, they showed not only that adversaries can find sufficient ROP gadgets within a program's binary code but also that by using long loops of NOP gadgets, they can create a long gadget chain and thereby break detection mechanisms for control-flow attacks.

## 2.2 Detection Techniques

We distinguish three main categories of techniques that have been proposed in the literature for host-based detection of attacks in embedded systems: (i) firmware integrity verification, (ii) memory verification and (iii) control-flow integrity.

- **Firmware integrity verification:** verifying the integrity of firmware allows one to detect or prevent firmware modification attacks. Such verification can be performed by the host when storing new firmware or at runtime.

Adelstein et al. [2] introduced a firmware-signing method that consists of a “certifying compiler” for firmware. The compiler allows the firmware to be verified by checking certain properties of the execution flow, memory and stack integrity in the firmware at runtime. Zhang et al. [67] introduced IOCheck, a framework to verify at runtime the integrity of firmware and the I/O configuration of computer I/O peripherals. After a (assumed trusted) BIOS boot, IOCheck leverages the System Management Mode of x86 CPU architectures to perform integrity checks that can be either executed at random polling intervals or driven by specific events. Finally, Duflot et al. [19] introduced NAVIS, a framework for the detection of firmware integrity manipulation in the memory of a network card by inspecting the memory accesses performed by the NIC processor against a model of expected behavior based on the memory layout profile of the adapter. A memory access that is outside the NIC memory profile is interpreted as an attempt to manipulate the NIC firmware.

- Memory verification: these techniques verify the integrity of executable code in memory at runtime. The most common technique for memory verification is attestation, which is used for low-power embedded devices.

Attestation is a challenge-response technique that allows an external application (the verifier) to verify the integrity of (parts of) the state of a system (the prover) against malicious modifications. Attestation techniques typically require the availability of dedicated hardware (e.g., a Trusted Platform Module). However, because of the practical limitations in embedded devices, certain works have focused on the development of pure software-based attestation techniques.

Seshadri et al. [55] introduced SWATT, a software-based attestation technique that can remotely verify the runtime memory contents of embedded devices and discover malicious modifications. SWATT uses a challenge-response protocol to remotely control the memory content of the embedded devices. LeMay et al. [38] proposed an ad hoc static kernel for smart meters that can cryptographically sign every new firmware version uploaded to a device. The signature is sent to the verifier to attest that the current (and previous) firmwares loaded on the smart meter are legitimate and integer. Armknecht et al. [3] introduced a framework for evaluating the security of software-based remote attestation techniques. The authors discussed the security properties of common basic cryptographic functions, such as pseudo-random number generators (PRNGs) and hash functions, when used for attestation purposes. They also discussed the possibility of leveraging time as a verification parameter to strengthen the security of an attestation scheme.

In an approach different from that of memory attestation frameworks, Cui et al. [13] proposed a new host-based deployment mechanism for embedded devices running operating systems, which they called a Symbiotic Embedded Machine or symbiote. The mechanism is specifically designed

to inject intrusion detection functionality into the firmware of such devices and to verify the integrity of its executable parts. A symbiote is a code structure embedded in a piece of firmware that can closely co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The symbiote is embedded in a randomized fashion to protect itself from removal, and the execution context of the symbiote is separated from that of the operating system to make it more resistant against adversaries. The authors demonstrated the deployment of a symbiote in the Cisco IOS firmware, with a low performance penalty and without an impact on the router’s functionality. Symbiotes cannot continuously monitor the entire firmware but rather set specific watchpoints and monitor certain executable locations of the firmware.

Reeves et al. [52] introduced a host-based intrusion detection system for embedded devices that leverages a built-in kernel tracing framework to identify anomalies in syscalls. The system is constructed by learning, for each monitored syscall, a list of known good source addresses. During detection, the system checks that when a certain syscall is invoked, the source of the call is on the safe list. Although its detection capabilities are limited, the approach also imposes a limited overhead on the system, which makes it suitable for being deployed in embedded devices such as those used in power grids (RTUs, IEDs and PLCs).

- Control-flow integrity:

Abad et al. [1] introduced a hardware-assisted CFI system for embedded devices. The system employs a dedicated hardware component to compare the control flow of the embedded device firmware at runtime to the CFG. The graph is constructed by disassembling the binary of the application to be protected. However, the method proposed for constructing the CFG does not consider indirect control-flow transfers (e.g., indirect function calls); therefore, the approach is incomplete and prone to certain types of control-flow attacks in which the adversary manipulates the control flow of the target application by changing the values of data memory areas. For example, the adversary may first spray the heap memory with shellcode instructions and then overwrite the value of a function pointer to point to a random heap address, which may contain the shellcode. Francillon et al. [23] proposed a hardware-assisted protection mechanism for AVR microcontrollers against control-flow attacks. The mechanism consists of separating the stack into a data stack and a control stack. The control stack is hardware-protected against unintended or malicious modifications (i.e., those not performed by call or ret instructions). Finally, Davis et al. [15] proposed a hardware-assisted CFI scheme that uses the hardware to confine indirect calls. This CFI scheme is based on a state model and a per-function CFI labeling approach. In particular, the CFI policies ensure that function returns can only transfer control to active call sides

(i.e., return landing pads of currently executing functions). Furthermore, indirect calls are restricted to target the beginning of a function, and finally, behavioral heuristics are used to address indirect jumps. To the best of our knowledge, no CFI for ECS exist in the literature.

### 3 Detection Mechanisms Applicable to PLCs

Not all of the defensive techniques described in Section 2.2 are practically applicable to embedded control devices such as PLCs. We consider five primary parameters to determine which defensive solutions are, in fact, practical. These parameters are as follows:

- Designed for embedded systems running an OS: there is a group of embedded devices, called low-end embedded devices, that do not have an operating system (OS). Devices that run micro-controller based processors (such as AVR or ATMEL) can be considered as low-end embedded devices. However, most of the ECS devices have a *real* OS. Therefore we only consider approaches that target embedded devices with a *real* OS.
- No hardware modification: the performance limitations make it difficult to introduce a complete host-based security mechanism for ECS. Most of the solutions described in Section 2.2 attempt to overcome these limitations by first considering hardware modifications of the embedded devices, thus making those solutions less attractive.
- Not virtualization required: the majority of embedded processors do not support hardware virtualization. Therefore, any implementation which purely relies on virtualization technologies can not be considered as solid solution for ECS.
- Limited performance overhead: The ECS devices are extremely heterogeneous in terms of processing power (which can be very low). Therefore we do not accept protection mechanisms which induce significant memory or CPU overhead. As observed by Szekeres et al. [57], protection mechanisms are likely to see widespread adoption if their average performance overhead is between 5% to 10%. Thus, we consider 10% overhead as the maximum acceptable performance overhead for applicable protection mechanisms.
- Overhead based on worst-case scenario: as mentioned earlier, performance characteristics play a huge part in determining whether security mechanisms are likely to be adopted or not. ECS devices usually have real-time constraints. Therefore, we do not consider average performance overhead values, but the worst-case overhead. If we only consider average performance overhead of a real-time ECS device and it exceed its average-case overhead bounds during a process control operation, it means the solution is unsuitable and potentially dangerous for system availability. Therefore,

it is essential to consider only worst-case overhead as acceptable performance overhead in the ECS.

Based on the parameters, we can identify two host-based detection mechanisms that, unlike most of the techniques described in Section 2.2, possess both desired qualifications. These host-based detection systems are Autoscopy Jr. [52] and Doppelganger [13].

These solutions are practically applicable, and because of their practical approach, they were adopted in the industry immediately upon their introduction [11, 52]. However, these approaches also exhibit certain weaknesses. Understanding these weaknesses assist us in designing better host-based solutions for embedded devices.

- Autoscopy Jr.: Autoscopy Jr. is a kernel control-flow monitoring system that searches for control-flow anomalies caused by function hooking in the kernel [52]. Autoscopy Jr. incurs only 5% CPU overhead, which is a significant achievement for a host-based detection system for embedded devices. Autoscopy Jr. specifically searches for kernel attacks in which the malicious code manipulates a function pointer. When a process calls a function with a manipulated pointer, the call is diverted to the malicious function instead of a legitimate one. The malicious function can then decide either to never call the original function or to call the legitimate function with a manipulated input. Autoscopy Jr. operates in two phases:
  1. Learning phase: In the learning phase, Autoscopy Jr. installs a character device driver that allows it to access the kernel memory by invoking `ioctl()`. Next, Autoscopy Jr. uses the device driver to monitor direct and indirect function calls and their corresponding return addresses. Afterward, it saves the return addresses of these functions, with certain runtime information (such as function arguments), to a data structure called the Trusted Location List (TLL). It then uses the TLL during the detection phase.
  2. Detection phase: During the detection phase, Autoscopy Jr. uses the previously installed device driver to monitor function calls. When a function that is listed in TLL is called, Autoscopy Jr. verifies the function address against the TLL entry for the same function. If the function address is not found in the TLL, it generates an alert.
- Doppelganger: Doppelganger is a host-based intrusion detection solution for embedded devices. It can detect both kernel- and application-level attacks in embedded devices. Doppelganger first analyzes the firmware of the embedded device to detect live code regions therein. Live code regions are executable parts of the firmware. Once Doppelganger detects the executable area of the memory, it randomly inserts its symbiotes (watchpoints) into the detected live code areas. Doppelganger symbiotes contain a CRC32 checksum of the randomly selected live code regions.

# Firmware

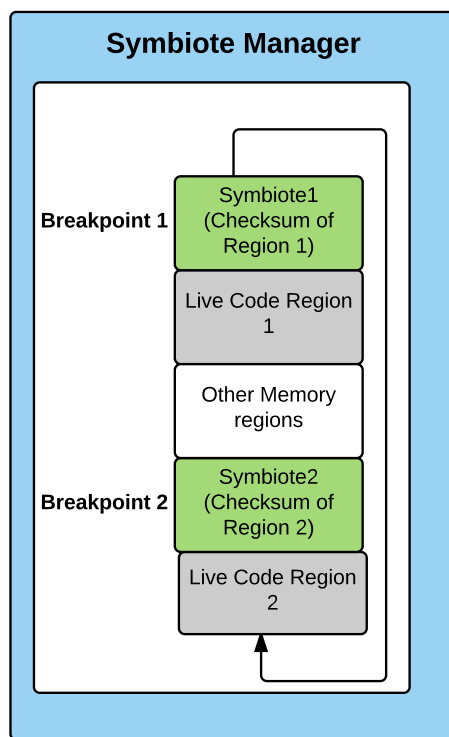


Figure 1: Structure of embedded device firmware controlled by Doppelganger

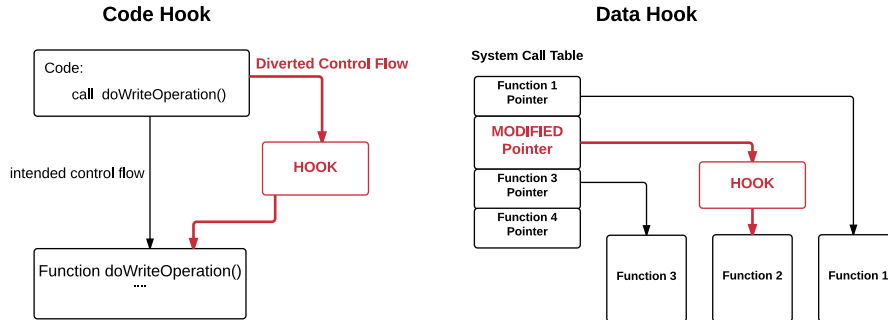


Figure 2: Typical function hooking

Doppelganger adds its symbiote manager at the beginning of the firmware. The symbiote manager can be regarded as a debugger that runs the firmware of the embedded system. The symbiote manager causes Doppelganger to run in a different context of the OS to make it resistant to attacks against its runtime. During the firmware execution, every time the symbiote manager detects a symbiote in memory, it stops the execution process (treating it as a breakpoint) and compares the current CRC32 checksum of the memory area with the symbiote checksum. If the checksum does not match, Doppelganger considers this finding to be evidence of a code modification attack and does not allow the processor to continue running the code. Figure 1 depicts the structure of embedded device firmware consisting of a symbiote manager and symbiotes.

## 4 Methodology for Evading Defensive Mechanisms

Both Autoscopy Jr. and Doppelganger provide practical host-based intrusion detection mechanisms for embedded devices with little performance overhead that can be applied to PLCs. Autoscopy Jr. detects kernel control-flow violations, and Doppelganger detects code modifications at runtime. However, both the Autoscopy Jr. and Doppelganger approaches suffer from certain shortcomings. These shortcomings can be divided into three types, each of which applies to at least one of the two approaches.

- **Static referencing:** Both Autoscopy Jr. and Doppelganger use static references to verify the execution flow or the integrity of an executable code region. Static referencing is comparable to signature-based approaches. If an attacker avoids the explicitly defined references, he can evade detection. The static references in Autoscopy Jr. are the entries of the TLL. In Doppelganger, the static references are the symbiotes.

None of these references can be modified during runtime. Autoscopy Jr. requires an additional learning phase to add more entries to the TLL, and Doppelganger requires the recreation of the firmware to insert additional symbiotes. These requirements limit the capabilities of both Doppelganger and Autoscopy Jr.: if an attacker inserts malicious code into locations that are not considered among the static references, then this malicious code can not be detected.

- **Function hooking:** In general, there are two types of function hooks: *code hooks* and *data hooks* [39, 60]. Both types of hooks are illustrated in Figure 2. In code hooking, an attacker can divert function calls by modifying executable parts of the kernel, such as the *.text* section. If the attacker wishes to hook the function call `doWriteOperation()`, as illustrated in Figure 2, he modifies the executable instructions that call `doWriteOperation()` to instead call its hook.

In data hooking, the attacker does not manipulate executable instructions; instead, he modifies the function pointers in the System Call Table (or other similar tables, such as the System Service Dispatch Table) to call their hooks. The System Call Table consists of pointers to system call functions. If an attacker modifies a function pointer and that function is then called by a process, the OS calls the *hook function* instead of the original function.

Unfortunately, Autoscopy Jr. detects only data hooks and is unable to prevent code hooking attacks. Moreover, because Autoscopy Jr.'s approach to detecting data hooking is not complete, an attacker can define his own versions of functions and call them separately. Autoscopy Jr. does not generate alerts for such unknown function calls since they are not functions that are listed in the TLL.

- **Dynamic memory:** Doppelganger sets its watchpoints prior to execution in the static executable parts of the firmware to be protected. We can compare the Doppelganger detection mechanism to code hooking detection mechanisms. Code hooking detection mechanisms search for modifications in the static parts of a kernel or application. This is a very similar approach to that of Doppelganger. Since it monitors only the static executable parts of the memory, Doppelganger is vulnerable to dynamic memory modification attacks (e.g., heap overflows). Doppelganger cannot detect any attack originating from dynamic memory.

The authors of Doppelganger claim that in their future research, it might be possible to verify the integrity of dynamic memory. Although verifying the integrity of dynamic memory might be possible, we argue that the detection mechanism they propose cannot be extended to dynamic memory since it is based on static information (the CRC checksum of the memory area). Therefore, it is not a straightforward extension to also monitor the content of dynamic memory.

Using a Loadable Kernel Module (LKM) is one of the methods an attacker can use to gain access to the kernel space to install a rootkit. The kernel uses `vmalloc()` to allocate LKMs into the heap area of the memory, which is dynamic memory. This type of allocation makes the executable instructions of a rootkit completely invisible to Doppelganger because it is not searching in dynamic memory.

Doppelganger, in its current implementation, can be bypassed when an attacker inserts malicious code into a part of the memory that contains dynamic contents. We call these parts of the memory *dynamic content memory*.

Dynamic content memory regions are memory regions that are statically allocated but whose contents can change dynamically. As a result, Doppelganger cannot create a checksum of these memory regions. An example of dynamic content memory is Thread-Local Storage (TLS). At the beginning of the execution of a process, the OS allocates a fixed chunk of memory for the TLS, but the TLS contents is used as dynamic content memory for temporary variables and data during the process. If an attacker inserts malicious code into the TLS and executes it from the TLS, Doppelganger will not be able to detect this malicious code execution because of the dynamic nature of the TLS.

One might assume that a combination of Autoscopy Jr. and Doppelganger could provide sufficient protection to detect both data hooking and code hooking. However, we have found that it is still possible to craft an attack that will go unnoticed even when both approaches are used in combination.

## 5 Pin Control in Embedded Systems

In an embedded SoC, pins are bases that are connected to the silicon chip. Each pin individually and within the group is controlled by a specific electrical logic with a particular physical address called a register. For example, "Output Enabled" logic means that the pin is an output pin and "Input Enabled" logic means that the pin is an input pin. In modern embedded systems these logic registers are connected to "register maps" within a SoC and can be referenced by the operating system (OS). These "Register maps" are a mere translation of physical register addresses in the SoC to reference-able virtual addresses in the OS. The concept of controlling these mapped registers with software is called Pin Control. Pin Control mainly consists of two subsystems namely Pin Multiplexing and Pin Configuration. Pin Multiplexing allows using a pin for different purposes by means of an electrical switch that changes the pin connection from one peripheral controller to another. Pin configuration is a process in which the OS or an application must prepare the I/O pins before using it. These two concepts are widely used in embedded systems and are part of the fundamental design within software and hardware architecture of both modern SoCs and OS kernels.

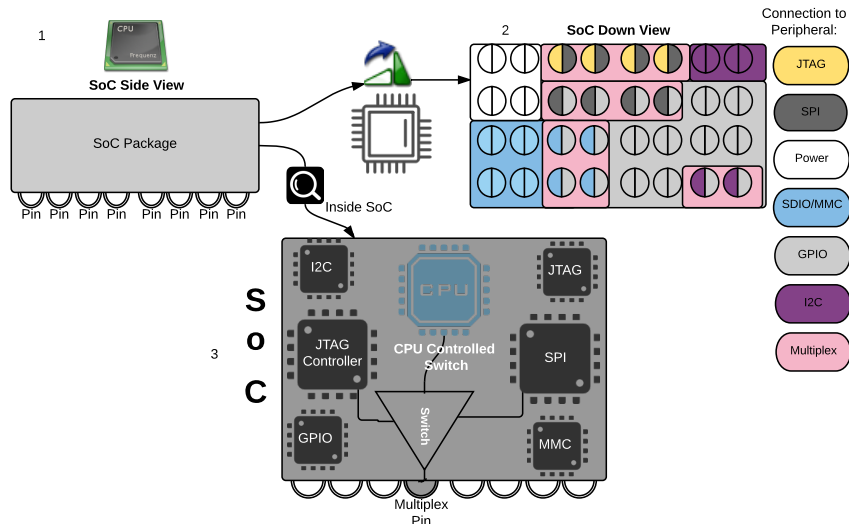


Figure 3: Part 1 shows an SoC from the Side view. Part 2 shows the design of multiple multiplex pins with different I/O peripheral. Part 3 shows how SoCs peripherals are located inside a SoC and how one multiplex pin is connected to two peripheral.

## 5.1 Pin Multiplexing

Embedded SoCs usually employ hundreds of pins connected to the electrical circuit. Some of these pins have a single defined purpose. For example, some only provide electricity or a clock signal. Since different equipment vendors with diverse I/O requirements will use these SoCs, the SoC manufacturer produces its SoCs to use a certain physical pin for multiple mutually exclusive functionalities, depending on the application [34]. The concept of redefining the functionality of the pin is called Pin Multiplexing and is one of the necessary specifications of the SoC design [25,61]. For example the SoC in Figure 3 has multiplex pins for JTAG/SPI, SPI/GPIO, MMC/GPIO and I2C/GPIO.

