POLITECNICO DI TORINO

Corso di Laurea Magistrale in Ingegneria Informatica

Analysis and detection of low-level I/O attack on Programmable Logic Controllers

Supervisor
prof. Antonio Lioy

Candidate
Andrea Genuise

Academic Year 2016-2017

† A nonno Carmelo

Summary

Recently embedded systems have become more and more integrated with several aspects of our lives, and their security concerns have risen as well. In particular, Programmable Logic Controllers (PLCs), which are embedded systems deployed within the context of an Industrial Control System (ICS), needs to be reliable and secure because they can have direct effects on critical processes. From an engineering perspective, they are connected to sensors and actuators and use Input/Output (I/O) interfaces to interact with the physical world. As demonstrated by a novel kind of attack, called Pin Control Attack or I/O attack, a malicious user can tamper with the integrity or the availability of legitimate I/O operations, possibly causing physical damage to people and environment. The purpose of this thesis can be divided into two main steps. First, we want to analyse this new threat and discuss its applicability on real PLCs. Second, we design and implement a detection system able to raise the bar for the attacker. The implementation, provided as a Linux kernel module, is evaluated on an ARM-based architecture, showing its effectiveness and impact on PLCs, which typically have very limited resources and strict timing requirements.

Acknowledgements

TODO Acknowledgements.

Contents

Chapter 1

Introduction

Embedded systems are widely used today in various applications, from cars, cell phones, home automation, to critical infrastructures like power plants and power grids, water, gas or electricity distribution systems and production systems for food and other products.

Since they were almost isolated from the network, embedded systems have not been designed and built with security concepts in mind. However, many recent cyber-attacks demonstrated that such an assumption is no more valid, and the security of embedded systems became an open question to deal with.

Unfortunately, this could be a more challenging problem with respect to security for desktop and enterprise computing, for the following reasons:

- the limited capabilities and the strict timing requirements these systems have;
- the physical side effects a security breach may lead to, e.g. property damage, personal injury, death and even environmental or nuclear disaster.

In the next sections we first describe a particular type of embedded systems, then present the problem on which the rest of the paper is focused, and define the goal of this thesis.

1.1 Programmable Logic Controllers

Within the context of an Industrial Control System (ICS), embedded systems are better known as Programmable Logic Controllers (PLCs). PLCs are special-purpose embedded systems, usually deployed into harsh environments to control critical processes, e.g. automotive systems, assembly lines, robotic devices or any industrial machine that requires high control precision and reliability.

A simplified version of the environment in which a PLC may operate is outlined in Fig. 1.1.

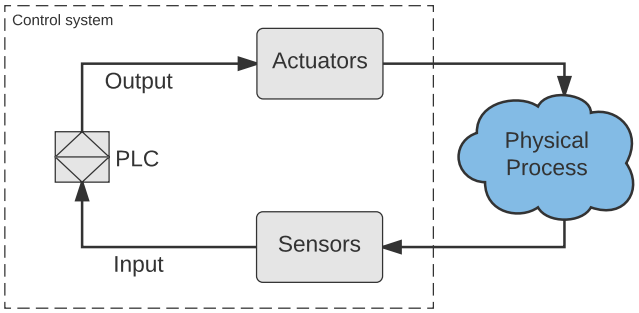


Figure 1.1: PLC environment

The PLC interacts with the physical world through external components called *sensors* and *actuators*. The sensors are responsible for reporting measurements about a set of properties of the physical process, while the actuators, as the name suggests, act on the process modifying its current state.

The main task of a PLC is contained into the control program (the *PLC logic*) which is programmed and loaded by an industrial operator. The logic is executed repeatedly as long as the controlled system is running. Each single execution is known as *PLC scan cycle*. For each scan cycle, the PLC reads the current state of the physical process measured by sensors. Internally, the logic takes these values as input and performs some calculations based on the loaded control algorithm. The output of the logic is then sent to the actuators, thus modifying the state of the controlled process.

The PLC main scan cycle is shown in Fig. 1.2.

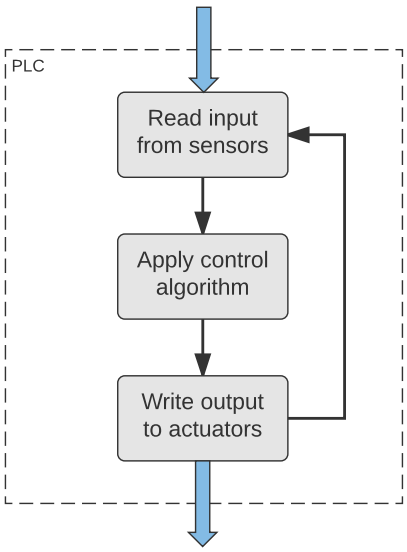


Figure 1.2: PLC logic main scan cycle

Based on the nature of the controlled process, the logic must satisfy different safety and timing properties. Furthermore, the majority of the PLCs are provided with a real-time operating system, in order to guarantee that the strict timing requirements are always respected. A violation of these requirements may lead to an unsafe behaviour of the physical process, and the consequences are heavily related to the nature of the process under control.

1.2 Problem statement

From a computer engineering perspective, PLCs control the outside world via their Input/Output (I/O) interfaces (or I/O *pins*). Therefore, it is crucial that these interfaces are both reliable and secure. They are accessible through specific memory regions inside the system, called I/O memory. By accessing I/O memory, I/O interfaces can be configured in several different ways, according to the desired behaviour. On almost every system, it is possible to change this configuration while the system is running. This feature is managed by a subsystem called *pin controller*.