In Figure 3 each multiplex pin, gives a vendor options to choose between those two functionalities. Regarding the interaction of the Pin Multiplexing with OS, SoC vendors recommend to only multiplex the pins during the startup since there is no interrupt for multiplexing. However the user still can multiplex a pin at runtime and there is no limitation on that.

## 5.2 Pin Configuration

Embedded SoC I/Os (e.g., ARM, MIPS or PowerPC) are controlled with a pin based approach and must be configured otherwise these can not function properly. The configuration can be divided to two groups: Configuration at boot-time and configuration at runtime. We briefly describe both of these con-

figuration types and shows an example application for each of them. Due to the diverse types of configuration, it is impossible to mention them all one by one.

- Pin Configuration at boot-time: the boot-time configurations can be divided to two groups. Safety/filtering related configuration and functionality related configuration. A Safety/filtering configuration is used since the pins in the circuit board might receive a fluctuating electrical current that can cause damage to the circuit board or make the I/O readings inaccurate. This type of configurations regulate such fluctuation. Therefore, the OS or boot-loader usually enables them during the system boot time. The other group of pin configuration at system boot is pin functionality configuration. Before the applications use the pins, the OS must prepare the pin I/Os at boot time. Wiping all previously written configuration data on pin registers can be an instance of such configurations. During boot time, the kernel writes to all I/O configuration related registers with nulls (zero) to make them ready for the next configuration stages (e.g. run-time configuration). Once the previous settings are wiped out from the pin registers, the kernel will configure them with its preferred I/O configuration settings.
- Pin Configuration at run-time: once the system boots up, the I/O pins can be configured by applications or the drivers that use them. For instance, the I/O pins in a PLC that are used for reading and writing values must be configured. The PLC must set pins that are used for reading to input mode and pins that are used for controlling/writing values to output mode. Depending on the device and the scenario they are used in, pins can be reconfigured any number of times during runtime.

Similar to Pin Multiplexing, there is no interrupt for Pin Configuration.

### 5.3 How the OS configures pins

In what follows we assume that the embedded system runs a modern operating system (either RTOS or Unix Based OS) with a MMU (Memory Management Unit). The Pin Configuration process starts by initializing the multiplexing features of the pins. Pin Multiplexing is usually done by the boot-loader. The boot-loader first maps the I/O multiplexing registers to virtual addresses and writes the configuration details to these addresses. In some cases in which the kernel does the multiplexing, it just receives a pointer from the boot-loader containing the memory address at which configuration details for pin multiplexing are located. During kernel startup, the configuration registers related to multiplexing are mapped and the configuration details are applied. Eventually, the kernel removes the configuration details from the kernel memory space.

Once the initial multiplexing setup is finished, the start-up configuration of the pins is initiated. Once again, the configuration starts by mapping the physical address of registers related to pin configuration at run-time to virtual addresses. However, this time a device driver performs the task. After

the mapping, the driver writes appropriate configuration detail to those virtual addresses that get mapped to the physical configuration registers in the SoC.

## 5.4 Security concerns regarding Pin Control

The Pin Control process raises two security concerns: (1) one can multiplex a pin or reconfigure it at runtime while another process is using it and (2) the lack of interrupt and interrupt handlers for both Pin Multiplexing and Pin Configuration. If the multiplexing or configuration of a pin changes, neither the driver nor the application will notice it and will continue their tasks.

For example, assume that an application uses a particular peripheral controller connected to a pin with a particular multiplexing setup. At one point another application (second application) changes the multiplexing setup of the pin used by the first application. Once the pin is multiplexed, the physical connection to the first peripheral controller gets disconnected. However, since there is no interrupt at hardware level, the OS will assume that the first peripheral controller is still available. Thus, the OS will continue to carry out the write and read operations requested by the application without any error. A similar problem exists in Pin Configuration at runtime. If a pin (e.g. GPIO) that is set to Output mode gets reconfigured to Input mode by a second application, the driver or kernel will assume that the pin is still in the output mode and attempt the write operation without reporting any error. The processor then ignores the write operation (since the pin is in input mode) but will not give any feedback to the OS that the write operation was ignored.

We have brought these security concerns to the attention of the Linux Kernel "Pin Control Subsystem" group, which has confirmed our findings, but could not suggest a viable solution that did not require expensive artifacts such as ARM TrustZone or TPMs.

## 6 Pin Control Attacks

In this section, we describe a new type of attack that targets PLCs based on the methodology described in Section 4. A pin control attack basically consists of misusing the pin control functionalities of the embedded system at runtime. An attacker can either block communications with peripherals, cause physical damage to them or manipulate values read or written to/from a peripheral by a legitimate process.

To block the communications with a peripheral, an attacker can just change the multiplexing features of a pin and physically terminate the connection. So while an application is interacting with a peripheral, an attacker modifies the multiplexing registers of the SoC and activates the second peripheral thus physically disconnecting the first peripheral.

To cause physical damage, an attacker can use a combination of Pin Configuration and Pin Multiplexing. For example, a pin which can be multiplexed between an MMC (memory controller) and a PWM (Pulse-width modulation)

controller can be targeted to cause physical damage. If the pin is multiplexed to be used as an MMC controller, an attacker can multiplex the pin to connect it to a PWM controller and modify the PWM controller to push a significant pulsing electrical current toward the memory controller causing the memory to burn.

The ultimate use of a pin control is to manipulate read or write operations to a peripheral. This attack can have significant consequences, since it can be used to alter the way an embedded system interacts (and possibly controls) the outside world. This can be done by misusing Pin Configuration. Therefore, we call this particular kind of attack a Pin Configuration attack. In a Pin Configuration attack, the attacker will change the mode of pins from input to output and vice versa to control what a legitimate process writes or reads from the pin. Because a Pin Configuration attack is the most complex variant of Pin Control attacks, we show its viability by providing a practical implementation in Section 7.

Finally the novelty of our attack lies in the fact that to manipulate the physical process we do not modify the PLC logic instructions or firmware [5, 6, 12, 43, 44]. Instead, we target the interaction between the firmware and the PLC I/O. This can be achieved without leveraging traditional function hooking techniques and by placing the entire malicious code in dynamic memory (in the rootkit version of the attack), thus circumventing detection mechanisms such as Autoscopy Jr. and Doppelganger. Additionally, the attack causes the PLC firmware to assume that it is interacting with peripherals while, in reality, the connection between the I/O pins and the PLC process is being manipulated.

## 6.1 Threat model

For modifying the Pin Configuration, we can envision an attacker with a system privilege that gives her access to the Pin Configuration registers. An attacker outside kernel space can get access to these registers by using interfaces such as `/dev/mem`, a driver call or via Exported Kernel Object File System (`sysfs`). In particular, the root user has access to all of these interfaces.

Unlike in personal computers, most services and applications run as root in embedded systems (mostly to increase performance by reducing syscall requests). Several vulnerabilities have been discovered in applications running as root in embedded systems [6, 8, 17, 27, 59].

It is worth mentioning that root access is not always required to access the Pin Configuration registers. Due to performance limitations and the need for I/O access for certain applications, many non-root users can still access I/O registers. For example, a PLC runtime that does not run as root still has the required privileges to modify the Pin Configuration registers to function properly.

For the detection part of our work, we assume that the attacker cannot tamper with operating system (kernel) functions and data structures, but it can still use dynamic memory to insert its malicious code and tamper with the I/O configuration. The attacker can also write its own version of kernel functions

if needed, and call them separately in order to avoid HIDSeS. This approach, anyway, requires very high effort and would really be the last option for an attacker.

The attacker, depending on the attack implementation, may need different running privileges on the target system to conduct the attack. Generally speaking, the minimum privilege required is the privilege level of the PLC runtime, which has access to the I/O configuration registers. As discussed earlier, many security advisories have shown that PLCs have vulnerabilities that could lead to malicious code execution and they may affect PLC runtime software as well. Thus, obtaining the same PLC runtime privilege level is feasible on real systems. We assume that the attacker knows both the physical process controlled by the target system and the mapping between I/O configuration and external sensors and actuators. The former is typically known by the attacker, which has reasonably studied its target before conducting the attack. This is confirmed by Stuxnet [21], where attackers had very deep knowledge of target system and physical process. The latter is given by a knowledge of the PLC logic, which, as said, already comes with the mapping between I/O interfaces and control variables used by the logic. Furthermore, the research presented in [42] shows that it is possible to infer the structure of the devices connected to the PLC and used by the logic, factually lowering the bar for attackers since she does not need an a priori knowledge of the I/O mapping anymore.

## 7 A Pin Control Attack in Practice

In this section we describe the practical implementation of an attack against a Programmable Logic Controller (PLC) environment by exploiting the configuration of the I/O pins used by the PLC to control a physical process.

PLCs play a significant role in the industry since they control and monitor industrial processes in critical infrastructures [31]. For this reason, the successful exploitation of a PLC can affect the physical world and, as a result, can have serious consequences for the safety of equipment and human life [36]. For example, an adversary may manipulate the value of tank pressure sensors in a pressure sensitive boiler thus leading to the explosion of the boiler, or, similarly to Stuxnet, change the frequency of variable speed drives of centrifuges in a uranium enrichment facility, leading to damage of the centrifuge cascades. Consequently, one of the main objectives when attacking a PLC is to manipulate the physical process by intercepting the signals received from sensors and altering the ones sent to the actuators controlled by the PLC in such a way that the PLC has no way to tell that his communication with the I/O is being manipulated. This can be achieved by pin control exploitation without leveraging traditional function hooking or kernel data structure modification techniques [4].

Generally speaking, an attacker can manipulate the PLC read and write operations to its I/Os by leveraging the configuration of pins as follows:

1. For write operations: if the PLC software is attempting to write a value to an I/O pin that is configured as output, the attacker reconfigures the I/O

pin as input. The write operation will not succeed, but the PLC software will be unaware of this.

2. For read operations: if the PLC software is attempting to read a value from an I/O pin that is configured as input, the attacker can reconfigure the I/O pin as output and write the value that he wishes to feed to the PLC software in the reconfigured pin.

We implement this strategy in two variants of the attack. In the first variant we assume that the attacker has root access to the PLC. In the second variant we assume that the attacker has the same access level as the PLC software. In this section before discussing the technical implementation of the attack, we discuss in more details how a PLC generally works, and how the specific environment in which we built our attack is set up.

## 7.1 PLC operations

The main component of a PLC firmware is a software called the *runtime*. The runtime interprets or executes process control code known as the *logic*. The logic is a compiled form of the PLC's programming language, such as function blocks or ladder logic. Ladder logic and function block diagrams are graphical programming languages that describe the control process. A plant operator programs the logic and can change it when required. The logic is therefore dynamic code, whereas the runtime is static code.

The purpose of a PLC is to control field equipment (i.e., sensors and actuators). To do so, the PLC runtime interacts with its I/O. The first requirement for I/O interaction is to map the physical I/O addresses (including pin configuration registers) into memory. As described earlier the drivers, kernel or PLC runtime map the I/O memory ranges. Additionally, at the beginning of logic execution, the PLC runtime must configure the processor registers related to pin configuration in order to set the appropriate mode (e.g., input or output) of each I/O according to the logic.

After pin configuration, the PLC runtime executes the instructions in the logic in a loop (the so-called program scan). In a typical scenario, the PLC runtime prepares for executing the logic at every loop by scanning its inputs (e.g., the I/O channels defined as inputs in the logic) and storing the value of each input in the variable table. The variable table is a virtual table that contains all the variables needed by the logic: setpoints, counters, timers, inputs and outputs. During the execution, the instructions in the logic manipulate only values in the variable table: every change in the I/O is ignored until the next program scan. At the end of the program scan, the PLC runtime writes output variables to the related part of the mapped memory that eventually is written to the physical I/O by the kernel. Figure 4 depicts the PLC runtime operation, the execution of the logic, and its interaction with the I/O.

A PLC typically consists of separate digital and analog inputs and outputs. Because PLCs are digital systems, they cannot control analog input and output without additional hardware components. Digital-to-Analog Converters (DACs)

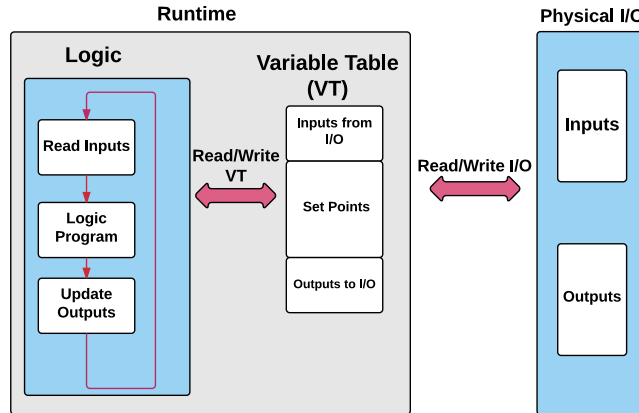


Figure 4: Overview of PLC runtime operation, the PLC logic and its interaction with the I/O

for analog outputs and Analog-to-Digital Converters (ADCs) for analog inputs form part of the analog interface of a PLC. These components read or write analog values by converting them to or from digital outputs or inputs to allow the PLC to interact with its analog interfaces. The DACs and ADCs are not separate components of the PLC but rather an integral part of the PLC circuit board. One can argue that the basis of I/O interaction in PLCs is digital. Analog control is simply a conversion of digital signals into analog signals or analog signals into digital signals.

## 7.2 Environment setup

### 7.2.1 Target Device and Runtime

To mimic a PLC environment we choose a Raspberry Pi 1 model B as our hardware, because of the similarity in CPU architecture, available memory, and CPU power to a real PLC. The Raspberry Pi 1 uses a Broadcom BCM2835 single-core processor with a clock speed of 700 MHz. As runtime we use the CODESYS platform. CODESYS is a PLC runtime that can execute ladder logic or function block languages on proprietary hardware and provides support for industrial protocols such as Modbus and DNP3.

Currently, more than 260 PLC vendors use CODESYS as the runtime software for their PLCs [17]. The combination of features offered by the Raspberry Pi and the CODESYS runtime make such a system an alternative to low-end PLCs.

The Raspberry Pi includes 32 general-purpose I/O pins, which represent the PLC's digital I/Os. These digital I/Os can also control analog devices by means of various electrical communication buses (e.g., SPI, I2C, and Serial) available for the Raspberry Pi with external hardware such as ADC or DAC

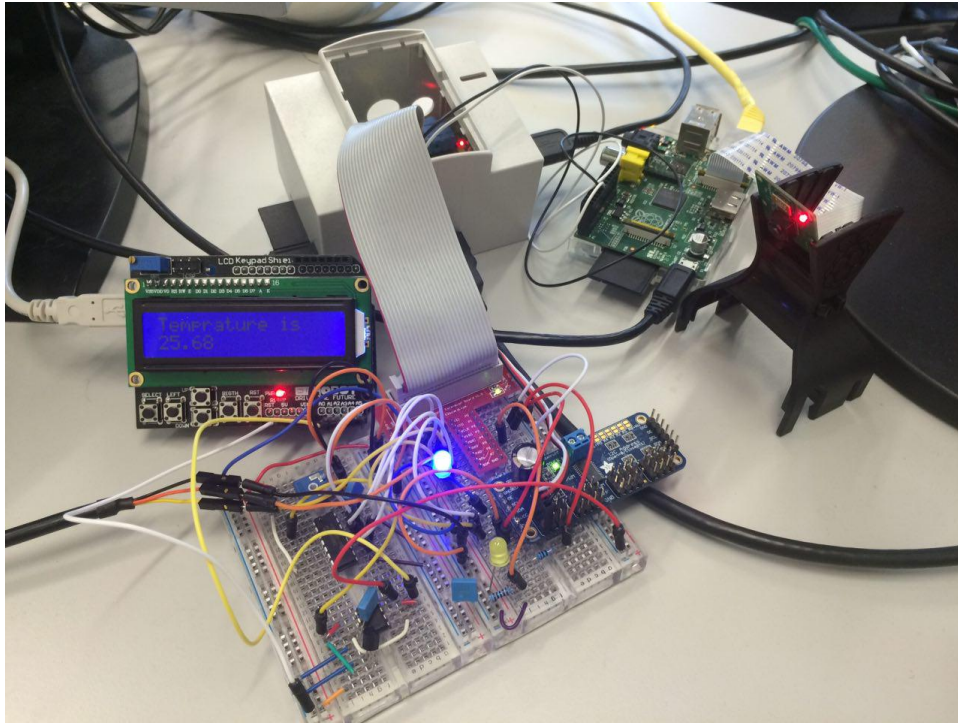


Figure 5: Target platform connected to multiple I/O interfaces

circuit boards. Figure 5 depicts our target platform connected to multiple analog and digital I/Os via ADC and DAC controllers.

### 7.2.2 The Logic and the Physical Process

We use pins 22 and 24 of the Raspberry Pi to control our physical process. In our control logic, we declare pin 22 as the output pin and pin 24 as the input pin. In the physical layout, our output pin is connected to an LED and our input pin is connected to a button. The electrical current becomes disconnected when the button is pressed and is reconnected when the button is released. From the perspective of the runtime, if no one is pressing the button, the input pin has a value of `TRUE`, whereas if someone is pressing the button, the pin has a value of `FALSE`.

The pseudo-code for the logic that controls these two I/Os is illustrated in Algorithm 1. According to our logic, the LED turns on or off every five seconds. If someone is pressing the button, the LED simply maintains its most recent state until the button is released, at which time the LED begins again to turn on and off every five seconds.

---

**Algorithm 1** Logic for our representative physical process

---

```
1: Input ← Pin24
2: Output ← Pin22
3: function MAINLOGIC(C)                                ▷ Read I/O
4:   while True do
5:     Read Input
6:     while Input is TRUE do
7:       function SWITCH_STATE(OUTPUT, FIVE SECONDS)(C)  ▷
       switching I/O state every 5 sec, States are High or Low.
8:     end function
9:   end while
10:  if Input False then
11:    Hold the hold the state of the output
12:  else if Go to first While then
13:    end if
14:  end while
15: end function
```

---

Table 1: I/O memory map in the BCM2835 processor

Operation	Size of a pin in the register	Address of Pin 0	Address of Pins 22 and 24
Input/Output Mode register	3 bits	0x20200000	0x20200002
SET register	1 bit	0x2020001C	0x2020001C
READ register	1 bit	0x20200034	0x20200034

### 7.2.3 Interaction Between the Virtual I/O Registers and the Runtime

To understand how the CODESYS runtime interacts with the two pins, we briefly describe how virtual I/O registers operate in a BCM2835 processor (the one used in the Raspberry Pi). The virtual I/O registers are simply I/O address ranges in the processor memory that perform different types of I/O operations. The operation types, the memory sizes, and the physical addresses of the three virtual I/O registers related to read, write, and I/O configuration operations in the BCM2835 processor are summarized in Table 1.

The three virtual I/O registers that are used in our logic are as follows:

1. Input/Output Mode register: this register sets pins to input or output mode. Each pin in this register has three bits of space. The first bit of each pin’s bit space defines whether the pin is used for input or output. If the first bit for a pin is set to 0, the pin is an input pin. If the first bit is set to 1, the pin is an output pin. The CODESYS runtime cannot change the value of a pin in input mode; it can only read from it. Even if CODESYS writes a value to an input-mode-enabled pin, the operation has no physical

effect and is ignored by the processor. However, the CODESYS runtime can change the value of a pin when the pin is in output mode. The input/output mode address range begins at offset 0x2020000 and ends at address 0x20200002. For pin 22, bits 6 to 8 of address 0x20200002 are used.

2. SET register: if a pin is declared as an output pin in the Input/Output Mode register, then every write operation related to this pin in the SET register address can immediately set the pin to high or low. High means that the Raspberry Pi directs an electrical current to the pin, and low means that the electrical current is disconnected from the pin. Every pin in this register has a 1-bit space. Assuming that pin 22 is in output mode, setting a value of “0” or “1” in bit 21 (bit 0 for pin 1, bit 21 for pin 22) of this register causes pin 22 to be set high or low (turning the LED on or off), respectively. The physical address for this I/O register is 0x2020001C.
3. READ register: the CODESYS runtime can read the values of the pins from the READ register. Every pin in this register has a 1-bit space. The values in the READ register have a direct relation with the values in the SET register. For example, if the CODESYS runtime writes a value of “1” to the SET register associated with pin 22 when this pin is configured as output mode, then the READ register value for the corresponding pin is updated to “1” as well.

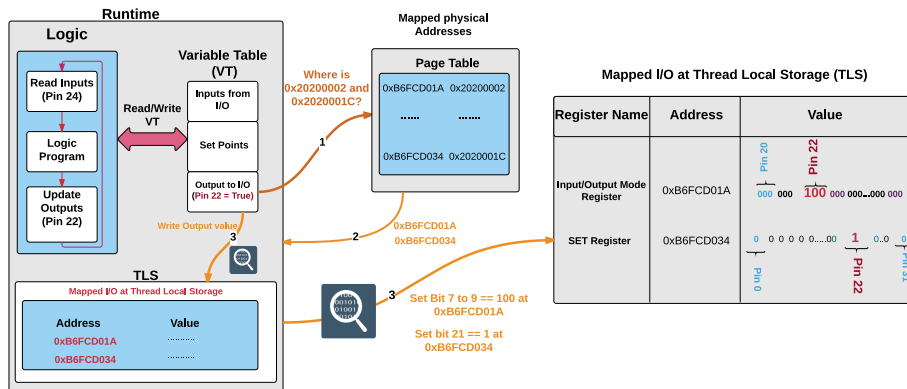


Figure 6: A search for the address of pin 22 and a write operation to it

We illustrate how the CODESYS runtime interacts with virtual I/O registers by describing a write operation for pin 22 (see Figure 6). Read operations follow a similar procedure. During the startup of the CODESYS runtime, the OS maps the addresses of the I/Os into the CODESYS Thread Local Storage (TLS). Once the I/Os are mapped, the mapped I/O addresses are recorded in the page table

and in a cache that is called the Translation Lookaside Buffer (TLB). The page table is a structure in which the OS maintains the list of mapped physical addresses and their corresponding virtual addresses for the process. The page table can become so large that searching it can be time consuming for the OS. To avoid this scenario, the OS also uses a cache for the page table, the TLB.

After our logic is uploaded to the Raspberry Pi, the CODESYS runtime evaluates the I/O configuration specified in the logic to determine which pins are designated as input or output. Pins 22 and 24 are designated as output and input respectively. Because pin 22 is declared as an output pin in our logic, CODESYS performs a write operation in the Input/Output Mode register and set bits 6 to 8 of offset 0x20200002 to 100. To execute this write operation, the CODESYS runtime asks for the virtual address of 0x20200002. The OS looks up the virtual address of 0x20200002 (mapped address of 0x20200002) in the TLB and page table. In our environment, this address was located as expected in the TLS memory area of the CODESYS runtime with value 0xB6FCD01A. Once CODESYS has retrieved the virtual address, it writes to it and continues operations. CODESYS then assumes that the I/O configuration sequence was successful and that the I/O pins are ready to use.

The CODESYS runtime then begins to execute the logic. When the logic updates the value for pin 22 to “1” (high) to turn on the LED, the CODESYS runtime writes a value of “1” to bit number 21 of the address 0x2020001C in the SET register. However, CODESYS needs to know the virtual address of 0x2020001C. Therefore, the CODESYS runtime looks in the TLB and page table for the mapped address of 0x2020001C. In our case, the address was located at 0xB6FCD034. Once the OS has found the virtual address, a value of “1” is written to the register and a *success* result is returned to the calling function. The CODESYS runtime then updates the I/O state in the SCADA or Human Machine Interface (HMI) software and reports that pin 22 is high (the LED is on).

### 7.3 Attack Implementation with Root Access

To be able to accurately tamper the control flow set in the logic we must be able to intercept each read and write operation of the CODESYS runtime. However, if we were to use conventional function hooking techniques (e.g hooking the CODESYS functions responsible for reporting the I/O status) or modifying the integrity of the code in the PLC (e.g., modifying the codes of the runtime or kernel data structure), most control flow and code integrity security mechanisms would be able to detect and block our attempts. Therefore, we leverage the processor debug registers for interception. Debug registers were introduced to assist developers in analyzing their software, and all new processors with various different architectures (ARM, Intel, and MIPS) have such registers. These registers allow setting hardware breakpoints to specific memory addresses. Once an address that is in a debug register is accessed by a process, the processor interrupt handler is called and customized code can be executed.

Unfortunately, for PLCs running a modern OS with a memory management

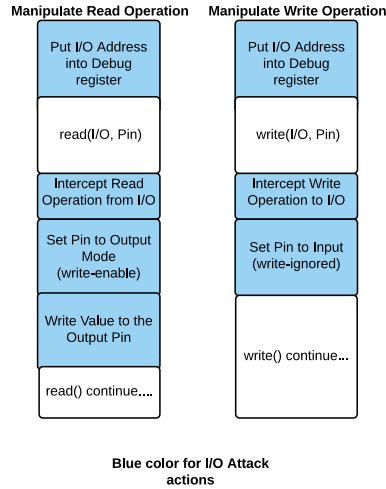


Figure 7: Steps of the Pin Control for read and write manipulation

unit, setting debug registers requires root access. This can be achieved either by leveraging default passwords, through a control-flow attack against the PLC runtime, or through a firmware modification attack [5, 12, 48, 58]. Regarding default passwords, several reported vulnerabilities suggest that some PLCs provide shell access with default root passwords [7, 18]. An attacker could just log in to these devices using the default password and execute the attack. Using a control-flow attack, the attacker can gain access at the same level as the PLC process. As described earlier, previous research has revealed that various PLCs run their runtime as the root user by default [6, 17]. In case that the PLC runtime is vulnerable to control-flow attack but is not running as root, the attacker needs a privilege escalation vulnerability to gain root access to the PLC. Finally, by installing rogue firmware into the PLC, the attacker can infect every binary in the PLC. This can give the attacker complete leverage over the PLC operating system.

We also assume that the attacker knows the physical process (by the mean of reading the logic exist in the PLC) and is aware of the mapping between the I/O pins and the logic. The PLC logic might use various inputs and outputs to control its process; thus, the attacker must know which input or output must be modified to affect the process as desired. The work presented by McLaughlin et al. [43, 44] can be used to discover the mapping between the different I/O variables and the physical world.

As the first stage of our attack, we set the mapped I/O addresses to the debug register and intercept every write or read operation of the CODESYS runtime. When the PLC runtime wants to read from or write to the I/O pins, the processor halts the process and calls the attacker interrupt handler. The handler

performs the I/O manipulation by exploiting the pin configuration functionality as discussed earlier. Figure 7 depicts this process.

For our experiment, we implemented the attack in an LKM (Loadable Kernel Module). Once our LKM module is loaded, it checks the CPU information of the machine and matches it against a hard-coded list of CPUs and their I/O memory ranges. As mentioned above, the I/O addresses of the BCM2835 processor begin at 0x20200000. Using this information, the LKM looks in the OS page table for a process ID and a virtual address that are mapped to the physical address 0x20200000. Because target pins are known in advance (in our case, pin 22), the correct register address for the SET, READ, and Input/Output Mode registers can be easily calculated.

To manipulate write operations, the LKM inserts the virtual address from the SET register associated with pin 22 (0xB6FCD034) into the BCM2835 processor’s debug register and installs its custom interrupt handler. When the CODESYS runtime requests a write operation to pin 22 (for example, let us assume that it writes a value of 0 into the SET register to turn off the LED), our custom exception handler is called. The exception handler changes the state of pin 22 from output mode to input mode by writing the values of “000” to bits 6 to 8 of the Input/Output Mode register in 0xB6FCD01A. After changing the state of pin 22, the processor allows the CODESYS runtime to execute its command. CODESYS attempts to write the desired value and returns a success result, even if the value is not set in the register as the pin’s mode has been switched to input. At this point, our LKM has full control over the CODESYS I/O operations and can freely decide whether to allow an I/O state change.

To manipulate read operations on pin 24 the rootkit inserts the virtual address from the READ register associated with pin 24 into the debug register and installs its custom exception handler. Once CODESYS accesses the virtual address from the READ register to read the value from the pin, the exception handler executes. The exception handler first sets pin 24 as an output pin by writing the values “100” to the related bits of the Input/Output Mode register and then writes the desired value for pin 24 into the SET register. The LKM then returns control to the CODESYS runtime, which will read the value written in the I/O register by the exception handler instead of the real value.

With this technique we were able to successfully alter the physical process described in Section 7.2.2. We modified the process by allowing the PLC to turn on and off the LED on pin 22 only every ten seconds instead of every five. Additionally, the read manipulation made the button in our physical layout ineffective. We could hold the last state of the LED by giving fake read input values to the runtime and make the runtime assume that someone pressed the electrical button while it was not the case.

### 7.3.1 Overhead of Attack with Root Access

Embedded devices typically have limited resources for the operations they execute. This is the case for PLCs as well. While in general performance overhead is not an issue for the attacker, it can be when a PLC controls processes that

are time critical. If in such processes the performance overhead causes significant delay in the I/O speed, it can uncover the attack. For this reason we evaluated the performance overhead imposed by this attack on our selected hardware (Raspberry Pi model 1 B). Regarding CPU overhead, based on our evaluation, in the rootkit based pin control attack on average incurs 5% CPU overhead for the manipulation of write operations and 23% CPU overhead for the manipulation of read operations. Read operation manipulation imposes a higher CPU load for two reasons. First, the PLC runtime environment reads the values from the I/O multiple times per second, thereby significantly increasing the CPU overhead, whereas for write operations, the number of I/O write operations depends only on the logic (in our case, every five seconds). Second, read manipulation requires two instructions (setting the pin to output mode and writing to it), whereas write manipulation requires only one instruction (setting the pin to input mode). Figure 8 depicts the CPU overhead incurred by the manipulation of read and write operations in a rootkit based pin control attack. The additional CPU overhead is not an important concern for the attacker, but it creates anomalies in the power consumption of the victimized device.

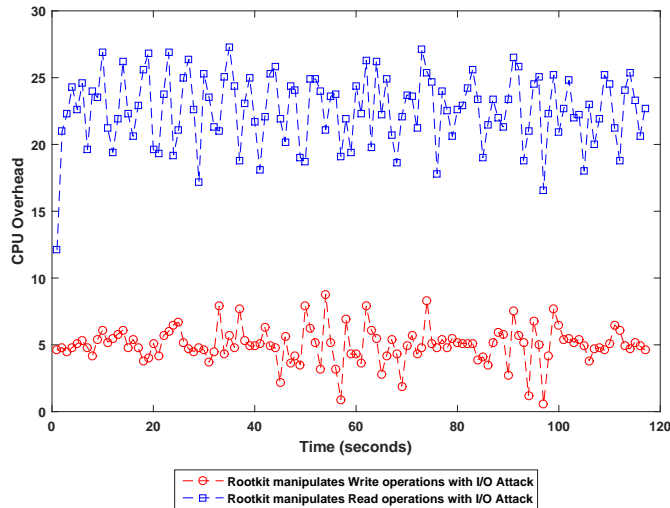


Figure 8: CPU overhead in rootkit based Pin Control attack

To understand the impact of an rootkit based pin control attack on control operations, we evaluated the I/O speed fluctuations in our selected setup (Raspberry Pi with CODESYS runtime running our sample logic). Figure 9 depicts the fluctuation of the I/O speed with and without our rootkit implementation. On average the speed where our hardware could write to the I/O (without our rootkit) was 3.97 milliseconds. When the rootkit manipulates the I/O (intercept the I/O write operation and write the same value), the average speed of the I/O increased to 4.01 milliseconds.

The difference in I/O speed with and without rootkit is insignificant. Additionally, in a normal state (no rootkit operating), the I/O speed has a similar fluctuation to when our rootkit is executing a pin control attack.

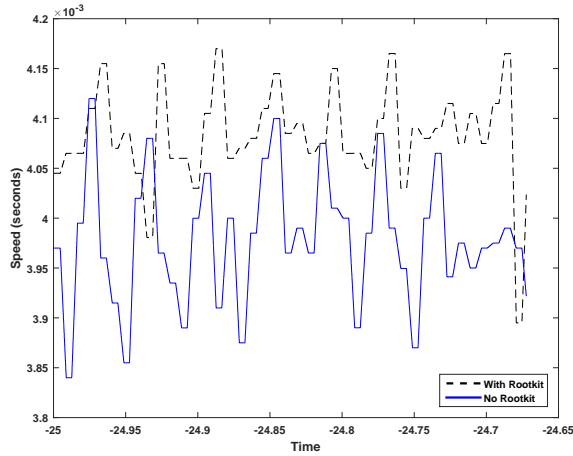


Figure 9: I/O speed with and without rootkit

## 7.4 Attack Implementation Without Root Access

The previous implementation of the attack allows to precisely tamper the I/O values in the logic. However such precision comes at the cost of the high privilege requirements and the non-negligible performance overhead due to the usage of debug registers (which causes the call of hardware interrupt handlers of the SoC). By relaxing a bit our precise I/O modification requirement, we can create a second implementation of the attack which has better performance and does not require root/kernel access, while still being able to manipulate the physical process as desired.

The requirement for this second implementation is that the attacker has the same access privileges as the PLC runtime. This is achievable for example by exploiting a memory corruption vulnerability that allows code execution, such as a buffer overflow [6, 7, 30, 65]. A remote code execution vulnerability of such kind is known to be affecting the CODESYS runtime [66]. Also, similarly to our previous implementation, we assume that our application already has access to the logic (since the logic will be inside the PLC) and knows the I/O mapping of the process.

The new implementation consists of an application written in C that can be converted and used by exploiting the vulnerability mentioned above. The application can use `/dev/mem`, `sysfs`, or a legitimate driver call to access and configure the pins. In our target platform the CODESYS runtime uses the `/dev/mem` for I/O access, therefore, our attack code use the same I/O interface.

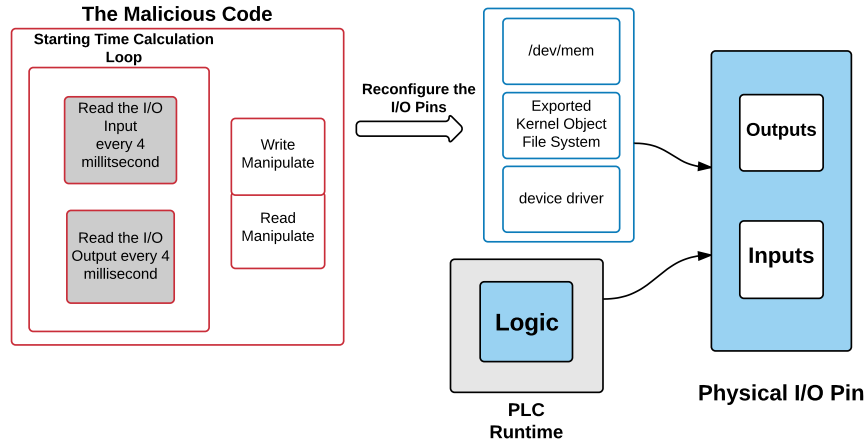


Figure 10: Overview of pin configuration attack without root access using the exported kernel objects, `/dev/mem` and a legitimate device driver

Figure 10 depicts the steps our application takes to execute the attack. The application first checks whether the processor I/O configuration addresses are mapped in the PLC runtime. The list of all mapped addresses is system wide available in various locations for any user space application (e.g., via `/proc/modules`, or `/proc/$pid/maps`). If at any extraordinary circumstance the physical I/O addresses are not mapped, our application can map the I/O base address of our target platform at `0x20200000`.

For manipulating write operations, the application needs to know a reference starting time. This is the relative time where the PLC runtime writes to the pin. While the application knows the logic and is aware that every five seconds there is a write operation to pin 22, it does not know at what second the last write operation happened. This can be easily found by monitoring the value of pin 22. Once the application intercepts the correct reference starting time, for every write operation in the logic it will carry out two tasks. First, right before the reference starting time (which is when the PLC runtime will start writing its desired original value to the I/O) the application reconfigures the pin to input mode. The CODESYS runtime then attempts to write to the pin. However, the write operation will be ineffective, since the pin mode is set to input. Our application then switches the pin mode to output and writes the desired value to it.

For manipulating read operations, the application changes the state of the pin from input to output and writes to it constantly with the desired value.

With this implementation we could successfully manipulate the process. The LED would turn on and off every ten seconds instead of five. Additionally, we could completely control input pin 24 and make its value 0 or 1 whenever we wanted, while the CODESYS runtime was reading our desired value.

This implementation is significantly more lightweight than our previous one, and only causes a two percent CPU overhead without any differences between read and write operations.

There is however a small chance that a race condition happens during read manipulation. For example, assume that we have a sensor connected to an input enabled pin in the PLC. If this sensor updates the value of the pin right after our application does and the PLC runtime reads the value right after this happen, the actual value will be reported instead of the attacker’s intended value. In our tests, however this race condition never happened.

## 7.5 Attack Implementation in a Real World PLC

As done for Raspberry Pi, we analyzed a real PLC and considered the attack possibilities on this system as well. The provided PLC is a *PFC200 750-8202* model from the *Wago* vendor. It runs a Linux kernel with the RT-preempt patch, which gives it hard real-time capabilities. In particular, the system runs the following kernel version on an ARMv7 architecture:

```
Linux PFC200-4106BA 3.18.13-pfcxxx-02.00.02_00+14-rt10 #1 PREEMPT RT armv7l GNU/Linux
```

Furthermore, Wago gives complete (root/kernel) access to its system, and the user can even program its own PLC runtime [10]. In this PLC, different kinds of I/O modules can be attached to the PLC. For our analysis, we attached a Digital I/O module to the communication bus. PLC and I/O module are based on an *AM3517* SoC from *Texas Instruments* and an *XE164* SoC from *Infineon Technologies*, respectively. The digital I/O module has 16 different pins, 8 for input and 8 for output, and we connected a button as input and an LED as output, obtaining the system shown in Fig. 11.

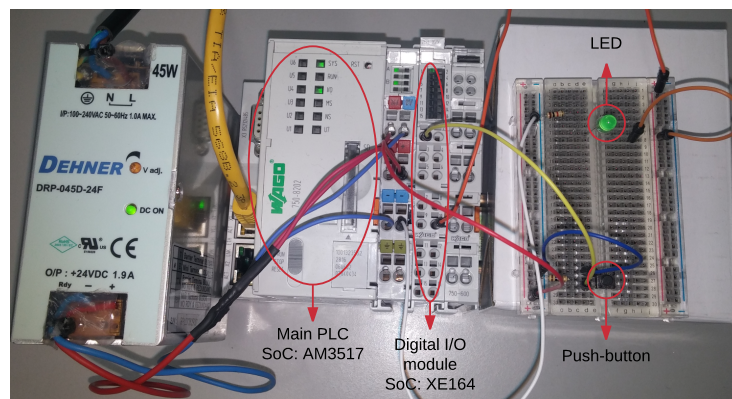


Figure 11: Wago PLC system used for experiments

The PLC runtime environment is the *e!RUNTIME* provided by Wago, which is based on CODESYS. From the Wago *e!COCKPIT* engineering software, we programmed a simple logic which toggles the value of the LED every 500 ms

only when the button is not pressed (high input). If the button is pressed, the LED is turned off until the button is released. The logic, in *Structured Text* (ST) language, is shown in Fig. 12. It assumes a scan cycle of 10 ms, and uses a software counter to achieve the timing of 500 ms. The picture also shows the mapping between the variables and the I/O pins of the external module. Differently from the Raspberry Pi system behavior, here the outputs are updated on every scan cycle, even if the value has not been changed during the last 10 ms. Given this system, we started the analysis of the PLC runtime to better understand the overall architecture.