The problem here relies on the fact that there is neither any memory protection mechanism nor any hardware interrupt designed for restricting the access to I/O memory addresses. They can be easily written by a malicious application even while the system is active, over-writing the hardware configuration. Therefore, a malicious user, can manipulate I/O configuration at his will. As a consequence, the behaviour of the I/O interface, hence the interaction with the physical

process, may be altered in a dramatic way. In [1], Abbasi et al. showed how this feature is exploitable by attackers, who can tamper with the integrity or the availability of legitimate I/O operations, factually changing how a PLC interacts with sensors and actuators. Based on these observations, they introduced a novel attack technique against PLCs, which they call Pin Control Attack (also I/O attack [2]). In the rest of the paper, we use these two names interchangeably. The salient features of this new class of attacks are:

1. It is intrinsically stealth. The alteration of I/O configuration does not generate any interrupt signal, preventing the Operating System (OS) to react to it.
2. It is entirely different in execution from traditional techniques (e.g. manipulation of kernel structures or system call hooking) which are typically monitored by off-the-shelf protection systems.
3. It is viable. It is possible to build concrete attack using it.

To better understand a possible attack scenario, Fig. 1.3 shows a simplified control system.

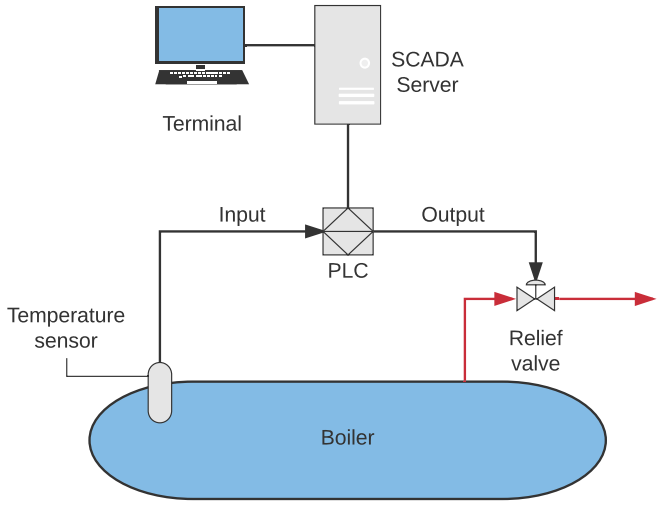


Figure 1.3: Simplified control system: a possible target of Pin Control Attack.

The system consists of a boiler equipped with a relief valve driven by the PLC according to the value provided by a temperature sensor. The PLC is connected to a SCADA server (Supervisory Control And Data Acquisition) to keep trace of its operation. The server is accessible from a terminal, through which an operator can load the logic into the PLC and monitor its current state.

Suppose that the PLC logic is programmed to read the values from the temperature sensor, and to open the relief valve if the temperature goes over $80°C$. An attacker could tamper the temperature value read by the PLC, re-configuring the I/O interface related to the sensor as output and writing its own desired value (e.g. always 50 °C) regardless of the actual temperature. Even simpler, one could reconfigure the I/O pin connected to the valve (e.g. setting it as input) factually disabling any eventual command to the actuator.

Actually there is no detection mechanism able to react to these configuration changes, neither in the PLC firmware nor in the control system. Moreover, the operator of the control system will not be able to see the real temperature nor the actual valve state from the terminal, so it may likely not notice the intrusion at all. Such a condition may lead to an uncontrolled overheating, and if it is not detected in time it could even make the boiler to explode.

1.3 Aim of the thesis

The goal of this thesis is to analyse the threat, design and implement a detection system for I/O Attack. It is a challenging task for two reasons:

1. The system lacks hardware interrupts, so it is not possible to directly react to any configuration change. More complex detection mechanisms are needed in order to achieve the highest possible detection rate.

2. The resources available within an embedded system like a PLC are extremely limited. Therefore, our solution must be extremely agile and light, since the smallest delay in the PLC I/O operation may have unintended consequences for the controlled process.

The work has been divided into three main steps. First, we discuss the idea behind the attack, analyse its threat model and its different implementations on an Embedded Linux ARM-based architecture. Second, based on the assumptions contained in the threat model and on the findings of the analysis, we describe the design and the implementation of our defense. Finally, we validate the proposed architecture with respect to the following parameters: detection rate and performance overhead.

The target of our implementation is a real-time version of the Linux kernel running on an ARM-based System on Chip, as this is one of the solutions actually used for mid-range embedded systems as PLCs. In the context of the Linux kernel, the pin controller is known as Pin Control Subsystem [3].

A brief organisation of the rest of the thesis follows in the next section.

1.4 Organisation

In Chapter 2 we list the known attacks to embedded systems existing in the literature, and then discuss the off-the-shelf protection mechanisms. To the best of our knowledge at the time of writing, as I/O Attack is a new kind of attack, none of the existing protections is able to prevent nor detect it. Chapter 3 contains, after a brief description of the embedded systems architecture, a detailed analysis of the attack design and implementation. In Chapter 4 we present the architecture of our proposed detection mechanism, and describe the implementation in detail. The Chapter 5 provides the results obtained during the experiments, showing the detection rate and the performance overhead on a PLC environment. Finally, in Chapter 6 we analyse the shortcomings of our defense and the possible future works and improvements.

Chapter 2

Related work

In this chapter we summarise the state of the art about security of embedded systems at the time of writing. First, we discuss about the attacks of the recent years, showing how the embedded systems security concerns are rising. Next, we analyse the defense mechanisms currently available in the literature, realising that they are still in a very early stage of their life. Finally, we consider the existing attacks and defenses targeting the hardware level, as Pin Control Attack does, showing that they are currently not applicable to embedded systems.

2.1 Attacks

In the past few years we have seen several attacks targeting embedded systems: most notably the infamous Stuxnet [4] targeting an Iranian nuclear facility in 2010. More recently Grandgenett et al. [5] analysed the authentication protocol between the RSLogix 5000 software and the PLC, based on a simple challenge-response mechanism. Since the protocol lacks freshness in its messages, is vulnerable to replay attacks, through which an attacker could repeat privileged commands to the PLC. Furthermore, they found that both the decoding of the challenge and the encoding of the response use an RSA-2048 key which is hard-coded in the RSLogix software, and it is actually valid for any Rockwell/Allen Bradley PLC. This indicates how the security mechanisms of these systems often have a really poor design, if any.