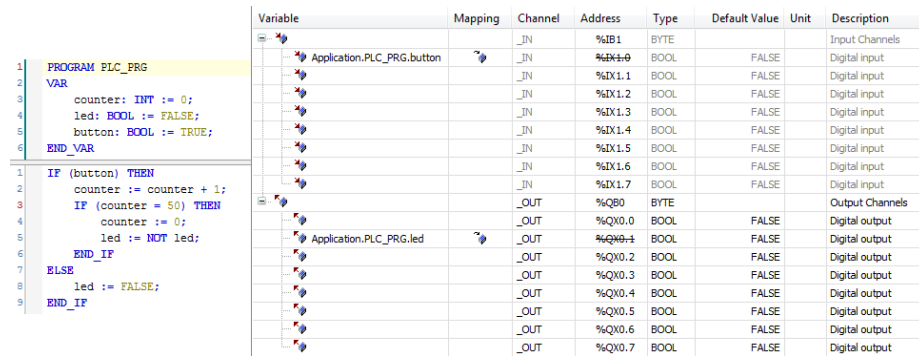


Figure 12: PLC logic loaded into Wago PLC

### 7.5.1 PLC e!RUNTIME analysis

The Wago implementation uses a communication bus between PLC and I/O named *KBUS*. It is implemented as a kernel driver, whose code is available as kernel patches included into the Board Support Package (BSP). The KBUS is basically a serial BUS built upon the Serial Peripheral Interface (SPI) protocol, used together with Direct Memory Access (DMA) to perform faster transfers. The PLC runtime uses the driver through `ioctl` calls to send and receive inputs and outputs at each scan cycle, as reported below.

```

[... ]
04:10:01.545185 clock_gettime(CLOCK_MONOTONIC, {122, 590121662}) = 0
04:10:01.545831 clock_gettime(CLOCK_REALTIME, {1335924601, 545923773}) = 0
04:10:01.546293 ioctl(11, _IOC(_IOC_WRITE, 0x4b, 0x01, 0x18), 0xb54a47d0) = 4
04:10:01.549585 clock_gettime(CLOCK_REALTIME, {1335924601, 549739477}) = 0
04:10:01.550047 clock_gettime(CLOCK_MONOTONIC, {122, 595045151}) = 0
04:10:01.550508 nanosleep({0, 3785000}, NULL) = 0
[... ]

```

The code shows one PLC scan cycle, which is basically made of an `ioctl` call, plus the corresponding calls needed to achieve the timing of 10 ms (`clock_gettime` and `nanosleep` functions). Each `ioctl` call from user space is directed to the internal `kbus_ioctl` function of the KBUS driver.

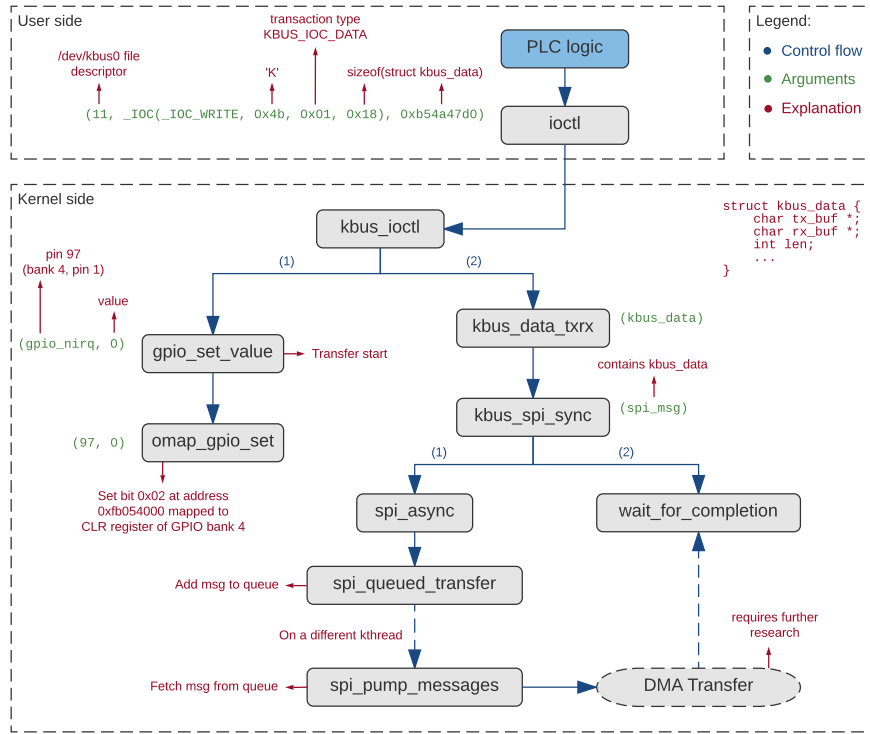


Figure 13: PLC I/O operations executed for each scan cycle

The operations performed at each scan cycle are outlined in the scheme of Fig. 13, which shows the results of our reverse engineering analysis on Wago PLC e!RUNTIME. From a quick look at the chart, we immediately deduce that the complexity is much bigger with respect to the CODESYS runtime for Raspberry Pi. The important difference between CODESYS and e!RUNTIME is that the former performs I/O by directly mapping physical addresses and reading from (or writing to) I/O pins, while the latter has to transfer data to the external I/O module. Thus, e!RUNTIME uses its I/O pins to communicate with I/O modules, by means of the KBUS driver interface.

KBUS is an abstraction layer built upon different lower-level drivers: basically GPIO, SPI and DMA. The main operation of the KBUS driver is managed through the `kbus_ioctl` call. The first argument (11) is the file descriptor number related to the `/dev/kbus0` file, which allows the kernel to forward the request to the KBUS driver. The second argument contains different information used by the driver to select the specific sub-function. In our case, the request is for a data transfer, and the third argument (`0xb54a47d0`) is a pointer to a `struct kbus_data`. The data structure contains, among other members: a pointer to a transmit buffer (`tx_buf`), a pointer to a receive buffer (`rx_buf`)

and a buffer length. Since `tx_buf` and `rx_buf` are actually pointing to the same buffer used for both directions, only one length is required. In our configuration, with the digital I/O module alone, the buffer contains at least 2 bytes: one for input pins and one for output pins (8 bits each). Thus, every `ioctl` call serves both as read and write operation. On each call, the input part of the buffer is filled in by the KBUS driver, while the output related to the previous scan cycle is written to the bus. From the new input obtained, the logic computes new output values, waits for the necessary time, and calls `ioctl` again with the updated buffer. We report below the output of one of our kprobes to show the actual content of the buffer during a call to `ioctl`:

```
ioctl: len = 2, bytes = [0x40, 0x00]
```

The first byte indicates that the PLC logic is sending a 1 to the second output pin (`0x40`), to turn the LED on. The second byte represents the input value related to the previous scan cycle, which will be overwritten by the KBUS driver during the current call. In the above case, `0x00` means that all the digital input lines were low.

Each I/O operation begins with a call to `gpio_set_value`, which writes on a GPIO pin, indicating to the I/O module that a transfer is starting. The pin used for this purpose is the GPIO pin 97, i.e. pin 1 of GPIO bank 4 (each bank controls 32 pins). The value written into the pin is 0, since the signal is active low. Afterwards, the data buffer is encapsulated into an SPI message, starting an SPI transfer. The SPI interface leverages four pins of the AM3517 SoC, two for data input and output (MOSI: Master Output Slave Input, and MISO: Master Input Slave Output), one for synchronization (SCL: Serial CLock) and one for chip select (SS: Slave Select). The PLC SoC is the SPI master, and, according to our actual configuration, the digital I/O module is the only SPI slave available. The transfer is delegated to the DMA controller, while the current thread, which corresponds to the PLC runtime thread, goes to sleep into a wait condition (`wait_for_completion`). Each DMA transfer uses an interrupt to signal when it has completed, or if some error occurred. Further analysis is required to understand the behavior of a DMA transfer and its corresponding IRQ handler, but that goes beyond the purpose of this report. Our best guess is that the DMA handler resets the initial GPIO pin and wakes up the PLC runtime thread.

Another interesting feature of the KBUS driver is that it exposes a `write` operation to the user, through which it is possible to enable/disable the above interrupt. The `e!RUNTIME` makes use of this system call to disable the interrupt while a new logic gets uploaded into the PLC, and then re-enables it before starting the logic. If a new logic comes, together with a new I/O configuration, the interrupt gets disabled because the runtime needs to reprogram the I/O module and the KBUS interface according to the new requirements.

## 7.6 Suggestions on Detecting Pin Control Attack

While the pin control attack we implemented was stealth to the CODESYS runtime, one might believe that it is relatively simple to devise countermeasures that could detect and possibly block pin control attacks. In this section we enumerate five of these possible countermeasures and discuss their effectiveness and practical applicability to embedded systems.

1. *Monitoring the mapping of pin configuration registers:* an attacker needs to use the virtual addresses of the pin configuration registers to write to them. To do so, the attacker needs to either map the physical registers by herself or use already mapped addresses. In the first case, one could monitor the mapping of pin configuration registers by hooking critical mapping functions such as `mmap()` or `ioctl()`. However, an attacker could implement her own version of `mmap()` or `ioctl()`, thus bypassing the monitoring point. Additionally, in an embedded device that is already using the target I/O, an attacker can use the pin register address already mapped by the application or kernel.
2. *Monitoring the change of pin configuration registers:* one may detect our attack by monitoring the frequency at which pin configuration registers are changed. This may be challenging for two reasons. First, since changes in configuration registers do not generate interrupts, an attacker could be able to bypass monitoring mechanisms. For example, in order to avoid performance overhead, the value of configuration changes which must be checked in a loop, could be monitored with a certain frequency (e.g., every second). This would give the attacker a window of opportunity to modify the configuration within the checking window. Second, since pins get re-configured legitimately, it may be difficult to tell with reliable accuracy whether a sequence of changes is legitimate or not.
3. *Monitoring the use of debug registers:* one could argue that the usage of debug registers in our first implementation of the attack is something that can be easily detected. For example, by monitoring the processor debug registers one could detect the point in our first attack implementation in which we set a breakpoint to the access of the I/O registers. However, in our experiments we noticed that constantly monitoring all processor debug registers (i.e., with a loop) can cause a non-negligible CPU overhead. This makes this approach not very attractive for embedded systems, which are always resource constrained.
4. *Monitoring performance overhead:* because of the performance overhead imposed by our first implementation of the attack, we could employ approaches based on monitoring the power consumption of embedded systems, such as those proposed in [28] to detect the attack. However, how our second implementation demonstrates, other approaches can impose a significantly smaller CPU overhead, which would probably go unnoticed. In addition, other pin control attacks presented in Section 6 just require

a single configuration or multiplexing operation, with practically no CPU overhead.

5. *Using a trusted execution environment*: The reliable solution to prevent all pin control attacks would be running a micro kernel in a trusted zones (e.g., an ARM TrustZone) within the kernel and verifying the write operations to the pin configuration registers with a dynamic key. However, as confirmed by the Linux Kernel Pin Control Subsystem group, using an ARM TrustZone for I/O operations would cause a significant performance overhead.

## 8 Pin Control Attack Detection

In this section we will discuss a mechanism for detecting Pin Control Attack. We describe the design of the protection system and then we will describe the implementation detail of our protection mechanism.

We assume that the PLC system, and our defense as well, is protected by the previously discussed HIDSeS. Starting from the analysis reported in 6, we identify the I/O configuration as the main resource we aim to protect. To target I/O configuration, the attacker may use other system capabilities, such as virtual address mapping and debug registers. First of all, we define the following components of a SoC, to which we refer several times in the rest of the report:

- **I/O subsystem**: the subsystem that controls the I/O configuration. I/O configuration is defined as the set of all the control registers actively used by the SoC, whose unauthorized modification may have direct or indirect effects on the controlled process. In particular, we are interested into pin control registers, which directly determine the behavior of the SoC I/O pins. However, a SoC typically contains many devices and controllers which may be in charge of a subset of I/O pins. In other words, some pins can be multiplexed to specific devices inside the SoC. Since these devices are programmed through their own control registers, altering these registers may indirectly affect I/O pins as well. Thus, both pin control registers and device control registers are considered as part of the I/O configuration. The attacker who has knowledge of the system and its I/O peripherals may access all these registers to alter the physical process.
- **Debug subsystem**: the SoC subsystem that enables debug capabilities for the operating system and its processes. Typically, it consists of a set of registers, called *debug registers*, inside the processor of the SoC. The operating system may provide an interface to access them, both for kernel side and user side. The attacker may leverage the debug subsystem to obtain accurate timing information and conduct more sophisticated attacks, either using the interface provided by the operating system or the low-level processor instructions directly (depending on its privilege level).

- **Mapping subsystem:** the system that manages mappings between physical and virtual addresses, both for kernel and user space. It is typically supported by the Memory Management Unit (MMU) in hardware, and by the operating system in software. Within the context of a PLC, the runtime may use this subsystem to configure the I/O and to perform read/write operations. An attacker can access physical I/O addresses (thus, I/O configuration) either by requesting a new mapping to the system or by using an already existing mapping.

Given the architecture of Pin Control Subsystem, protecting I/O configuration is not straightforward because the hardware lacks any protection mechanism related to the I/O subsystem.

Thus, we need to define an alternative approach that does not have the above limitations: it must be easily applicable and have a minimal overhead. Since the hardware-based solution is unpractical, we analyzed the possible software-based solutions. In order to choose the best strategy against Pin Control Attack, we compared the following approaches, as proposed in Section 7.6:

1. monitoring I/O configuration to detect changes;
2. monitoring the use of debug registers;
3. monitoring the mapping requests targeting I/O configuration;

These approaches are not mutually exclusive, and may be used simultaneously to get a higher protection level. Before describing our design phase, we briefly discuss two further protections in the following sections.

## 8.1 Defense Architecture

Based on the previous considerations, we designed a detection system which is able to protect against Pin Control Attack at three different levels, corresponding to the SoC I/O, debug and map subsystems. With respect to the attack name, we refer to our detection system as *Pin Control Defense*. The system is designed to run as part of the OS, thus having kernel privilege level. Its overall architecture is shown in Figure 14. Considering that the attacker can possibly follow any of the paths highlighted in red in the figure, our monitoring system has been divided into three main components:

- **I/O monitor:** its purpose is to watch the I/O configuration, detect and react to any malicious change.
- **Debug monitor:** it aims to protect the debug subsystem from malicious usage.
- **Map monitor:** it acts as a filter for mapping requests targeting I/O memory.

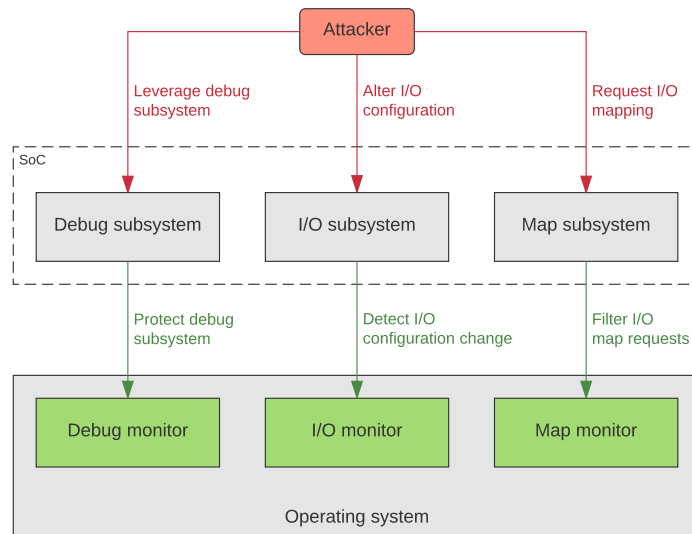


Figure 14: Pin Control Defense general architecture

Each monitor is responsible for reporting detection information related to any interesting event of the respective subsystem, and reacting to these events according to its own configuration. The events are not necessarily due to Pin Control Attack, e.g., an I/O configuration change event may be caused by the PLC runtime. Therefore, the I/O monitor should decide whether a particular I/O modification can be considered legitimate or not. Debug and map monitors, instead, serve at least the following purposes:

- **early detection:** the attack can be detected before it modifies the I/O configuration;
- **raising the bar:** they limit the attacker possibilities, by restricting the access to the corresponding subsystem;
- **reporting info:** they provide additional information, useful to figure out how an eventual attack has been conducted.

Since these modules are designed to be customizable, the actual effects depends on their configuration. The following sections describe in more detail the design and the role of each module.

### 8.1.1 I/O monitor

The I/O main task is to cyclically check the values contained into I/O configuration registers, to verify that they are conforming with the logic currently loaded into the PLC. If a change into a target register has been detected, it determines

whether this modification is legal or not, according to a given trusted behavior. Note that a new I/O configuration may come with a new PLC logic, and the PLC runtime must be able to apply the change without having our defense to interfere. Therefore, an automated mechanism able to distinguish between trusted and malicious configurations is needed. This is not an easy problem, because an optimal solution would require an authentication between PLC runtime and I/O subsystem, and this is not reasonable in our highly constrained system. To tackle this problem, we designed the following strategy, which we used to implement the I/O monitor:

1. define I/O configuration registers;
2. define a trusted behavioral model of the I/O configuration;
3. constantly monitor the I/O configuration to detect possible modifications;
4. if a change is detected, verify whether it is conforming to the defined behavior or not;
5. if it is, accept the new configuration and go back to 3;
6. otherwise, it is probably I/O Attack: react according to the configured monitor action and go back to 3.

The first step is to choose which registers should be included into the I/O configuration, i.e. which registers should be protected. This set of registers includes, ideally, all the registers whose modification may produce an effect on the physical process. However, it is not always possible to define a behavioral model or enable the protection for each register. For instance, some registers may be write-only (e.g., pull-up/down registers), and there is no way to verify their current value. Whether to include or not a register into I/O configuration should be decided case by case, according to the protection feasibility and the possible effects of a malicious modification.

To define the behavioral model, it is required to determine the set of configuration registers actively used into the target system, i.e. the set of registers that may affect the physical process. Then, for each register (or for each bit of each register if necessary), the model should define under which conditions the corresponding value may change. These conditions are highly dependent on the target implementation. Generally speaking, they can be represented by simple time constraints (e.g., the value cannot change twice within 20 ms), logical conditions (e.g., the value must be conforming to the running PLC logic), a statistical model, or a combination of these. Each condition can either provide an exact distinction between a trustworthy and a malicious modification (typically logical conditions), or a heuristic only. For this purpose, we analyze in more detail the proposed logical condition, since it is able to completely exclude false positives. The condition states that a change can be accepted only if the new I/O configuration is in line with the operations performed by the running PLC logic. Checking this condition is feasible as long as enough knowledge of the PLC runtime is available. To obtain this knowledge, two ways are doable:

- **reverse engineering:** by analyzing the PLC runtime it is possible to dynamically obtain the required information from the running logic;
- **PLC runtime vendor collaboration:** if the PLC runtime is designed to be aware of the defense mechanism, it could better expose the required information to the I/O monitor at run-time, thus removing the need for reverse engineering.

Once the I/O monitor is able to determine which operations the PLC logic is carrying on, it can easily decide if an I/O configuration is malicious or not, and can effectively detect Pin Control Attack. Note that, if the information on the current I/O operations is made available by the PLC runtime, the attacker can modify it as well while conducting the attack. In this case, however, the attacker needs to modify the PLC logic, and this may be easily noticed if further protection mechanisms are provided by the PLC runtime. Therefore, the attack would lose one of the features that made it stealth. Moreover, from our experiments, we found that the PLC programming software typically does not perform any check about the I/O configuration. For instance, it is possible to upload a new PLC logic together with an I/O configuration which is in conflict with the logic itself. Thus, our approach could also be useful to detect not only Pin Control Attack, but any conflict between the actual I/O configuration and the expected behavior of the logic.

An important parameter to discuss is the time interval of the main monitor loop. Since no other mechanisms are provided by the hardware, such as interrupts, we can only detect I/O configuration changes by cyclically checking its current values. Greater scanning intervals may give the attacker enough time-window to reach its purpose. For instance, given a PLC scan cycle of 10 ms and a monitor interval of 50 ms, if the attack is able to synchronize itself with the monitor loop, it has enough time to alter  $\lfloor \frac{50 \text{ ms}}{10 \text{ ms}} \rfloor = 5$  consecutive I/O operations and then restore the configuration back before getting noticed. Smaller intervals, instead, may cause too much performance overhead.

Another challenging aspect of this approach is to decide which action the monitor should follow if an attack has been detected. We distinguished at least three main reactions for I/O monitor:

- report the event to the system;
- revert the configuration back to the last known before the attack;
- stop the control process.

The choice among these actions (or a combination of them) again depends on the target implementation. Typically, reporting the event is the minimum that the monitor can do, while other reactions should be decided according to risk associated with the physical process and to the monitor reliability. If a detection is proven to be correct, due to an exact condition, then reverting the configuration back could be the best choice. Otherwise, if the detection condition is

a heuristic and the risk is critical, stopping the control process may be considered as a more cautious alternative. In general, if a monitor only reports about events we call it *passive*, otherwise it is an *active* monitor.

### 8.1.2 Debug monitor

This monitor is responsible for protecting the debug subsystem from malicious usage. Debug registers may be used by the attacker to gain accurate timing information about the PLC logic I/O operations. Similarly to what the I/O monitor does, the debug monitor continuously watches the values contained into debug registers to detect malicious modifications. In our design, we assumed that there is no need to use debug registers into the target PLC if it is already deployed into a real control system. As confirmed by our experiments, they are actually never used. Typically, the SoC debug subsystem may only be needed by PLC vendors during design and implementation of their own product. Since the operating system may provide user level access to debug registers (e.g., `ptrace` API on Linux), we can simply disable this user interface. However, there is no mechanism to permanently disable debug registers also at kernel-wide level without kernel recompilation.

We designed the following monitor strategy, that is required only if debug support is enabled into the OS:

1. disable debug registers user space interface;
2. constantly monitor debug registers to detect possible modifications (from kernel space);
3. if a change is detected, it is Pin Control Attack: react according to the configured monitor action and go back to 2.

The strategy is (in part) a simplification of the I/O monitor approach, on which the trusted behavioral model assumes that debug registers never change in a production system. Based on this assumption, the debug module allows our defense to provide an early detection of the attack, before I/O configuration is actually altered. The discussion about the monitor time interval is the same as for the previous I/O monitor: a trade-off between attacker time-window and monitor overhead.

When an attack has been detected, restoring the previous values of debug registers is surely the best action to take, because the assumption ensures that only malicious changes may occur. Halting the PLC process, instead, is certainly not needed because the control process is not directly affected by a modification of the debug subsystem. In any case, the event is reported to the system. To uniform the design, the debug monitor can be configured either as passive or active, although the passive mode is strongly discouraged for the above reasons. If the monitor is active, it actually raises the bar for the attacker, who cannot leverage debug registers anymore.

### 8.1.3 Mapping monitor

The attacker may either request a new mapping between physical and virtual memory, or re-use an existing one before modifying I/O configuration. The map monitor leverages this fact, providing a further detection mechanism usable in combination with the previous two. Typically, the operating system provides an interface, for user space, through which each process can map a physical address region to a corresponding virtual region. In the following part of this document we refer to it as “mapping interface”. The actual mapping is performed inside the kernel, and the process receives a valid virtual address as result. The attacker can leverage this mechanism to gain access to the I/O configuration registers from user space. According to the implementation, the PLC runtime may use this mechanism as well, and if it does, the attacker may try to re-use the PLC runtime virtual address.

Thus, the aim of this monitor is only to monitor *new* mapping requests. To achieve the goal, we dynamically replace the functions belonging to the mapping interface with our own versions (hook). We can summarize the strategy of the map monitor as follows:

1. define a trusted behavioral model for I/O mapping requests of the PLC runtime;
2. hook all the functions belonging to the mapping interface;
3. at each new mapping request, verify whether the requested physical address range overlaps the I/O configuration region or not;
4. if there is no overlap, forward the request to the original system function;
5. if an overlapping region is detected, verify whether the request is conforming to the trusted behavior or not;
6. if it is, forward the request to the original system function;
7. if it is not, it is probably I/O Attack: react according to the configured monitor action.

The behavioral model of step 1 should describe if and how the mapping interface is used by the PLC runtime. We analyzed the behavior of our target systems, with the following results:

- Raspberry Pi: the CODESYS runtime un-maps and re-maps the I/O every time a new PLC logic is uploaded;
- Wago PLC: e!RUNTIME never maps physical I/O from user space, because it is managed by the system driver.

The next step of the strategy, function hooking, must satisfy at least the following requirements:

- it must be efficient: mapping functions may be called many times by processes (e.g., in Linux, they are used not only to map physical memory, but any file, device, etc.);
- considering the threat model discussed in 6.1, it must be applied before the Autoscopy Jr. detection system is deployed; otherwise, our modification will be considered as malicious.

Finally, the reaction of the map monitor to an eventual detection depends on the PLC runtime behavior. If the PLC runtime maps the I/O (as in Raspberry Pi), then a mechanism to distinguish between good and malicious requests is needed. To accomplish this, we may list the following alternatives:

- heuristic approach based on statistical data;
- integration of the defense with the PLC runtime.

Since the first approach cannot give an exact detection, the monitor may simply report the detection to the system. The second approach, instead, may be implemented in different ways. For instance, the map monitor could provide a separate mapping interface reserved only for the PLC runtime process. In any case, if the system allows to have an exact detection mechanism, the monitor may directly deny the malicious request, factually raising the bar for the attacker.

## 9 Pin Control Attack Detection Implementation

In this section we describe our implementation for detecting Pin Control Attack named as *Ghostbuster*. *Ghostbuster* is a kernel module written in C language, targeting Embedded Linux running on ARM architecture. It is designed to be highly configurable and as architecture-independent as possible, following the guideline of the Linux kernel itself. A portion of the code, i.e. the lowest level code, is still dependent from the specific architecture (e.g., ARM), but is separated from the general implementation template. This allows *Ghostbuster* to get extended to other architectures and SoCs running Linux. At the same time, we focused on maintaining the lowest overhead possible, which is always crucial for PLCs. We proceed with the description of the overall architecture, and then we go deep into each module of the architecture. Finally, we describe the usage of our kernel module.

### 9.1 Implementation architecture

The implementation architecture is based on the general one described in 8.1. Thus, we implemented I/O, debug and map monitors.

The architecture of *Ghostbuster* is shown in Fig. 15, which associates each specific role to the corresponding source file(s) (in red). Furthermore, the compilation of our module can be parameterized in different ways, as shown by

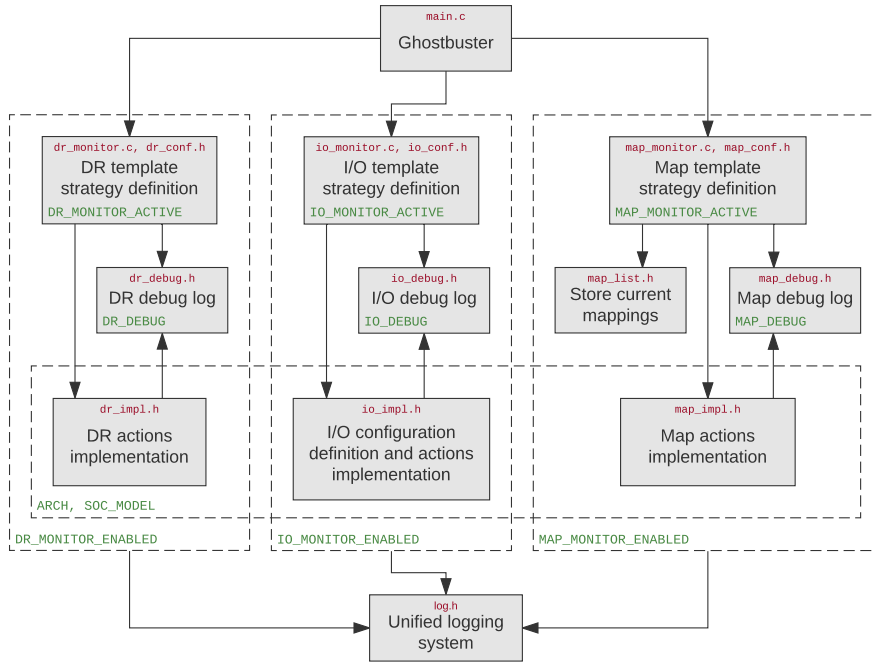


Figure 15: Ghostbuster implementation architecture

the green labels in the figure. Each green label, associated to a rectangle container in the figure, corresponds to a compiler flag or parameter that affects the compilation of the enclosed portion, either enabling/disabling the component or changing its default behavior. The compilation is managed by a `Makefile`, in which all the required flags are defined and passed to the compiler. The `Makefile` needs to know the location of the target Linux kernel source directory, in order to link Ghostbuster with the kernel code.

When Ghostbuster is loaded, the three main monitors are started in parallel. The monitors are independent from each other, and every monitor has its own kernel thread. When a monitor has to report an event to the system, it uses the common logging subsystem available into the kernel. The target systems used to test our defense are exactly the same as the ones described for the attack part in 6, with the same PLC logic and I/O configuration. In particular, for each monitor, we first refer to the Raspberry Pi system (BCM2835 SoC); then we discuss the modifications required to run on Wago PLC as well. In the following sections we describe in more detail the structure and the operations performed by each monitor. From now on, we will refer to the abstract part of each monitor as the *interface*, and to the architecture-dependent part as the *implementation*.

## 9.2 I/O monitor

This monitor is responsible for protecting I/O configuration memory from malicious usage. With reference to the BCM2835 manual [22], we defined, into the implementation part, the set of I/O registers we want to protect. In particular, since our system uses GPIO pins, two of which multiplexed to I2C, we chose to protect GPIO pin control registers, whose physical base address is 0x2020000. These registers are responsible both for pin configuration and pin multiplexing. Each register is 32-bit wide, having 3 bits for each I/O pin: it can control 10 different pins. Although our target system uses only four pins for its I/O, to be more general, we decided to protect all the available GPIO pins (54). Hence, we included all 6 pin control registers into the definition of our I/O configuration.

The next step is to define the trusted behavior of the I/O configuration. In particular, we have pin multiplexing (*pinmux*) and pin configuration (*pinconf*) registers, and we assumed the following behavior:

- **pinmux registers:** they are initialized at boot time, and never change during run-time;
- **pinconf registers:** they are initialized at boot time, but can be modified at any time by the PLC runtime, maintaining the following invariant: every pin configuration (input or output mode) must be conforming to the running PLC logic.

After I/O configuration has been defined, we can describe the interface and implementation of the I/O monitor. The abstract part is essentially made of the monitor loop, which executes the main strategy, and the detection handler, called by the implementation when an I/O modification has been detected.

Each iteration of the main loop is executed every  $t$  ms. The loop terminates only if an external signal indicates that Ghostbuster is being stopped. The verification of the current I/O configuration is performed at block level against the golden reference, which is obtained by getting current IO state before starting the loop. Since we assume that the system is in a safe state when our module is deployed, the initial I/O state read from configuration registers is considered as trusted. When a modification is detected by our monitor, the detection handler is called back and the monitor strategy is applied. Note that the handler may be called many times because there may be more than one single modification within one contiguous block. The detection granularity is decided by the implementation part (e.g., for each single pin in our implementation). If the modification is considered legitimate, then the reference I/O state is updated with the new configuration. Otherwise, we report the attack to the system and, since we have an exact detection condition which cannot give false positives, we may safely restore the previous configuration back if the I/O monitor is in active mode.

All the low-level I/O functions are defined into the implementation part, and may be different for each target system.

The detection events are managed at pin level, i.e. if an attacker modifies the configuration of more than one pin at a time, only one pin at a time is verified. The critical function of the implementation part is when Ghostbuster decides whether a configuration change is legal or not. Based on our behavioral model, pin multiplexing can never be legitimate during run-time, so the implementation is straightforward. In the case of pin configuration, instead, the implementation needs to check if the I/O configuration after the change is in conflict with the PLC logic. A conflict occurs in one of the following two cases:

- **write**: the pin is set as input and the logic is trying to write from it;
- **read**: the pin is set as output and the logic is trying to read from it.

The most challenging problem here is to figure out which operation the logic is performing on a given I/O pin. To solve this problem, we applied the reverse engineering approach proposed in 8.1.1.

Inside the BCM2835, a specific set of 32-bit registers is used to interact with I/O pins: LVL registers to read the pin value, CLR registers to write a 0 and SET registers to write a 1 [22]. Every register contains a bit for each pin, for a total of 32 pins per register. To have access to the operations performed by the PLC logic on these registers, we leveraged the debug subsystem. In ARM architecture, the debug subsystem provides two different types of debug registers: breakpoint and watchpoint (see 9.3). When a pin configuration change is detected, the I/O monitor inserts a watchpoint to the corresponding LVL, CLR or SET register of the affected pin, in order to intercept the I/O operation performed by the PLC logic. The watchpoint can be set for either a read or a write, according to the specific case we want to verify. From a reverse engineering analysis of the PLC runtime, we found that read and write operations are performed with the following instructions, respectively:

- **STR R2, [R3]** (Opcode 0x002083E5, R3 contains the address of a STR or CLR register);
- **LDR R2, [R3]** (Opcode 0x002093E5, R3 contains the address of a LVL register).

R3 always contains the virtual address targeted by our watchpoint. When a watchpoint is hit, the debug exception handler is called, and all the execution context of the process is passed as argument. Inside the debug handler, we proceed as follows, according to the case:

- **write**: we look at the content of R2 to check if the PLC logic is trying to write to the specific pin changed to input. If R2 contains a 1 corresponding to the target pin, then we can conclude that the new configuration is in conflict with the logic.
- **read**: in this case more reverse engineering of the PLC runtime is needed to know which pins (i.e. bits) are actually used by the logic as input, because the instruction always loads the entire register (32 pins). To

simplify the detection in our prototype version, we supposed that the PLC logic may only have one input pin for each register. In this case, reading from the register implies that the PLC logic is trying to read from the given pin. Thus, we can report a conflict. To remove this assumption, which limits the applicability of our defense, more knowledge about the PLC runtime is required. As previously discussed, this knowledge can be obtained either from a deeper reverse engineering analysis or from a collaboration with the PLC runtime vendor. For instance, each PLC logic may be designed to contain a constant bit mask having one bit for each pin, where each bit specifies whether the corresponding pin is currently used as input or output. If such data was available, the I/O monitor could look at it instead of inserting a watchpoint and looking for the actual operations. This mechanism raise the bar for the attacker, who would need to alter the PLC logic code as well to conduct the attack, thus defeating its stealthiness.

Note that from a performance perspective our approach is feasible, because it inserts a debug exception only in case of an I/O modification to determine if it is malicious. Thus, this may cause an overhead when a good I/O configuration is updated by the PLC runtime as well, which only occurs when a new logic gets uploaded to the PLC.

Besides its main functions, the I/O monitor provides an additional interface, which is needed by the MAP monitor. When the MAP monitor has to filter mapping requests coming from system processes, it needs to know whether a request of a physical address range includes at least one address belonging to the I/O configuration. This is provided by the I/O monitor, which accepts a pair of physical addresses representing the start and the end addresses of the given range. Since this functionality is needed by the MAP monitor independently from I/O monitor operations, we designed it to be available even if the I/O monitor is disabled. Note that, there might exist some target systems in which disabling the I/O monitor has sense. For instance, consider a system having the following two properties:

- the root/kernel access is properly protected (e.g., kernel protected by execute only memory or TrustZone), or it has sufficient security level for which is not worth to insert the I/O monitor (e.g., no admin default passwords, hardened kernel, etc.);
- the PLC runtime, or any other process, does not use the mapping interface (i.e. on Wago PLC).

On such a system, the MAP monitor alone could be enough to protect the I/O configuration, because it can simply deny any access to I/O physical addresses from user space, while the system itself is already protected from kernel space accesses.

### 9.2.1 Wago PLC version

To port the I/O monitor to the Wago PLC system, only the following changes are necessary:

- **I/O configuration:** the I/O configuration and the behavioral model must be redefined according to the registers used by the Wago PLC. In particular, the implementation may include pin configuration and pin multiplexing registers as well, plus SPI, DMA and IRQ registers. The way to access registers is pretty much equal to the Raspberry Pi system, because are both ARM architectures with 32-bit wide registers. On Wago PLC, pin configuration and pin multiplexing are managed by two different set of registers. Therefore, the low-level implementation is even simplified, because it does not need to distinguish configuration and multiplexing bits inside registers. However, including other registers (e.g., SPI, DMA, IRQ, etc.) may lead to more complex behavioral models which need to be analyzed.
- **Debug subsystem:** given the engineering problem described later in 9.3, the same solution using watchpoint cannot be directly applied. To get the needed information about the PLC logic, two ways are possible. Either the debug interface is implemented into the kernel for AM3517 SoC, or a better integration with the PLC runtime is required, as already discussed.

## 9.3 DR monitor

The DR monitor aims to protect the debug subsystem. As described for the design phase, the DR monitor needs to accomplish two goals:

- disable debug interface access from user space;
- watch over debug registers value to detect malicious events.

In Linux, the debug user interface is based on the following functions [40]:

```
struct perf_event* register_user_hw_breakpoint(struct
    perf_event_attr* attr,
    perf_overflow_handler_t handler,
    struct task_struct* tsk);

int modify_user_hw_breakpoint(struct perf_event *bp,
    struct perf_event_attr *attr);

void unregister_hw_breakpoint(struct perf_event *bp);
```

To disable userland access to these functions, we dynamically patched the kernel text replacing the prologue instructions of each function.

Once the user access to debug registers has been disabled, we activate the DR monitor to protect them from attackers who gain kernel access.

The ARM architecture supports two type of hardware debug registers: breakpoints for instruction addresses, and watchpoints for data addresses. The DR

monitor protects both of them, and the detection is handled at register level. Each register is accessed through specific co-processor instructions. A special read-only register, the *Debug ID Register* (DIDR), specifies the number of debug registers available and other useful information on the SoC configuration. In particular, since the BCM2835 SoC supports 2 watchpoints and 6 breakpoints, the DR monitor can be deployed to protect them all.