Papp et al. [6] analysed the existing attacks on embedded systems, relying on the proceedings of security conferences, with a focus on computer hacking, and on the Internet for media reports, blogs and mailing list. They built a taxonomy based on five dimensions: precondition, vulnerability, target, attack method and effect of the attack, showing that the threats to embedded systems are similar to the ones that affect traditional IT systems. However, embedded systems still lack solutions and tools to address these issues, and many ongoing research efforts are trying to deploy and adapt them to the needs of this field.

For our purpose, we may classify the attacks found in literature using a simpler criterion based on the attack method. We may distinguish three main categories:

1. **Firmware modification**: all the attacks aiming to upload a malicious firmware version (or part of it) in the device belong to this category.
2. **Logic modification**: this category consists of the attacks that modify the PLC logic in some way. In this case the integrity of the firmware is not violated, but a malicious program, or logic, is inserted into the PLC to make it misbehave during the control process.
3. **Control flow modification**: it includes the attacks that alter the normal control flow of a process by leveraging classic programming vulnerabilities such as buffer overflow or expired pointer dereference.

We briefly report about these different kind of attacks in the following sections.

2.1.1 Firmware modification attacks

Almost all modern embedded systems provide a way to update the firmware, and the attackers could exploit this feature to upload its own malicious firmware. Basnight et al. [7] reverse engineered an Allen-Bradley ControlLogix L61 PLC firmware showing how to bypass the firmware update validation method and successfully upload a counterfeit firmware. Peck et al. [8] demonstrate how using commonly available software an attacker can write and load his malicious firmware into Ethernet cards of devices used in control systems, potentially infecting the entire industrial control system. Cui et al. [9] discovered a vulnerability in the HP-RFU (Remote Firmware Update) feature of LaserJet printers, that allows remote attackers to make persistent modifications to the printer's firmware by simply printing to it.

2.1.2 Logic modification attacks

Stuxnet [4] belongs to this category. Along with its several components, mainly used to replicate and control the malware, its core is essentially an infected version of a SCADA software library used to program the PLC itself. By hooking some of the library functions it is able to load infected code and data blocks into the PLC and hide itself from the operator. McLaughlin et al. [10, 11] evaluated some techniques and implemented a tool (SABOT) to infer the structure of a physical plant and craft a dynamic payload, allowing an attacker to cause an unsafe behaviour without having a deep *a priori* knowledge of the target physical process. Similar techniques might mitigate the precondition needed by an attack, making it even more viable. Beresford [12] showed how the PLCs and the protocols used for communication in control systems were built without any security in mind, and demonstrated that they are affected by many vulnerabilities which may also enable the attacker to know the current configuration and rewrite the PLC logic. More recently, Klick et al. [13] used an internet-facing PLC as a network gateway by prepending a backdoor, made of a port scanner and a SOCKS proxy, to the existing logic code of the PLC. They developed a proof-of-concept tool called *PLCInject* to demonstrate their approach and measure the effects on the network.

2.1.3 Control flow modification

Many recent advisories [14–19] from ICS-CERT (Industrial Control System Cyber Emergency Response Team) report about various programming vulnerabilities affecting both PLC firmwares and control softwares. Most of them allow remote code execution and could be exploited without requiring particularly high skills. The vulnerabilities discovered by Beresford [12] also allow the attacker to insert a payload into the logic and subvert the control flow to execute malicious code. Nevertheless, the majority of the PLCs run the applications with root privileges, so it is quite simple for an attacker to get a root shell. One of the most dangerous kind of control flow attacks consists of ROP (Return-Oriented Programming) techniques, or similar variants [20, 21] which leverage different sequence of instructions equivalent to a return instruction. Since code vulnerabilities may affect embedded systems [12, 16–19], ROP techniques are applicable as well. Furthermore, due to the limitations imposed by these systems, is even more challenging to defend against them.

2.2 Defenses

Even though many incidents in SCADA systems occurred in the past decades [22, 23], the scientific community, together with PLC producers (Siemens, Hitachi etc.) and antivirus producers (Symantec, Kaspersky etc.), started to explore the security concerns of these systems only after the discovery of Stuxnet malware in 2010.

Since the communication protocols are the most vulnerable area in embedded systems, many efforts have been directed to *network-based* defenses [24]. In 2013, Clark et al. [25] designed a defense scheme against Stuxnet in which commands from the system operator to the PLC are

authenticated using a randomised set of cryptographic keys. Hadiosmanovi et al. [26] proposed a semantic-aware intrusion detection system, which is able to build a prediction model by observing the values of the process variables from the network communication, and then detect unauthorised changes with respect to the model.

In our work, however, we will focus on *host-based* defenses, in which the protection mechanism resides in the embedded system itself. Similarly to what we did for attacks, we can divide host-based defenses into the corresponding categories:

1. **Firmware integrity**: includes all the available techniques for preventing or detecting any malicious change in the PLC firmware.
2. **Intrusion detection**: host-based intrusion detection systems responsible for detecting rootkits or malicious software that could tamper with the normal process controlled by the PLC.
3. **Control flow integrity**: all the available mechanisms to defend against control flow attacks, like control flow integrity techniques or anti-ROP defenses, belong to this category.

2.2.1 Firmware integrity

In order to detect malicious modifications in the firmware of embedded systems, Wang et al. [27] proposed ConFirm, a low-cost technique, embeddable into the bootloader, based on measuring the number of low-level hardware events that occur during the execution of the firmware. To count these events, ConFirm leverages a set of special-purpose registers, the Hardware Performance Counters (HPCs), which readily exist in many embedded processors. The approach is divided into two phases: offline profiling and online checking. The offline profiling is executed before the system is deployed, and consists of registering the HPC signatures of code paths from a clean copy of the targeted firmware. The signature database is then embedded into the bootloader together with the ConFirm payload executed in the second phase. During the online checking, the same monitored paths are measured and compared to the golden references. Although this technique raises the bar for firmware modification attacks, if an attacker is able to reverse engineer and modify the bootloader, which usually has some update procedure as well, then the entire mechanism could be circumvented.

Other approaches that could defend against firmware modifications are based on a Trusted Execution Environment. While this kind of technology is commonly deployed in more capable systems, such as desktop or enterprise, the most of the embedded systems need a solution with lower resource requirements. The TrustZone Technology [28] enables trusted computing for ARM platforms by extending the hardware architecture, essentially the system bus, the processor core and the debug infrastructure, with security-aware components. Furthermore, Koeberl et al. [29] proposed a hardware-enforced security architecture named TrustLite, which is able to provide trusted computing capabilities on resource-constrained embedded devices without requiring CPU security extensions.

A trusted computing architecture can also enable attestation, through which a system, called verifier, can verify the integrity state of another system, called prover, which should provide a cryptographic proof of its integrity. Similar technologies could also be used to design a secure firmware upgrade mechanism. The PLC vendors need the possibility to provide firmware updates for their own devices, most likely when some vulnerabilities are discovered after release. Fuchs et al. [30] discussed the benefits of Trusted Computing Group's Trusted Platform Module (TPM) 2.0 as a security-anchor for embedded systems, and proposed a proof-of-concept implementation of advanced remote firmware upgrade for embedded systems relying on the unique features of TPM 2.0.

A different approach is proposed by Lee, B. & Lee, J. [31], which leverages blockchain technology to securely check firmware versions, validate the correctness of firmware, and download the latest firmware for the embedded devices. Even though their work is focused on intensively interconnected embedded systems, such as in an IoT environment, the increasing number of internet-facing controllers let us suppose that may be worth investigating whether this technology could be applied to PLCs as well.

2.2.2 Intrusion detection

The PLC logic is usually executed in a scan cycle, in which the PLC reads the inputs from the sensors, executes the logic and writes the outputs to the actuators. Zonouz et al. [32] devised an approach capable of detecting whether a PLC logic could violate physical plant's safety requirements. Their technique is based on logic binary code analysis and model checking, and could be integrated in the PLC runtime itself, so that the check is made every time a new logic is uploaded to the system. Garcia et al. [33] leveraged the advanced computational power of embedded hypervisors, that could be coupled with modular embedded controllers, to provide both a memory verification solution and an intrusion detection system from within the PLC itself. The embedded hypervisor provides a library directly accessible from the PLC scan cycle either synchronously or asynchronously, and the hypervisor and the PLC can communicate through shared memory. Their approach may be extended to integrate any kind of security solutions inside the PLC. Cui et al. [34] proposed a host-based defense mechanism called Symbiotic Embedded Machines (SEM), designed for injecting intrusion detection functionalities into embedded system firmware code. The injected Symbiotes are basically code structures that will co-exist with the legacy firmware, sharing computational resources with it while protecting it against code modification. The SEM injection process is randomised, so that the firmware payload is divided into slices at random locations, called control-flow intercepts. Each code slice has its own Symbiote, which is basically a checksum of the corresponding section. When an intercept is reached by the firmware execution, the control goes to the Symbiote Manager which verifies the current portion of the code against the stored checksum. In this way SEM executes itself alongside the original OS while remaining stealthy and causing minimal overhead.

Moreover, Trusted Computing could also be used to enable malicious code detection capabilities at runtime. The main problem with this technology is to deploy it into embedded systems without impacting its limited resources and real-time constraints. Seshadri et al. [35] proposed a software-based attestation technique, named SWATT, to verify the memory contents of embedded devices and establish the absence of malicious code through a challenge-response protocol. SWATT is designed in a way that the minimum change to the code will result in a detectable delay. However, further research [36] has shown that this time-based approach is hard to design for embedded systems, and that some attacks are still possible.

Finally, another potential solution for intrusion detection in embedded controllers is based on power fingerprinting [37]. It consists of a physical sensor through which is possible to analyse and collect statistical data about power consumption and electromagnetic emissions, determining whether or not the process deviates from the expected operation model. Even though it is a powerful mechanism and does not interfere with critical operations, it provides limited support for forensic analysis once the attack has been detected, so this technique should be applied as a part of a security solution.

2.2.3 Control flow integrity

Control flow hijacking is one of the most used attacks to computer systems, because it usually leverages programming errors that are actually much more common than they should be. As more and more sophisticated control flow modification techniques were discovered, new defenses have been proposed, but only some of them are applicable to embedded systems. In 2012 Reeves et al. [38] proposed Autoscopy Jr., an intrusion detection system designed for embedded systems, which is focused on detecting control flow alterations instead of malicious code insertions. Its approach consists of two phases: the learning phase and the detection phase. During the learning phase it collects control flow information from the function pointers used within the kernel, building a data structure, named Trusted Location List (TLL), which will be used in the second phase. The data structure basically maintains, for each monitored function pointer, a list of function addresses reachable from that pointer together with other control flow information (e.g. valid return addresses). During the detection phase it continues monitoring direct and indirect calls, and it generates an alert if an unknown function is reached from a monitored function pointer.

Habibi et al. [39] proposed a defensive technique for ARM architecture, named DisARM, effective against both code-injection and code-reuse attacks. Relying on the assumption that buffer overflow attacks lead the execution to a different return address than expected, they designed a mechanism for verifying the actual return address at runtime, thus protecting any potentially vulnerable program. First, they look for all the critical instructions contained into the program, defined as the ones that take input from the stack and affect the program counter directly or indirectly (e.g. through the link register). Second, they modify the binary code by putting a verification block before each critical instruction, so that the execution is stopped whenever a control flow manipulation is attempted through the stack.