### 9.3.1 Wago PLC version

The AM3517 SoC model embedded into this PLC provides a different interface for debug registers. In particular, it uses a memory mapped interface instead of specific co-processor instructions [32]. Unfortunately, it turns out that this particular interface is not supported by the Linux kernel debug framework, as reported in [63]: “The memory-mapped extended debug interface is unsupported due to its unreliability in real implementations”. Apart from this engineering issue, the DR monitor is designed to be applicable to any other ARM implementation that supports debug registers. To port it for other architectures different than ARM, only the implementation part must be adapted.

## 9.4 MAP monitor

The purpose of the MAP monitor is to filter user space requests delivered to the mapping subsystem. On Linux, users can use the mapping interface to map files or devices, and this mechanism can be used to map physical addresses to virtual addresses through special devices such as `/dev/mem`. Among all physical memory addresses, an attacker may be able to map specific I/O memory regions to alter I/O configuration registers. The set of devices that can provide access to I/O physical memory depends on the specific system (e.g., for the Linux version installed on our Raspberry Pi, either `/dev/mem` or `/dev/gpiomem` can be used). Linux provides the following system calls to deal with mapping requests:

```
long mmap2(unsigned long addr, unsigned long len,
           unsigned long prot, unsigned long flags,
           unsigned long fd, unsigned long pgoff);

long mremap(unsigned long addr,
            unsigned long old_len, unsigned long new_len,
            unsigned long flags, unsigned long new_addr);

long remap_file_pages(unsigned long addr, unsigned long len,
                     unsigned long prot, unsigned long pgoff,
                     unsigned long flags);

long munmap(unsigned long addr, size_t len);
```

Note that we are describing the interface from a kernel point of view: the interface exposed to user space may be slightly different. The `mmap2` system call supersedes the old `mmap`, and both use the same internal function which is `sys_mmap_pgoff`. Although these functions are used for many other types of

mappings, we focus on mappings between virtual addresses and physical memory addresses. These system calls serve the following purpose, respectively:

- **mmap2**: request a new mapping related to the `fd` address space (e.g., `/dev/mem` for physical memory) having `len` bytes and starting from page offset `pgoff`; returns the mapped virtual address pointing to the starting offset of the target address space;
- **mremap**: modify an existing mapping, either remapping it to a new virtual address `new_addr` (*move*) or resizing it to `new_len` bytes (*grow* or *shrink*); returns the virtual address after the modification;
- **remap\_file\_pages**: remap an existing mapping to point to a different start page offset `pgoff` of the target address space (e.g., to point to a different physical address); returns 0 if successful, an error code otherwise.
- **munmap**: delete an existing mapping of `len` bytes related to the virtual address `addr`; the `len` parameter is aligned to the next page boundary, and the resulting range of pages is removed.

The type of a requested mapping can be inferred from the `fd` (file descriptor) parameter in `mmap2`, or from the `addr` parameter in the other calls, which refer to previously mapped regions. Every mapping is managed at a page level, and the arguments are aligned to page boundaries before handling the request. The scheme in Fig. 16 summarizes a significant sequence of operations performed via the mapping interface, assuming physical and virtual address spaces based on 4 kB pages.

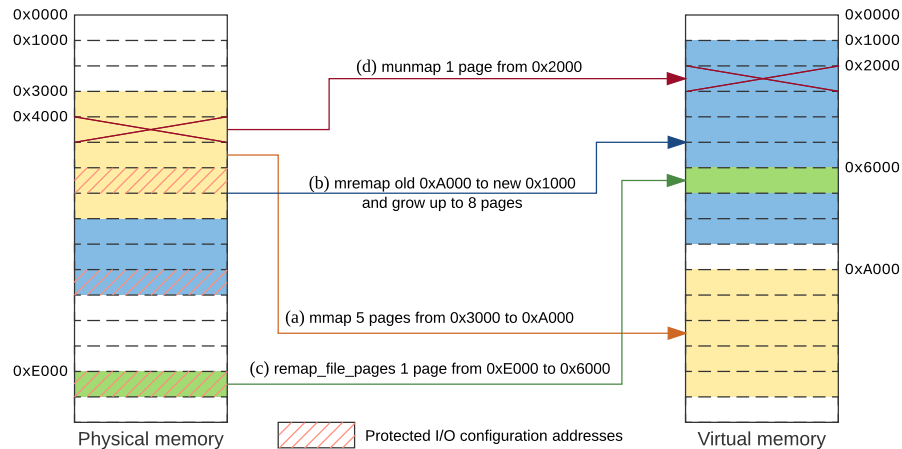


Figure 16: Accessing physical addresses through Linux mapping interface

As shown in the figure, there exist 3 different ways of accessing protected I/O physical addresses by using mapping functions. When a mapping includes

a portion (even one byte) of the I/O configuration defined by the I/O monitor, we say that an *overlap* occurs. First, an attacker may call `mmap2` (or `mmap`) to directly include the I/O configuration into the requested range as illustrated in Figure 16.a. Second, he may use `mremap` to extend a current mapping, possibly causing overlap on an I/O address which was not mapped before as illustrated in Figure 16.b. Third, he can re-map an already mapped virtual address to a different physical address, including a protected I/O address as well as illustrated in Figure 16.c. Note that, the second and the third system calls identify the existing mappings by means of the virtual addresses returned by the first call to `mmap2`. Therefore, the MAP monitor must keep track of the current existing mapping (and virtual addresses) to verify if a subsequent re-map may cause an overlapping. Finally, the `munmap` call deletes an existing mapping (or part of it) as illustrated in Figure 16.d. Even if it cannot cause an overlap, the `munmap` system call needs to be monitored as well in order to update the data structure containing the existing mappings and remain consistent with the kernel data structures.

To keep track of the current mappings related to physical memory, our monitor uses a global list of pages. Each page is a structure containing a physical address, a virtual address, and a process identifier. We assume that the number of processes normally interested in physical memory is quite low. Hence, a global list containing mapped pages of all the processes together performs well enough for our purpose. If that is not the case, the monitor could be improved, e.g., by using a hash-table based on process identifiers.

Since the mapping system calls may be called by different processes concurrently, our data structure needs to be protected by a mutual exclusion mechanism. The implementation supports classical operations such as insert, search, update and remove, each one at a page level.

The abstract part of our monitor is basically hook functions. These functions replace the original system calls, to receive all the mapping requests coming from user space. When a map request is not interesting (e.g., does not target physical memory) or it is allowed, the monitor forwards the request to the original system call. Otherwise, the monitor policy is applied, and the request may be denied before reaching the original system call.

Since in our target system the PLC runtime requests a new mapping from user space every time a PLC logic is uploaded, we configured the monitor as passive, to only report the mapping requests. Thus, it only provides additional information about an eventual attack from user space. This could be improved by providing reports based on a statistical model of the PLC runtime requests, or even better by designing a defense-aware PLC runtime. For instance, if our detection mechanism is integrated within the PLC runtime, the MAP monitor may be used in the following way:

- when a new mapping request has been detected, Ghostbuster can signal the PLC runtime;
- the PLC runtime knows if the request is due to a current logic update; if it is not, the runtime can re-throw the alert to the operator terminal.

The optimal solution, of course, would be to completely avoid using mapped I/O addresses from user space, and manage the I/O from kernel space only. However, depending on the actual implementation, this could require several changes in the PLC software. Therefore, using the signaling approach might be preferred in that case.

#### 9.4.1 Wago PLC version

On Wago PLC system, the I/O is entirely managed from kernel space already, and from our experiments we found that no user processes send mapping requests for physical memory. Thus, the MAP monitor can be configured as active, to directly deny any I/O map request from user space. Since this PLC is based on ARM architecture as well, the MAP monitor does not need any modification to be ported to this system, except for the `is_phys_mem` function that should target only the `/dev/mem` device (`/dev/gpiomem` only exists on Raspberry Pi). Therefore, for Wago PLC, the MAP monitor alone can actually prevent any attack residing in user space, because the operating system does not provide any other mechanism for users to access physical memory.

### 9.5 Module usage

Our defense can be either dynamically inserted as a Loadable Kernel Module (LKM), or can be built-in into the Linux kernel. The second approach requires a kernel re-compilation to obtain a new kernel image including our module. There are no performance differences between the two configurations, the only difference is that if an LKM is used, it can be unloaded as well. In our prototype version for Raspberry Pi, the I/O monitor needs to set a watchpoint on a PLC runtime virtual address to verify the I/O configuration against the current I/O operations. Therefore, we need to pass this virtual address as parameter, together with the process identifier of the PLC runtime, to our kernel module. This could be avoided by a better integration between our module and the PLC runtime. For instance, it could be the Ghostbuster module itself to start the PLC runtime process and provide the mapped virtual address.

When the module is running, all the information about detection events is reported among the other kernel messages as follows:

```
root@raspberrypi:~ # dmesg -T
[Tue Nov 15 10:54:42 2016] Ghostbuster: I/O monitor started
[Tue Nov 15 10:54:42 2016] Ghostbuster: DR monitor started
[Tue Nov 15 10:54:42 2016] Ghostbuster: MAP monitor started
[Tue Nov 15 10:54:42 2016] Ghostbuster: Ghostbuster started
[Tue Nov 15 10:55:50 2016] [RK] init
[Tue Nov 15 10:55:50 2016] [RK] Pin Multiplexing Hijacked!
[Tue Nov 15 10:55:50 2016] Ghostbuster: I/O change detected:
phys[0xf2200000] [old value = 0x00048924, new value = 0x00048824]
[Tue Nov 15 10:55:50 2016] Ghostbuster: Illegal change: Pin Control Attack!
[Tue Nov 15 10:55:50 2016] Ghostbuster: I/O state restored
```

In the code above, the kernel module variant of Pin Control Attack has been immediately detected by Ghostbuster. In the current implementation, the reporting mechanism is just a prototype that prints messages only useful for testing and logging.

## 10 Experimental Results

This section describes the evaluation phase of our defense solution. Our purpose is to give an estimation of its efficacy and efficiency on different scenarios. After specifying the configuration of the target system used for our experiments, we define and implement different test cases and provide the experimental results for each one.

### 10.1 Experiments definition

To validate our detection system and to estimate its overhead, we performed different experiments. Given the technical problems described in Section 9.3 for the Wago PLC version, we chose the Raspberry Pi with CODESYS runtime as target system for our tests. Note that this system does not have a real-time operating system, and the accuracy of experimental results can be improved by running them on a real PLC. The system has the same PLC logic and I/O configuration for attack analysis. The attack variants used for tests, instead, are reported in the following sections, because they may differ according to the specific test case. For our test cases, we chose  $t = 10, 5, 2$  ms as most significant monitor scan intervals.

In the first phase we evaluate the effectiveness of our solution by estimating the detection rate separately for each monitor. We discuss about its variation over different values of the monitor scan intervals, both theoretically and with the support of experimental results. In the second phase, we defined the following test cases to measure the performance overhead:

- **normal conditions:** we measure the overhead during normal PLC logic operations, without any attack or external influence;
- **pin configuration:** we estimate the overhead when a pin configuration attack is executed;
- **pin multiplexing:** same as above, but for a pin multiplexing attack (note that their detection is different, see Section 8.1.1);
- **logic upload:** we provide an estimation of the overhead during the process of uploading a new logic (with a different I/O configuration) to the PLC runtime.

To measure the defense overhead, we leveraged Hardware Performance Counters (HPCs). HPCs allow the user to measure the number of active CPU cycles

executed by a system process during a certain amount of time. The CPU cycles are strictly related to the operations performed by the CPU, since each instruction corresponds to a certain number of CPU cycles. Basically, the usage scheme of HPCs is made of the following operations:

1. reset and start a hardware counter;
2. wait for target operations to complete, or set a timeout;
3. read the value contained into the hardware counter.

In ARM processors, hardware counters are managed by a specific component called Performance Monitoring Unit (PMU). For our purpose, we leveraged PMU by using the following methodology:

1. we measure the operations performed by the whole kernel in a definite time interval  $t$ ;
2. we deploy our detection system (loading the kernel module);
3. we measure again the whole kernel operations during the same time interval  $t$ ;
4. we compare the results of the two cases.

To measure the operations performed by the entire kernel, we built a minimal kernel module that uses a PMU hardware counter from kernel space, thus counting every operation performed in kernel context. To improve the accuracy of our results, this methodology is repeated several times, separately for each test case defined above. Moreover, we chose  $t$  as multiple of the PLC logic scan cycle, in order to have an estimation of the overhead easily comparable to the PLC timing. The rest of the chapter reports and describes the results of our experiments.

## 10.2 Detection rate

This section contains the results of our detection rate analysis for each Ghostbuster monitor.

### 10.2.1 MAP monitor

Since our MAP monitor implementation does not use any heuristic or statistical approach, it does not introduce any uncertainty window. In fact, we only have the following two cases:

- **Raspberry Pi:** the MAP monitor is in passive mode, providing just logging information about mapping requests; since the PLC runtime uses mapping requests as well, a mechanism to detect malicious requests needs to be implemented (see Section 9.4);

- **Wago PLC:** the MAP monitor is in active mode, denying any possible mapping request from user space; the PLC runtime does not use direct I/O from user space;

In the second case, the monitor is able to exactly detect all the attacks from user space. In the first case, instead, a detection rate cannot be defined at all. Thus, for the rest of this analysis we focus on I/O and DR monitors: both of them execute a time-based protection mechanism.

### 10.2.2 DR monitor

Based on our threat model, the attacker may either modify I/O configuration directly or use debug registers to obtain better time accuracy before accessing I/O. Thus, we can analyse the DR monitor detection rate first, independently from the I/O monitor. Note that, since the DR monitor disables the user interface, the attacker needs to gain kernel privileges to use debug registers.

Our analysis is based on the fact that the attacker uses debug registers to get a reference time of the PLC logic I/O operations, performed at each scan cycle. For the following theoretical considerations, we assume a 10 ms PLC scan cycle, a DR monitor having a scan interval of 10 ms as well, and a real-time operating system (i.e. precise timing operations). In this configuration, the DR monitor verifies debug registers once per each PLC scan cycle. Since DR monitor and PLC logic are unrelated programs, they are not synchronised; hence, their reference times are completely independent. When the attacker modifies a debug register, he gets a debug exception on the exact moment of the next I/O operation. In the worst case (for the attacker), this will happen 10 ms after setting debug registers. When the attacker has gained this timing information, he can immediately restore the value of the used debug register back to the trusted value. The elapsed time between modification and restoring is our *detection window*. In this context, our DR monitor already raises the bar for the attacker. In fact, in the original version, the attacker could set debug registers and keep them as long as he needed (i.e. unlimited detection window) in order to intercept all the desired I/O operations. Since this is no more doable because of our monitor, we consider a more sophisticated version in which the attacker tries to minimise the detection window.

The probability that the attacker gets noticed is equal to the probability that our DR monitor falls within the detection window. If we make the assumption that both attack and DR monitor are blind and independent, i.e. both are uniformly distributed within a scan cycle, the above probability is the same as the probability that the attacker does not get noticed (i.e. the attack comes after the DR monitor check). Theoretically, with this configuration and hypothesis, we have a 50% detection rate. Fig. 18 shows these two cases. In the first case the attack is detected because the DR monitor check point occurs within the detection window. Otherwise, it is not detected. Note that the detection window is the time interval during which the attack can be detected because debug registers are holding a malicious value.

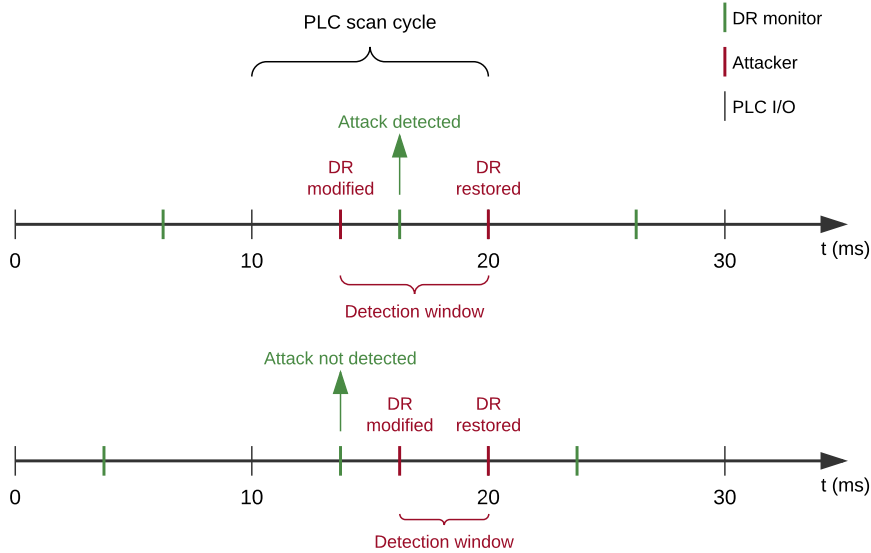


Figure 18: DR monitor detection window

Intuitively, the closer the DR monitor is to the next I/O operation, the higher is the probability to detect the attack. Thus, this result can be improved in different ways.

1. **Increase:** the number of checks per scan cycles may be increased, by reducing the monitor scan interval. This would increase the probability of detecting the attack. For example, consider a monitor interval of 2 ms, implying that the DR configuration is verified 5 times per scan cycle. In this case, if the attack detection window spans more than 2 ms before I/O, the attack would be 100% detected. Otherwise, if the attacker is lucky and the window is less than 2 ms, the detection probability is 50% as before. Therefore, considering the whole range of 10 ms, the total detection rate would increase to  $1 \times 0.8 + 0.5 \times 0.2 = 0.9 = 90\%$ , as confirmed later by our results. This approach, however, may cause too much overhead (see Section 10.3).
2. **Randomise:** the DR monitor could be improved by periodically randomising its reference time; for instance, every  $n$  PLC scan cycles, it can be programmed to sleep for a random time  $t \in [5, 15]$  instead of 10 ms. If  $n = 1$ , the monitor is completely randomised.
3. **Synchronise:** the monitor can be synchronised with the PLC logic before getting started, to verify debug registers as close to the next I/O operation as possible. The synchronisation can use debug registers as well, thus resulting in a precise timing.

Furthermore, the above approaches may be combined to achieve better results. For instance, combining 2 and 3 approaches may be useful to generate minimal time alterations on a synchronised monitor. In particular, consider a synchronised monitor that is triggered 1 ms before each I/O operation. If we generate a random value  $r_i \in [0, 1]$  at each cycle  $i$ , with  $r_0 = 1$ , we can define a variable monitor interval  $t_i = (1 - r_{i-1}) + 9 + r_i = 10 + r_i - r_{i-1}$  instead of fixed 10 ms. If the PLC I/O operation  $i$  is performed at  $p_i$  instant, the random interval ensures that the monitor is triggered in an unpredictable time instant  $t \in [p_i - 1, p_i]$  (i.e. 0.5 ms before I/O, on average). According to the system implementation, the synchronisation may go even closer to the I/O operation. Since our Raspberry Pi system does not have real-time capabilities, the experiments of such an accurate solution cannot give significant experimental results (e.g. it is even possible that the monitor checkpoint goes *after* the I/O operation). Therefore, our monitor should first be ported to a real-time system, such as the Wago PLC, to evaluate this approach.

In our previous considerations, we assumed that the attack is blind, hence it can appear at any time within a PLC scan cycle. Unfortunately, a more sophisticated attack may synchronise itself with the PLC scan cycle by manually watching the target I/O value. After obtaining a rough synchronisation, it would be able to set a debug register for only a few milliseconds before the next I/O operation, thus reducing the chances of being noticed. Although we could not properly test it on a real-time environment, the synchronised DR monitor should be able to detect the attack. The reason is that the attacker can only get an approximate reference time, while the DR monitor is more accurate because it takes advantage from debug registers. Note that the attacker may also repeat the use of debug register at each I/O operation, but this behaviour dramatically increases the chances of getting noticed. Moreover, if the attacker wants to get more accuracy, he has to poll the I/O value faster, probably causing a non-negligible performance overhead.

In any case, the results of our analysis and experiments demonstrate that the DR monitor practically raises the bar for the attacker. We tested the DR monitor with different time intervals (approach 1), obtaining the detection rates shown in Fig. 19. The detection rate increases as the monitor scan interval decreases, as expected. These experimental results confirm our theoretical considerations for approach 1, but they are slightly lower than the expected values. Probably, this is due to the fact that we cannot exactly reproduce uniformly distributed events inside a scan cycle, because the system is not real-time and the events may depend on the current CPU load. The results are obtained by repeating the following operations  $n = 1000$  times:

- start the DR monitor;
- wait for random time interval (to achieve pseudo-independence);
- execute the attack;
- wait for enough time to complete a scan cycle;

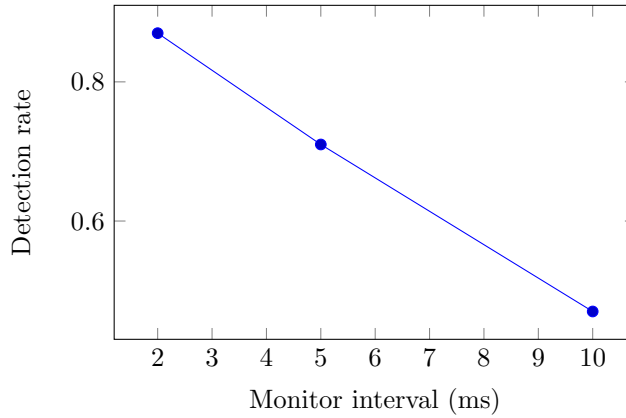


Figure 19: DR monitor detection rates

- terminate the attack and stop the monitor;
- check whether the attack has been detected or not.

The detection rate is computed as  $\frac{d}{n}$ , where  $d$  is the number of detected attacks out of  $n$ .

### 10.2.3 I/O monitor

The detection rate analysis for I/O monitor is similar to what we did for DR monitor. The main difference is that, in order to reach his goal, the attacker will likely need to tamper with many subsequent I/O operations, while it needs debug registers just once at the beginning. Thus, the probability of remaining stealth against I/O monitor is the product of all the probabilities of each scan cycle. For instance, if the probability of being unnoticed within a scan cycle is 0.70, and the attacker needs at least 20 operations, the probability of escaping the detection is  $0.70^{20} = 0.0008 = 0.08\%$ . Note that we did not assume a high detection rate for scan cycle (0.30), and we only considered 20 operations, which means just  $10 \text{ ms} \times 20 = 200 \text{ ms}$ .

For the I/O monitor analysis, we distinguish the following two cases:

- the attacker directly modifies I/O configuration without using debug registers;
- the attacker is able to circumvent DR monitor and use debug registers, and modifies I/O configuration in proximity of the PLC I/O operation.

In the first case, we do not repeat our considerations because they are similar to what discussed for DR monitor. For the second case, the attacker can leverage debug registers either for initial synchronisation only, or for each scan cycle. As discussed in Section 10.2.2, the former case allows the attacker to get an initial

starting reference time, while the latter case is practically unfeasible due to the presence of the DR monitor. Thus, we assume that the attacker is no more able to be extremely accurate, but it can only change the I/O configuration in proximity of each PLC I/O operation. In this scenario, if a PLC I/O is performed at time  $p_i$ , the best approach for the monitor is to check I/O configuration randomly within  $[p_i - \epsilon, p_i + \epsilon]$ . Note that this behaviour must be achieved without using debug registers for each scan cycle, because it would cause too much overhead. Therefore, it must be done manually. However, since in our non-real-time target system we were not able to accurately measure its detection rate, we relaxed the above constraint. Given a fixed monitor interval  $t$  and a random value  $r_i$  for each scan cycle, we check I/O configuration every  $t_r$  ms, with  $t_r \in [t - r_i, t + r_i]$ . Thus,  $t$  becomes the *average* I/O monitor interval. We tested this monitor against an attack which changes I/O configuration 0.5 ms before PLC I/O and restores it 0.5 ms after the I/O operation. We estimate the probability that the attack is detected at each scan cycle, as we did for DR monitor.

The results of our approach are shown in Fig. 20. The detection rate  $r$  related to one single scan cycle is lower with respect to the DR monitor case, because we used a more sophisticated attack. However, since the attacker has to repeat the I/O attack for more than one scan cycle to obtain significant effects on the physical process, we consider the probability of getting noticed after  $n$  manipulated scan cycles, which is  $1 - (1 - r)^n$ , where  $r$  is the detection rate related to each scan cycle. Therefore, the probability of remaining unnoticed actually decreases exponentially at each scan cycle. For instance, given the detection rate  $r = 0.2$  of the monitor with  $t = 5$ , the probability that the attacker gets noticed after just 10 scan cycles is  $1 - (1 - r)^{10} = 0.89 = 89\%$ . Note that, if the attacker is using debug registers, both DR monitor and I/O monitor detection rates must be taken into account, making the probability of being detected even higher.

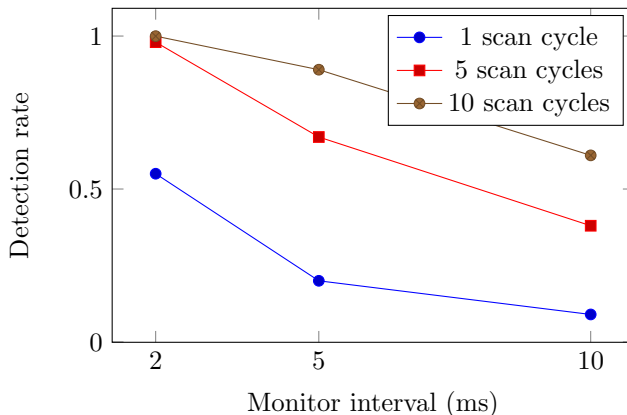


Figure 20: I/O monitor detection rates

### 10.3 Performance overhead

Finally, we evaluate our implementation by measuring the performances in 4 different test cases, as previously described in Section 10.1. Since our overhead is always lower than 1% and our target system is not accurate enough (not real-time), we could not get reliable results separately for each monitor. Therefore, we provide the overhead estimation for our whole implementation as a function of the monitor time intervals. We chose  $t \in \{2, 5, 10\}$  as representative time intervals (in ms), which are the same used for detection rate estimation. In this case, a configuration having time interval  $t$  means that both DR and I/O monitor are configured to be triggered every  $t$  ms, while the MAP monitor remains unmodified among different configurations. The following sections report the results related to the corresponding test case.

#### 10.3.1 Normal conditions

For this test case, we just observed the CPU cycles of the entire system and its PLC runtime, with and without Ghostbuster. No attacks are executed during the test, to estimate our defense overhead when the system is working under normal conditions, which on a real PLC is true for most of the time. The experimental results are reported in Fig. 21, showing CPU cycles over time, in 4 different configurations: Ghostbuster deployed with  $t = 2, 5, 10$ , and without Ghostbuster. The amount of CPU cycles is collected every 50 ms (5 PLC scan cycles) by leveraging the specific Performance Monitoring Unit (PMU) counter register.

We estimated the Ghostbuster overhead from the data shown in the plot, considering the average of CPU cycles over time. When our defense is not present, the system performs  $0.992 \times 10^7$  CPU cycles on average. After deploying Ghostbuster, the system consumes  $1.019 \times 10^7$ ,  $1.049 \times 10^7$ , and  $1.107 \times 10^7$  CPU cycles on average, causing an overhead of 2.72%, 5.82%, and 11.59% for  $t = 10, 5, 2$  ms respectively. Thus, depending on the target system, the user must choose the best trade-off between detection rate and overhead. For instance, the solution having a monitor scan interval  $t = 2$  ms has a desirable detection rate, but its overhead may be unacceptable for many embedded systems with limited resources and strict timing requirements.

#### 10.3.2 Pin configuration detection

For this test case (and the following ones), we focused on just one Ghostbuster configuration, having  $t = 10$  ms. In fact, the overhead in case of an attack detection is independent from the value of  $t$ . The test is conducted in the same way as the previous case, except that in the middle of measurements we execute a pin configuration attack. Note that our solution, in order to distinguish a malicious manipulation from a PLC runtime configuration update, sets a watchpoint and looks into the actual PLC logic. The results are shown in Fig. 22. The plot shows just an instance of our test case. To estimate the detection overhead, we repeated the same test several times. The overhead has been computed as the

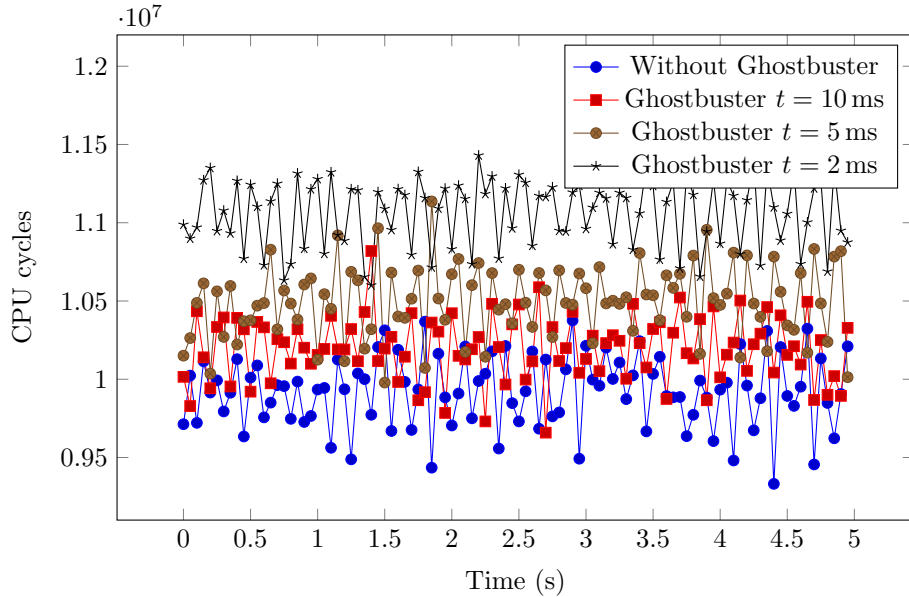


Figure 21: Ghostbuster overhead during normal condition

extra *detection area* below the red plot line (with Ghostbuster) with respect to the area of the blue graph (without Ghostbuster). Note that the overhead of the blue plot is entirely related to the attack itself, and it has been excluded from our defense overhead. Therefore, the estimated overhead of the detection phase for pin configuration attack is 28.75% on average.

### 10.3.3 Pin multiplexing detection

We reproduced the same test for pin multiplexing attack, whose detection is easier because it does not require debug registers. The results are shown in Fig. 23. As for the above test case, the graph reports the results of a particular test instance. To estimate the overhead with more accuracy, we repeated the test several times to obtain an average value. Using the same concept of the extra detection area below the plot line, we found that the overhead is around 5.76% on average. As expected, it is much lower than the overhead related to pin configuration detection. Note that, in both cases (pin configuration and pin multiplexing) our monitor is configured as *active*. Thus, besides the detection, both overheads include the monitor reaction. In fact, the attack is immediately reverted back after detection, erasing any malicious effect on the PLC output.

### 10.3.4 PLC configuration update

Our solution is able to automatically recognise malicious I/O configuration, to simplify the job for the industrial operator who wants to upload a new valid

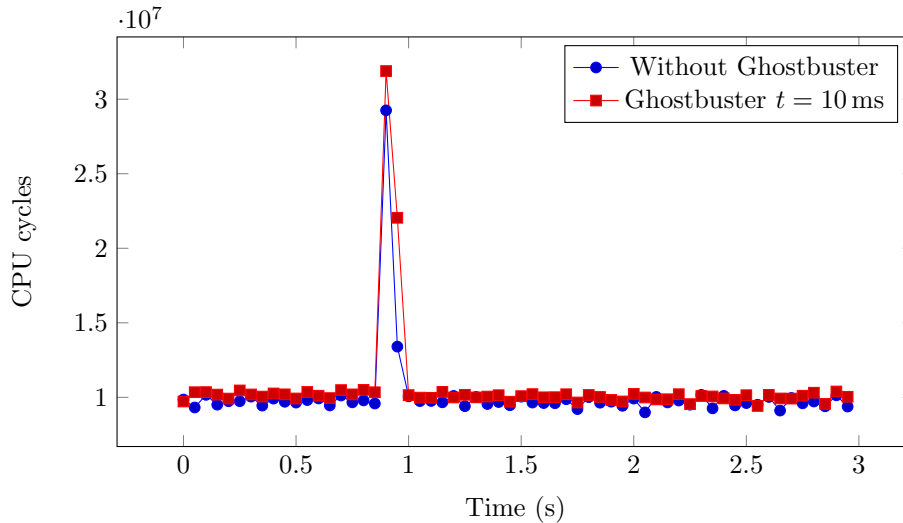


Figure 22: Ghostbuster overhead during normal condition

configuration to the PLC without getting a detection alert. We briefly analyse the behaviour of this test case and report our experimental results. In particular, we re-use the first test case (normal condition, without attack), repeated with and without our defense. Additionally, while the test is running, we manually upload a new I/O configuration from the PLC CODESYS environment. Since we assumed that no pin multiplexing is allowed during run-time, we modify pin configuration just by changing a pin from output to input. To have smaller delays and simplify the tests, we slightly modified the PLC logic, to toggle the output every 1s instead of 2s, and we changed the Ghostbuster wait time according to it. This wait time is the time between a detection of an I/O configuration change, and the decision of accepting or rejecting it. On Raspberry Pi system, since the output is written only when it is modified, Ghostbuster needs to wait at least 1s as well. As a safety measure, we let it wait for 1.2s. This behaviour is not needed on real PLCs, whose default behaviour is to write at every scan cycle (i.e. waiting for few milliseconds would be enough).

The results of this test case are shown in Fig. 24. Since the system is highly noisy during this test, because of the active operations on the PLC software, we are not able to estimate the overhead. However, the experimental result is still interesting to show the operations performed by Ghostbuster. Since the upload of a new configuration is done manually, we aligned the graphs on the I/O configuration update instant. The update is performed by the PLC runtime at  $t = 0.35$  s. At this time, Ghostbuster notices the change and sets the watchpoint to check PLC logic as soon as possible. The mechanism is asynchronous and the PLC logic is started independently, we do not cause any direct delay on the PLC software. From now on, Ghostbuster waits for the next interesting I/O operation related to the re-configured pin. At  $t = 1.35$  s, when the watchpoint

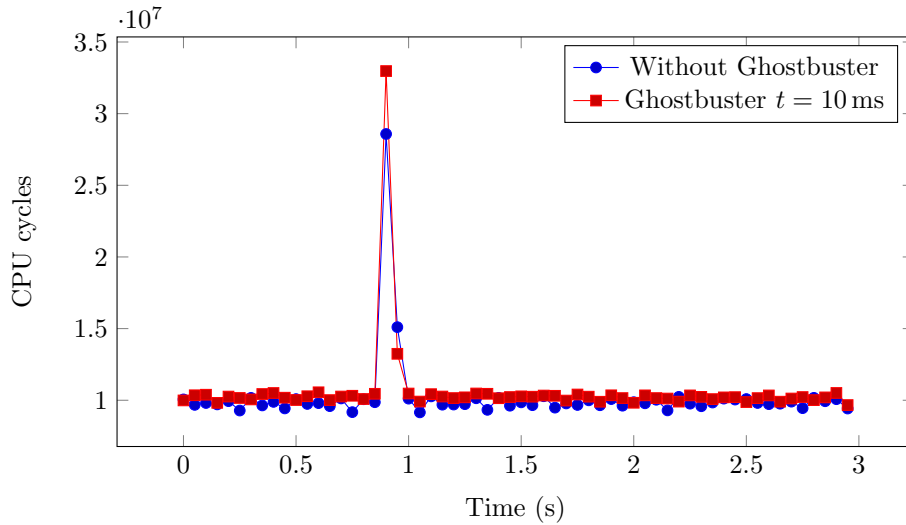


Figure 23: Ghostbuster overhead during normal condition

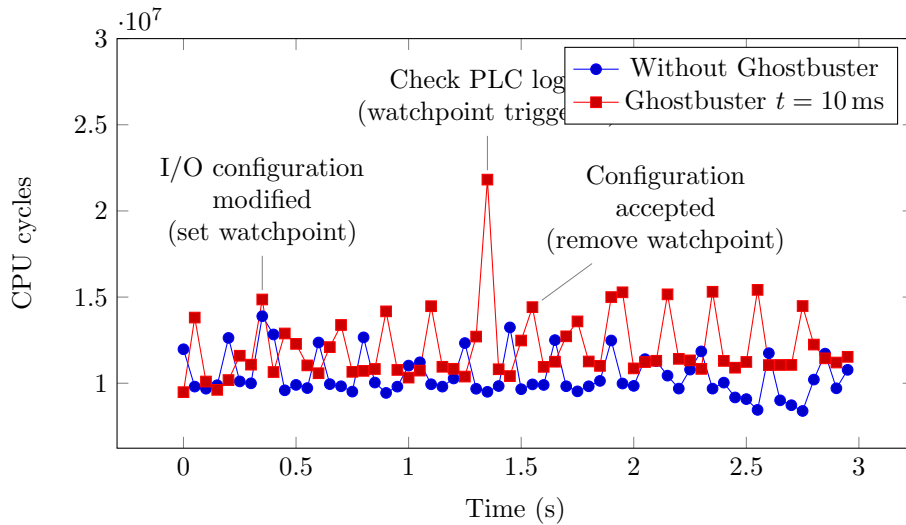


Figure 24: Ghostbuster overhead during PLC configuration update

is triggered, Ghostbuster checks if the actual logic operation is conforming with the new configuration detected at  $t = 0.35$  s. In this case, the write watchpoint does not refer to our new input pin. Thus, everything is correct and no alert is thrown. After the timeout 1.2s, the trusted configuration is updated inside our module as well, at  $t = 1.55$  s. The main overhead in this test case is caused by the watchpoint, but it cannot be accurately estimated because of the excessive

noise. However, since Ghostbuster performs the same operations as for pin configuration attack test case, the overhead should be similar to that test case.

## 11 Security considerations

Our detection system has been designed having in mind the threat model in 6.1 as well as the attack implementations presented in Sections 7.3 and 7.4. The design is meant to be as much general as possible, since it aims to cover different attack vectors and to be applicable on several target systems that may be heterogeneous in their characteristics. The definition of the monitor strategies and policies is a key aspect for having a significant solution that raises the bar for the attacker. For instance, the MAP monitor may have two completely different usages on the target systems we used for experiments. On Raspberry Pi with CODESYS runtime, the map monitor must be able to distinguish between PLC runtime mapping requests and malicious requests. The optimal solution would require to design the PLC runtime to be aware of our defense and cooperate by using some sort of authentication. On Wago PLC with e!COCKPIT runtime, instead, since the PLC runtime does not request any user space I/O mapping, the MAP monitor could be used to deny any request coming from user space, factually preventing any possible attack from user space and forcing the attacker to gain kernel level access.