Many other control flow integrity (CFI) solutions for embedded systems rely on hardware modifications in order to require smaller overheads. Francillon et al. [40] presented a technique to protect low-cost embedded systems against malicious manipulation of their control flow by using a simple hardware modification to divide the stack in a data and a control flow stack (or return stack). The access to the control flow stack is restricted only to return and call instructions, thus implementing an Instruction Based Memory Access Control (IBMAC) in hardware. Abad et al. [41] proposed a hardware-based security approach with predictable overhead for embedded real-time systems. They perform CFI checks on a real-time task by adding an On-chip Control Flow Monitoring Module (OCFMM) to the processor core with its own isolated memory unit. OCFMM monitors the run-time control flow and compares it to the stored Control Flow Graph determined in advance. Davi et al. [42] designed novel security hardware mechanisms to enable fine-grained CFI checks, based on three security policies. First, each function call enforce the processor to switch to a new state in which the only accepted instruction is a CFI instruction, thus restricting function calls to only target valid function entry points. Second, return instructions can only target a valid return of a function whose CFI instruction is active. The CFI instructions are identified by labels, managed through a CFI Label State Table embedded in the program data memory. Third, behavioural heuristics are used to cover typical patterns of ROP attacks. Afterwards, they provided an implementation for Intel Siskiyou Peak and SPARC, named HAFIX [43], demonstrating its security and efficiency in code-reuse protection while incurring only 2% performance overhead. Another hardware-based approach has been presented by Das et al. [44]: a fine-grained CFI at a basic block level, named basic block CFI (BB-CFI). A basic block is defined as a sequence of instructions, having a single entry and a single exit point. The policies used by BB-CFI are defined as follows: first, each function call can only target the first basic block of the function; second, each return can only target the basic block following the function call; third, indirect jumps must target a starting address of a basic block. Their approach is divided into two steps: 1) offline profiling of the program to collect control flow information data, and 2) runtime control flow checking. The control flow checker has been implemented on FPGA as a proof-of-concept, showing $< 1\%$ performance overhead, a small dynamic power consumption and a very small area footprint.

Finally, the attestation mechanism could be used to address control flow integrity as well. Abera et al. [45] presented the design and the implementation of Control-FLow ATtestation (C-FLAT), based on ARM TrustZone hardware security extensions. In their model, a verifier wants to attest runtime control flows of an application module on a remote embedded system, the prover. First, the verifier has to generate the Control Flow Graph from a clean binary of the application, storing the measurements of all the possible control-flow paths. Then, it sends a challenge to the prover, containing the name of the application and a nonce. The prover executes the application, computes a digital signature over the challenge and the executed path, and sends it back to the verifier for validation.

2.3 Hardware-level attacks and defenses

At the time of writing, we are not aware of any attack in literature which targets I/O of embedded systems. Neither any defense mechanism specific for embedded systems (or for PLCs) has been designed to protect I/O configuration. Although hardware level attacks are quite rare due to their complexity, they are very powerful and stealthy, because they are very close to the hardware. Here we provide an example of an existing hardware-level rootkit, and its corresponding defense. Even

though they are not applicable to embedded systems, they are still interesting for our purpose, due to their similarity with Pin Control Attack and Defense, respectively.

Embleton et al. [46] implemented a hardware-level rootkit which they call System management Mode Based Rootkit (SMBR). System Management Mode (SMM) is an isolated execution mode of the x86 architecture designed for low-level system control functions, such as power management. When the CPU is running in SMM mode, it uses a private memory space, it is completely non-preemptible, and it lacks any concept of privilege level and memory protection mechanisms. Thus, the code running in SMM mode is capable of directly accessing the underlying hardware. This could be very attractive for malicious users, as they demonstrated by implementing a chipset level keylogger and a network backdoor which interacts with the network card to send logged keystrokes to a remote machine via UDP stealthily. Their technique can be divided into two main steps. First, the rootkit needs to install its own SMM interrupt handler in place of the OS handler into a particular portion of memory called System Management RAM (SMRAM). Depending on the target system, this could require to modify and reflash the BIOS. Second, it re-programs the I/O APIC (Advanced Programmable Interrupt Controller) of the target peripherals in order to enable SMM mode on their interrupt signals. This is achieved without hooking the Interrupt Descriptor Table, making the rootkit undetectable. Once the target interrupt is routed, the rootkit SMM handler is called whenever the interrupt occurs (e.g. at each keyboard event). After its job is done, the rootkit may forward the interrupt back to the previously assigned handler, or may hide it, depending on its purpose.

To overcome this SMM rootkit, a defense framework called IOCheck has been proposed in [47]. The IOCheck framework leverages System Management Mode as well. It checks the integrity of the I/O configuration and the firmware of I/O devices at runtime. It also locks the System Management RAM (SMRAM), which is the portion of memory where the SMM handler is stored. Once locked, SMRAM cannot be overwritten until a reset occurs. Furthermore, the BIOS firmware is securely stored in SMRAM at boot time, such that its integrity can be verified at runtime. In order to protect the BIOS against offline modifications as well, a Trusted Platform Module is needed.

Both SMBR and IOCheck leverage SMM, which is an execution mode existing on x86 architectures. Since x86 architecture is rarely used in embedded devices, we can state that those techniques are not applicable to PLCs. Furthermore, even if x86 is used, switching to SMM mode would take about 4 milliseconds, as reported in [47]. This might be a low performance overhead for many computer systems, but it is certainly not acceptable for real-time embedded systems like PLCs.

Chapter 3

Attack analysis

Before describing the attack, a deeper analysis of the architecture of the target system is needed. Note that, although this paper is focused on PLCs, the architecture described in the next section is still valid for almost any embedded system. After having a proper knowledge of the underlying architecture, we can go deep into the description of the design of the I/O Attack. We consider the idea behind the attack, showing how it is able to evade the currently available detection mechanisms. Next, we go further and describe some implementations of the attack, extending its applicability to a real Programmable Logic Controller. Based on our architecture analysis and our tests, we can finally demonstrate that the attack is actually viable on real PLCs, even on a higher abstraction level.

3.1 Embedded architecture

As briefly reported into Chapter 1, PLCs use I/O interfaces to communicate with sensors and actuators, and in general with any external device. Digging into their architecture, we know that PLCs are usually based on a so called System on Chip (SoC). A SoC is basically an integrated circuit made of a microprocessor, a memory block and a set of peripheral controllers all enclosed together in the same chip substrate. Thus, the SoC technology provides fully capable computers having both very small size and low power-consumption. A SoC usually comes with a set of connections, also known as *pins*, usually soldered to a printed circuit board to facilitate interconnection with external modules. Many types of pins may be included in a SoC, having different purposes (power, clock, I/O, etc.). An example of such a system is the Raspberry Pi board shown in Fig. 3.1, based on a Broadcom System on Chip. Actually almost all of the embedded systems use a SoC with similar boards, each one with its own size and configuration.

In order to accommodate many board implementations from different companies, each pin (or group of pins) of a SoC may have multiple configuration and operating modes. The configuration of the pins is managed by the pin controller, a particular subsystem of a SoC. Through a specific set of registers belonging to the pin controller, the system can configure the operating mode of the pins, such as their input or output mode. These registers are typically accessible through a particular memory region called I/O memory. Such a kind of I/O access is known as *Memory Mapped I/O*.

The features provided by a pin controller can be grouped into two main categories:

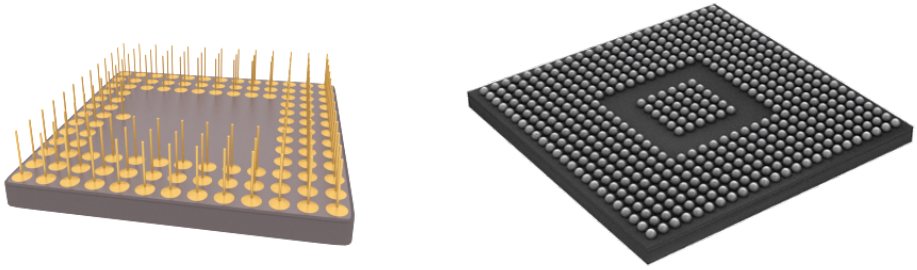
- **pin configuration**: allows the system to change some electrical properties of the pins, such as direction, event detect, interrupt, etc.;
- **pin multiplexing**: each pin of the SoC may have many usages, also known as *alternate functions*, depending on what is needed by the external board. The pin multiplexing feature enables the system to specify which type of function is needed on each pin.

Figure 3.1: Raspberry Pi [48] with Broadcom System on Chip

As the I/O attack is a very low-level attack, it is necessary to dig further into the electrical world to know how these I/O interfaces work.

3.1.1 SoC pins

I/O interfaces of a System on Chip provide a connection between internal modules and external electronic devices. As shown in Fig. 3.2, they are physically visible from the outside of the chip package, usually in the form of pins (a) or soldering balls (b).



(a) (b)

Figure 3.2: I/O connections packaged as (a) Pin Grid Array and (b) Ball Grid Array

Internally they are connected to the silicon die through bonding wires, and are managed by a specific I/O circuit which may vary according to the specific chip. Although there are many different implementations, almost all of the available SoCs have I/O ports with very similar functionalities. For our purpose, we can describe them in an implementation-independent manner by using simplified generic schematics.

Pin configuration

The schematic depicted in Fig. 3.3 helps us to discuss about the first set of features: pin configuration. The operation mode described in this section is also known as General Purpose I/O (GPIO).

Apart from the protection diodes that serve as shield against input currents lower than V_{SS} or higher than V_{DD}, the circuit is divided into two different parts: one for output and one for input.

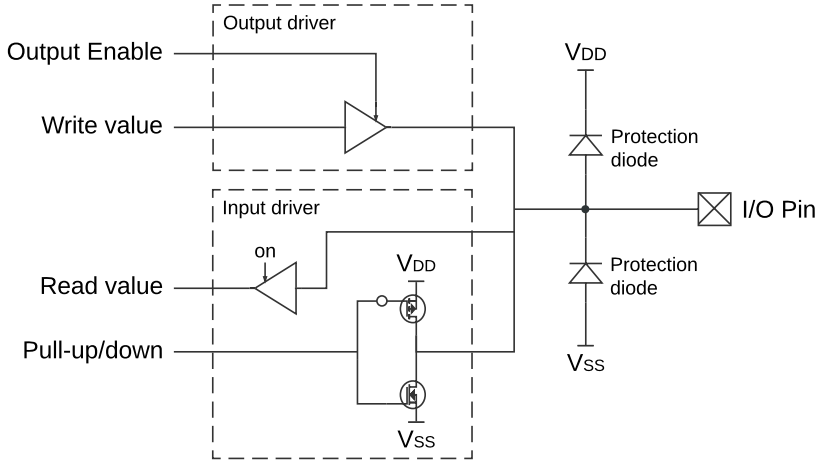


Figure 3.3: General Purpose I/O pin configuration circuit

- **Output driver**: The output module is basically a buffer controlled by an output enable signal. This signal controls the direction of the pin (input or output). If it is high, then the pin is in output mode and the value coming from a write operation goes through the buffer to the actual pin. If it is low, the pin is in input mode and the write signal is blocked, so it is not possible to change the external value of the pin from inside anymore.
- **Input driver**: The input driver has a similar buffer to read the value, usually having hysteresis capability which is useful for filtering unstable external values. Since the read buffer is always active, the input value is always available, even if the pin is currently working in output mode. The reason for this is merely physical: even if the external value was blocked by the buffer, one would always get a value by reading the input signal, because a value is nothing but an interpretation of the voltage level on a wire. When the pin is set as input, the pull-up/pull-down network enables the user to have a "default" value on the pin, namely a defined state maintained while the pin is not actively driven from outside. This feature is useful to avoid so called "floating" inputs.

For Pin Control Attack, what is more interesting about the circuit in Fig. 3.3 are the following two properties:

- there is no checking about the input state, so it is possible to perform a read even when the pin is in output mode;
- vice versa, it is not possible to write to a pin which has been configured as output.

Furthermore, it is also possible to drive the pull-up/down network, factually disturbing the real value of the I/O pin in an unpredictable way. Virtually, it is even possible to interfere with the pin value by means of external electromagnetic fields. In these cases the effects strongly depends on the actual implementation of the printed circuit board as well as on the external components connected to the I/O pin.