The integration between PLC runtime and Ghostbuster could also serve another purpose. Since in our current implementation we need the PLC runtime mapped address to detect a conflict between the actual I/O configuration and the loaded PLC logic, we pass this information as argument for our module. Of course this is not a final solution, because our detection system is supposed to be started much earlier than the PLC runtime, and should not be related to the PLC runtime process life time. Therefore, we propose the following approach, which makes use of the Public Key Infrastructure (PKI), to design and integrate Ghostbuster with the PLC runtime:

1. compile Ghostbuster with a hard-coded public key certificate owned by the PLC runtime vendor and a digital signature of the PLC runtime code, computed with the corresponding private key;
2. configure Ghostbuster to be loaded at boot time (either as loadable module or built-in into the kernel);
3. when the PLC runtime is started (or stopped), Ghostbuster can recognize the process by computing the hash on the actual PLC runtime code, and comparing the result with the hard-coded signature.
4. once the PLC runtime is recognized, Ghostbuster can get its identifier and virtual address dynamically.

This approach should be evaluated, at least with respect to performance overhead. Moreover, the PLC runtime vendor must design its own process to

support secure PLC runtime software updates. In fact, each update would require a new digital signature to be loaded into Ghostbuster as well (e.g., it could be done dynamically, by including the upload procedure into the already signed PLC runtime code). Furthermore, since we assumed that a modification of the PLC logic is easily detectable (see 8.1.1), the PLC runtime must implement this feature. For instance, the PLC programming software may use an authenticated channel to upload a new PLC logic, together with the corresponding digital signature. Once the signature is verified by the PLC during the upload phase, the hash of the PLC logic can be stored in kernel space and compared to the actual hash, computed at random times.

In our work we assumed that the system is protected from classical function and data hooking techniques, as it will likely be in the next years. In this scenario, malicious users will be forced to avoid these techniques, and to leverage lower level mechanisms such as I/O configuration. At the time of writing, the above protection mechanisms are not complete, and they still have limitations. In particular, an attacker who is able to gain kernel privilege level, can still duplicate some kernel code or directly write to kernel text without being noticed. Note that, as shown by recent research [6] and confirmed by our experiments, the PLC runtime software is typically executed having admin privilege level (i.e. root). Thus, is not unrealistic to gain kernel level access, by simply exploiting a vulnerability in the PLC runtime. One of the main problems to overcome for kernel manipulation is leaking of kernel information, in particular kernel addresses. Once the attacker knows the address of its specific target inside the kernel, he can easily reach its goal. For instance, if the attacker knows the starting address of our monitor instructions that are responsible for verifying a configuration, then he could simply replace them with no-operations. For this reason, our defense should be protected against these attacks as well, in order to prevent defense-aware attackers from circumventing the protection mechanisms. This is not a problem of our module, but is quite a general problem affecting the whole kernel. The approach aimed to protect the kernel itself and reduce its attack surface is known in literature as *kernel hardening* [33]. Kernel hardening techniques, such as Kernel Address Space Layout Randomisation (KASLR), read-only memory, execute-only memory, etc., would be useful to protect our defense from kernel level attacks. Of course, the ultimate solution to this problem would be using a Trusted Execution Environment. Unfortunately, this is not always applicable, as in our case, due to its unacceptable overhead.

As previously discussed, Ghostbuster could be deployed as loadable kernel module, or it can be built-in into the kernel itself. In general, integrating the module directly into the kernel would be a better choice from a security point of view, because a loadable kernel module may be unloaded at any time. Furthermore, a module is loaded into dynamic memory, i.e. it is not part of the static kernel code; thus, it cannot be protected by static mechanisms such as Symbiotes. However, using a loadable version allows module users to upgrade it when necessary, without requiring a machine reboot. If the loadable version is chosen, it should be deployed with care. First, the privilege level required to unload the module (typically root) should be properly protected (e.g., no default

password). Second, the module itself can be protected against unloading, by leveraging the mechanisms available into the kernel (e.g., a module which is in use by another module cannot be unloaded). Third, information leaking to user space should be avoided. In particular, the virtual dynamic address where the module is loaded should not be available. Note that this is typically not the case (e.g., any (even non-root) user can get such an address from `/proc/modules` or similar).

Another useful mechanism to prevent attackers from gaining kernel level access is to use signed kernel modules [26]. If an attacker is able to insert its own loadable kernel module, he has access to the entire kernel space, and may be able to circumvent our defense as well. To avoid this, the kernel can be compiled including hard-coded public keys of trusted entities, who will be able to sign their own modules. A module, which is an ELF (Executable and Linkable Format) file, may be signed by simply including an extra section containing a digital signature computed on all the `text` and `data` sections. With this mechanism, the authenticity and the integrity of a module can be verified before being loaded. If a malicious users attempts to load its own unsigned module, the system can block it and may raise an alert signal, which can be forwarded to the industrial operator. Alternatively, if the system does not need loadable modules at all, the kernel module functionality may be completely disabled [41]. Since it is available during run-time, this mechanism can be applied for both Ghostbuster variants. For built-in version, modules can be disabled at boot time; otherwise, they can be disabled as soon as Ghostbuster module is loaded. Note that this usage model is not based on an unreasonable assumption, because PLC systems are typically made of a very small and stable environment, and might be very likely that they do not need dynamic modules support. In fact, as confirmed by our experiments on Wago PLC, the system does not make use of any loadable kernel module.

## 12 Related Work

Various research works have addressed attacks against embedded systems. For example, a significant stream of work has explored the manipulation of embedded systems' firmware [5, 12, 48, 58]. Another relevant stream of work, instead, has explored memory corruption vulnerabilities in embedded systems [6, 14, 65]. However, to the best of our knowledge, we could not find other approaches discussing the security implications of the pin control system in embedded devices.

Part of our work bears some similarities with System Management Mode (SMM) rootkits [20, 53, 56] for X86 architectures. These rootkits tap the system I/O, similarly to what we did in our Pin Configuration attack. However, while the goals is similar, the way to reach the goal is different. For example, the modification of system I/O in SMM causes interrupts which need to be suppressed by SMM rootkits, typically by attacking kernel interrupt handlers. In our case, this operation is not needed due to the lack of interrupts for pin multiplexing and configuration.

Among the works focusing on protecting embedded systems, a stream of research focuses on firmware verification [2, 19, 67]. Another stream of research focuses on detecting kernel level attacks by monitoring syscall/function hooking techniques and kernel data structure manipulation [13, 35, 37, 52, 62].

None of the existing detection mechanisms monitor I/O memory ranges and specifically I/O configuration registers.

## 13 Conclusions

In this chapter, we first looked into the current state of host-based detection techniques for embedded devices, with a particular focus on programmable logic controllers. We found that current practical host-based intrusion detection techniques for embedded devices suffer from three major shortcomings. First, they completely ignore the control of dynamic memory when verifying memory contents. Second, they do not apply effective practical control-flow measures due to performance limitations. Finally, they mostly rely on static references to protect embedded devices. In the second part, we have proposed a new type of attack that leverages these weaknesses, and we have shown that it can be used by adversaries to manipulate the physical process in a way that the PLC runtime and the SCADA applications are unaware of the manipulation. This makes the attack interesting and relevant since current detection techniques are not effective against this new type of attack or any type of attack that exploits the weaknesses discussed in the Section 4. In the third part of this chapter we discussed GhostBuster: an approach to detect Pin Control Attack. GhostBuster can detect anomalous changes to the configuration of I/O and raise and alert upon detection. However GhostBuster alone is not sufficient to protect the system, it needs to run alongside other host-based protection mechanisms.

## References

- [1] F. Abad, J. van der Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, “On-chip control flow integrity check for real time embedded systems,” in *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, Aug 2013, pp. 26–31.
- [2] F. Adelstein, M. Stillerman, and D. Kozen, “Malicious code detection for open firmware,” in *18th Annual Computer Security Applications Conference, 2002. Proceedings*, 2002, pp. 403–412.
- [3] F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, “A security framework for the analysis and design of software attestation,” in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS ’13, S. Merz and J. Pang,

- Eds. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: <http://doi.acm.org/10.1145/2508859.2516650>
- [4] A. Baliga, V. Ganapathy, and L. Iftode, “Detecting kernel-level rootkits using data structure invariants,” *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 5, pp. 670–684, Sept 2011.
  - [5] Z. Basnigh, J. Butts, J. L. Jr., and T. Dube, “Firmware modification attacks on programmable logic controllers,” *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76 – 84, 2013.
  - [6] D. Beresford, “Exploiting siemens simatic s7 plcs,” *Black Hat USA*, vol. 16, no. 2, pp. 723–733, 2011.
  - [7] D. Beresford and A. Abbasi, “Project IRUS: multifaceted approach to attacking and defending ICS,” in *SCADA Security Scientific Symposium(S4)*, 2013. [Online]. Available: <http://vimeopro.com/s42012/s4x13/video/58983658>
  - [8] B. Binary, “Seagate nas remote code execution vulnerability,” 2015. [Online]. Available: <https://beyondbinary.io/articles/seagate-nas-rce/>
  - [9] Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, “ROPecker: A generic and practical approach for defending against ROP attacks,” in *Symposium on Network and Distributed System Security (NDSS)*, 2014.
  - [10] W. K. G. . Co, “Linux - automation for the future.” [Online]. Available: <http://global.wago.com/en/products/new-items/overview/basic-page-2600.jsp>
  - [11] A. Cui, “Red ballon security.” [Online]. Available: <http://www.redballoonsecurity.com>
  - [12] A. Cui, M. Costello, and S. J. Stolfo, “When firmware modifications attack: A case study of embedded exploitation,” in *Symposium on Network and Distributed System Security (NDSS)*, 2013.
  - [13] A. Cui and S. J. Stolfo, “Defending embedded systems with software symbiotes,” in *Recent Advances in Intrusion Detectio: 14th International Symposium*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Springer, 2011, pp. 358–377.
  - [14] L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, “Return-oriented programming without returns on arm,” Technical Report HGI-TR-2010-002, Ruhr-University Bochum, Tech. Rep., 2010.
  - [15] L. Davi, P. Koeberl, and A.-R. Sadeghi, “Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation,” in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC ’14. New York,

- NY, USA: ACM, 2014, pp. 133:1–133:6. [Online]. Available: <http://doi.acm.org/10.1145/2593069.2596656>
- [16] L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, “Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection,” in *USENIX Security Symposium*, 2014.
- [17] DigitalBond, “3S CoDeSys, Project Basecamp,” 2012. [Online]. Available: <http://www.digitalbond.com/tools/basecamp/3s-codesys/>
- [18] —, “WAGO IPC 758/870, Project Basecamp,” 2015. [Online]. Available: <http://www.digitalbond.com/tools/basecamp/wago-ipc-758870/>
- [19] L. Dufлот, Y.-A. Perez, and B. Morin, *What If You Can not Trust Your Network Card?* Springer Berlin Heidelberg, 2011, pp. 378–397.
- [20] S. Embleton, S. Sparks, and C. C. Zou, “Smm rootkit: a new breed of os independent malware,” *Security and Communication Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.
- [21] N. Falliere, L. O. Murchu, and E. Chien, “W32. stuxnet dossier,” *White paper, Symantec Corp., Security Response*, vol. 5, 2011.
- [22] R. P. Foundation, “Bcm2835 documentation,” 2012. [Online]. Available: <https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md>
- [23] A. Francillon, D. Perito, and C. Castelluccia, “Defending embedded systems against control flow attacks,” in *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, ser. SecuCode ’09. New York, NY, USA: ACM, 2009, pp. 19–26.
- [24] I. Fratrić, “Ropguard: Runtime prevention of return-oriented programming attacks,” <https://ropguard.googlecode.com/svn-history/r2/trunk/doc/ropguard.pdf>, 2012.
- [25] P. Ghosh, P. S. Hira, and S. Garg, “A method to make soc verification independent of pin multiplexing change,” in *Computer Communication and Informatics (ICCCI), 2013 International Conference on*. IEEE, 2013, pp. 1–6.
- [26] G.K.Hartman, “Signed Kernel Modules,” <http://www.linuxjournal.com/article/7130>, 2004.
- [27] R. P. GmbH, “Avm fritz!box: Remote code execution via buffer overflow,” 2016. [Online]. Available: <https://www.redteam-pentesting.de/en/advisories/rt-sa-2015-001/-avm-fritz-box-remote-code-execution-via-buffer-overflow>

- [28] C. A. Gonzalez and A. Hinton, “Detecting malicious software execution in programmable logic controllers using power fingerprinting,” in *Critical Infrastructure Protection VIII*. Springer, 2014, pp. 15–27.
- [29] ICS-CERT, “Schneider electric modicon quantum vulnerabilities (update b),” 2014. [Online]. Available: <https://ics-cert.us-cert.gov/alerts/ICS-ALERT-12-020-03B>
- [30] —, “Rockwell automation micrologix 1100 plc overflow vulnerability,” 2016. [Online]. Available: <https://ics-cert.us-cert.gov/advisories/ICSA-16-026-02>
- [31] V. M. Ijure, S. A. Laughter, and R. D. Williams, “Security issues in SCADA networks,” *Computers & Security*, vol. 25, no. 7, pp. 498–506, 2006.
- [32] T. Instruments, “Am3517 sitara processor technical documents,” 2012. [Online]. Available: <http://www.ti.com/product/AM3517/technicaldocuments>
- [33] J. Corbet, “The status of kernel hardening,” <https://lwn.net/Articles/705262/>, 2016.
- [34] Kernel.org, “Pin control subsystem in linux.” [Online]. Available: <https://www.kernel.org/doc/Documentation/pinctrl.txt>
- [35] Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima, “Monitoring integrity using limited local memory,” *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 7, pp. 1230–1242, July 2013.
- [36] P. Koopman, “Embedded system security,” *Computer*, vol. 37, no. 7, pp. 95–97, 2004.
- [37] V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, “Code-pointer integrity,” in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.
- [38] M. LeMay and C. Gunter, “Cumulative attestation kernels for embedded systems,” *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, June 2012.
- [39] Z. Liang, H. Yin, and D. Song, “HookFinder: Identifying and understanding malware hooking behaviors,” in *Symposium on Network and Distributed System Security (NDSS)*, 2008.
- [40] LWN.net, “Hw-breakpoint: shared debugging registers,” 2009. [Online]. Available: <https://lwn.net/Articles/353050/>

- [41] M.Boelen, “Increase kernel integrity with disabled Linux kernel modules loading,” <https://linux-audit.com/increase-kernel-integrity-with-disabled-linux-kernel-modules-loading/>, 2015.
- [42] S. McLaughlin and P. McDaniel, “SABOT: Specification-based Payload Generation for Programmable Logic Controllers,” in *ACM Conference on Computer and Communications Security (CCS12)*, 2012.
- [43] —, “SABOT: Specification-based payload generation for programmable logic controllers,” in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS ’12. New York, NY, USA: ACM, 2012, pp. 439–449.
- [44] S. E. McLaughlin, “On dynamic malware payloads aimed at programmable logic controllers,” in *HotSec*, 2011. [Online]. Available: [https://www.usenix.org/legacy/event/hotsec11/tech/final\\_files/McLaughlin.pdf](https://www.usenix.org/legacy/event/hotsec11/tech/final_files/McLaughlin.pdf)
- [45] Microsoft Corporation, “Enhanced mitigation experience toolkit,” 2014. [Online]. Available: <https://www.microsoft.com/emet>
- [46] O. S. V. D. (OSVDB), “D-link dir-605l wireless n300 cloud router captcha data http request parsing remote buffer overflow,” 2012. [Online]. Available: <http://www.osvdb.org/86824>
- [47] V. Pappas, “kBouncer: Efficient and transparent ROP mitigation,” 2012. [Online]. Available: <http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf>
- [48] D. Peck and D. Peterson, “Leveraging ethernet card vulnerabilities in field devices,” in *SCADA Security Scientific Symposium (S4)*, 2009.
- [49] pt, “Oops, I hacked my PBX. Why auditing proprietary protocols matters,” *28th Chaos Communication Congress*, 2011. [Online]. Available: [https://events.ccc.de/congress/2011/Fahrplan/attachments/2023\\_oops\\_i\\_hacked\\_my\\_pbx.pdf](https://events.ccc.de/congress/2011/Fahrplan/attachments/2023_oops_i_hacked_my_pbx.pdf)
- [50] Rapid7, “D-link hnap request remote buffer overflow,” 2014. [Online]. Available: [http://www.rapid7.com/db/modules/exploit/linux/http/dlink\\_hnap\\_bof](http://www.rapid7.com/db/modules/exploit/linux/http/dlink_hnap_bof)
- [51] —, “Linksys wrt120n tmunblock stack buffer overflow,” 2014. [Online]. Available: [http://www.rapid7.com/db/modules/auxiliary/admin/http/linksys\\_tmunblock\\_admin\\_reset\\_bof](http://www.rapid7.com/db/modules/auxiliary/admin/http/linksys_tmunblock_admin_reset_bof)
- [52] J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, “Intrusion detection for resource-constrained embedded control systems in the power grid,” *International Journal of Critical Infrastructure Protection*, vol. 5, no. 2, pp. 74–83, 2012.

- [53] J. Schiffman and D. Kaplan, “The smm rootkit revisited: Fun with usb,” in *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, Sept 2014, pp. 279–286.
- [54] F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, “Evaluating the effectiveness of current anti-ROP defenses,” in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer, 2014, pp. 88–108.
- [55] A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, “SWATT: SoftWare-based attestation for embedded devices,” in *2004 IEEE Symposium on Security and Privacy. Proceedings*, May 2004, pp. 272–282.
- [56] S. Sparks, S. Embleton, and C. C. Zou, “A chipset level network backdoor: bypassing host-based firewall & ids,” in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 2009, pp. 125–134.
- [57] L. Szekeres, M. Payer, T. Wei, and D. Song, “Sok: Eternal war in memory,” in *IEEE Symposium on Security and Privacy*, 2013.
- [58] P. Traynor, K. Butler, W. Enck, P. McDaniel, and K. Borders, “Malnets: Large-scale malicious networks via compromised wireless access points,” *Security and Communication Networks*, vol. 3, no. 2-3, pp. 102–113, 2010.
- [59] Tripwire, “Readynas flaw allows root access from unauthenticated http request,” 2013. [Online]. Available: <http://www.tripwire.com/state-of-security/vulnerability-management/readynas-flaw-allows-root-access-unauthenticated-http-request/>
- [60] S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, “Dynamic hooks: hiding control flow changes within non-control data,” in *USENIX Security Symposium*, 2014.
- [61] L. Walleij, “Pin control subsystem overview,” in *Embedded Linux Conference*, 2012.
- [62] Z. Wang, X. Jiang, W. Cui, and P. Ning, “Countering kernel rootkits with lightweight hook protection,” in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS ’09. New York, NY, USA: ACM, 2009, pp. 545–554. [Online]. Available: <http://doi.acm.org/10.1145/1653662.1653728>
- [63] W. Deacon, “PATCH 2/4] ARM: hw-breakpoint: add ARM backend for the hw-breakpoint framework,” <http://lists.infradead.org/pipermail/linux-arm-kernel/2010-July/019884.html>, 2010.
- [64] S. Wegner, “Security-analysis of a telephone-firmware with focus on backdoors,” Bachelor’s thesis, Ruhr-Universität Bochum, 2008. [Online]. Available: <https://git.fabrik17.de/mrgitlab/embedded-multimedia/>

raw/437afd92da4b438f95fa3efad28564a9d0baffbd/Dokumentation/  
\_thesis\_template.pdf

- [65] R. Wightman, “Project basecamp at s4,” *SCADA Security Scientific Symposium*, 2012. [Online]. Available: <https://github.com/digitalbond/Basecamp>
- [66] K. R. Wrightman, “Vulnerability Inheritance in PLCs,” 2015.
- [67] F. Zhang, H. Wang, K. Leach, and A. Stavrou, “A framework to secure peripherals at runtime,” in *Computer Security-ESORICS 2014*, M. Kutylowski and J. Vaidya, Eds. Springer, 2014, pp. 219–238.