Pin multiplexing

Inside a chip, each I/O pin may be connected to more than one device, which can be selected depending on the application (i.e. the way of soldering or wiring the package into an electronic board). This SoC feature is known as pin multiplexing (also ball multiplexing, pad multiplexing, alternate functions). Even though pin multiplexing is designed for hardware configuration, in almost all modern chips it is possible to change the function at run-time.

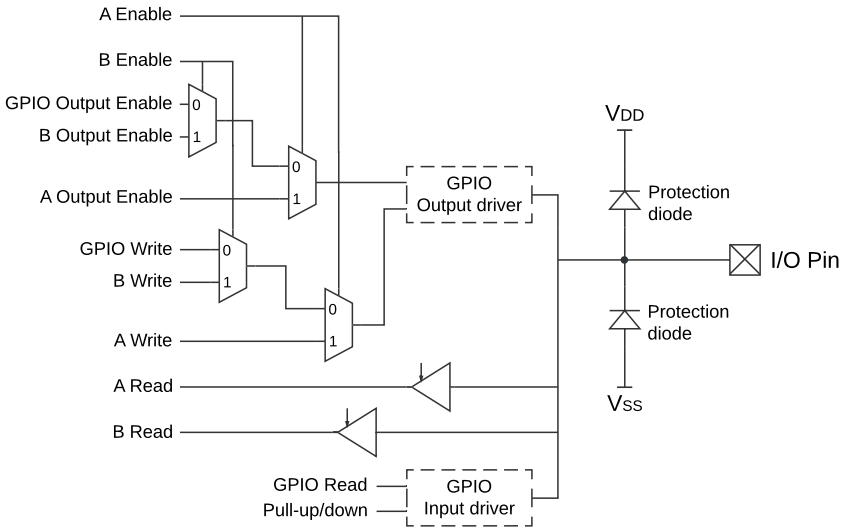


Figure 3.4: Generic I/O pin multiplexing circuit

Fig. 3.4 shows a possible hardware implementation of pin multiplexing. The I/O pin in the figure is connected to two different peripherals inside the chip, namely A and B, and it is also accessible through basic GPIO as described in Section 3.1.1 above. The access to the GPIO output driver is regulated by two in cascade multiplexers for each signal of the module. If the peripheral A is enabled, then the signals driven by A go through the multiplexers and can drive the actual value of the pin, while GPIO and peripheral B output signals are blocked. Instead, if only peripheral B is enabled then only its signals are able to reach the I/O pin. Note that in this last case the peripheral A should not be enabled, because A multiplexers have precedence against B ones. Thus, the cascading of multiplexers actually implements a priority mechanism between peripherals A and B. If neither A nor B are enabled, then no alternate function is active for the I/O pin and it could be driven by GPIO signals. Each peripheral may also have its own dedicated input line, to get values from the I/O port in the same way as GPIO does.

For our purpose, at least two interesting properties can be inferred from pin multiplexing schematic of Fig. 3.4:

- it is possible to block output signals from peripherals by simply changing the multiplexing configuration;
- the GPIO value can be read at any time, independently from the current multiplexing state of the output.

3.1.2 PLC architecture

As introduced in Chapter 1, Programmable Logic Controllers are a particular kind of embedded systems, designed to work into harsh environments and to control a physical process, typically an industrial or other critical processes. In this section we examine the hardware and the software architecture of a PLC. This analysis will be needed later to better explain the threat model and the implementation of the attack.

Hardware architecture

Since it is built for working into a rough environment, the internal hardware of a PLC is shielded and not directly visible as the Raspberry Pi one. Fig. 3.5 shows an example of a basic PLC, provided by Wago. This model will be later used for our tests. Note that, although we provide

this specific example, the basic concepts described here are still valid for most of the PLC on the market, unless otherwise specified.

Figure 3.5: PFC200 Programmable Logic Controller from Wago

Internally, the PLC has a System on Chip whose architecture is substantially equivalent to the one discussed in the above sections. Therefore, the same concepts are applicable to PLCs as well. Anyway, from our analysis we found that their architecture could be a bit more complicated with respect to a plain SoC. Since the environment in which PLCs work may greatly differ case by case, they are designed to be as versatile as possible. Vendors know that their PLC could be deployed in many different physical processes, possibly having completely diverse sensors and actuators to interact with. Clearly, it is unreasonable to put every possible I/O interface on the same SoC. For this reason, most of the available PLCs comes with the possibility of connecting a certain number of external components called *I/O modules*. Each I/O module contains its own embedded SoC, responsible for a specific subset of I/O interfaces (e.g. digital input/output, analog input/output, pulse-width modulation, communication, etc.). This hardware architecture is summarised in Fig. 3.6.

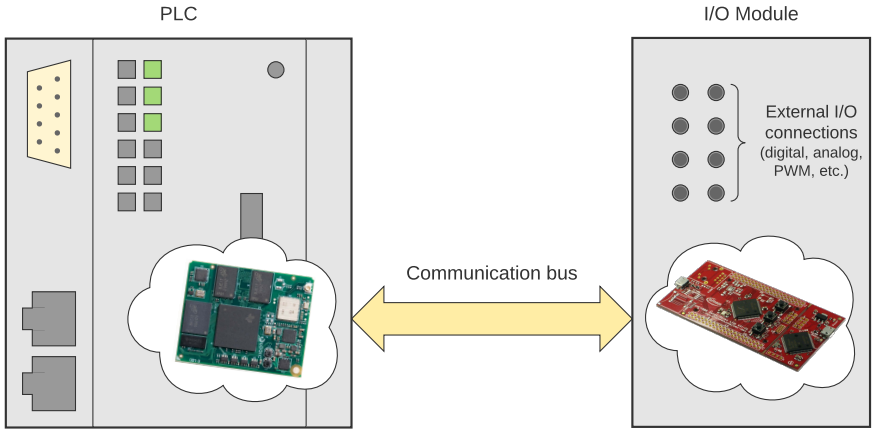


Figure 3.6: PLC hardware architecture

The main portion of the PLC communicates with the I/O modules via a system bus, which is, in turn, connected to the I/O interfaces of the PLC. This architecture with I/O modules adds one level of indirection between the targeted PLC and the external world, since there is another SoC in the middle. Anyway, the same concepts of Pin Control Attack can be applied to such an architecture as well. It is sufficient to consider that the I/O interface of the PLC, connected to the internal bus, is now the new direct target of our attack, while the final I/O becomes an indirect target. As we demonstrate in Section 3.3.2, by tampering with the I/O related to the communication bus it is possible to achieve equivalent results, proving that I/O Attack is still applicable on real PLCs.

Software architecture

The PLC typically comes with a real-time operating system (e.g. Linux with RT-preempt patch, VxWorks etc.), because of the time requirements imposed by its main task. Above the operating system, a software called *PLC runtime* is responsible for managing the execution state of the control process and regulating the access to the PLC. Through the runtime, the industrial operator can connect its terminal to the PLC and upload the desired control program, the *logic*. The PLC logic is the application code responsible for the control of the physical process, dealing with input and output interfaces. As shown in Fig. 3.7, this architecture can be divided into three layers, laying one above the other.

- **Firmware**: the lowest layer, which basically corresponds to the operating system kernel. Although they are typically two distinct parts, here we consider the bootloader also as "part" of the firmware, because in this context their separation is irrelevant.
- **Runtime**: a software which is part of the application layer above the firmware, and it is started by the operating system itself.
- **Logic**: the control program running within the runtime environment. Its execution is started and stopped by the runtime, and its code is dynamic: it may change whenever the industrial operator decides to upload a new control program to the PLC.

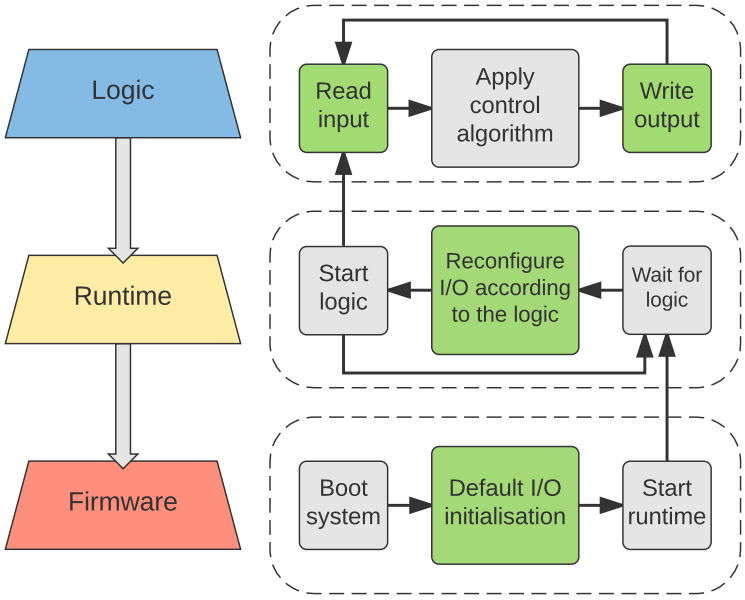


Figure 3.7: PLC software architecture

Fig. 3.7 also summarises the execution flow of each layer, from bottom to top, highlighting (in green) the tasks directly related with the I/O.

When the firmware is loaded into memory and the system is booting, one of the early operation performed is known as I/O initialisation sequence. In this phase, the drivers within the kernel configure their own I/O registers with their default values. I/O initialisation includes both pin configuration and pin multiplexing. Normally, pin multiplexing is only performed at boot time, and, even though it is not forbidden, pin multiplexing is never modified at run-time. Conversely, pin configuration may be changed later at run-time. Thus, depending on the implementation, the configuration written during I/O initialisation may correspond or not to the one desired by the final application.

After the I/O has been initialised, the firmware loads the main application needed by the controller: the runtime. The PLC runtime waits for a logic to be uploaded into the PLC from a terminal connected via the network interface. The execution flow reported in Fig. 3.7 for the runtime layer is a general process, including the case when a PLC is started for the first time. If a PLC is already in production, it is rarely restarted. However, when a reboot is needed (e.g. for firmware update), it may be configured to persistently save the current logic into a non-volatile memory and restore it after reboot. In this case the "Wait for logic" step in Fig. 3.7 can be simply skipped, because a logic may be already available from disk.

When a logic is available, from network or from disk, the PLC runtime analyses its content. The I/O configuration expected by a specific logic is bundled with the logic itself, because it is decided by the operator who has knowledge of the physical process and knows how sensors and actuators are connected to the PLC I/O ports. With respect to the hardware architecture shown in Fig. 3.6, a change in the input/output required by the operator is actually reflected into a configuration change of the external I/O modules. Whether this process involves a change in the I/O configuration of the PLC itself depends on the implementation and on the modification extend (e.g. if a new I/O module has been attached, this will probably require a change in the communication protocol and so on the I/O interface of the PLC). After the new I/O configuration has been applied, the logic can be executed. From an operating system point of view, the logic is running in the same context of the PLC runtime.

As briefly discussed in Chapter 1, the PLC logic executes the main *scan cycle*. For each iteration, it reads from inputs, executes the control algorithm, and writes to outputs. The control algorithm has been programmed by the industrial operator, and loaded through the interface provided by the PLC runtime. When the logic is running, input and output pins have already been configured by the runtime, and the logic assumes that the I/O configuration does not change during its execution. Both the logic and the runtime interacts with the I/O, but in different ways. The runtime deals with I/O configuration registers, while the logic performs read/write operations related to I/O values. Both kind of interactions, anyway, require the same privileges.

3.2 Attack Design

Given the properties discussed in Section 3.1, it is possible to misuse the registers used to configure I/O peripherals, and that is what Pin Control Attack does. This can happen at run-time, during the execution of the controller on a real process, without any reaction from the PLC runtime or the operating system, thus remaining completely stealth.

We can argue that this low-level I/O Attack is actually applicable on any System on Chip having the architecture described in Section 3.1. However, PLCs represent a particularly interesting target among all the embedded systems based on SoC. PLCs leverage the interfaces of a SoC to interact with sensors and actuators, having direct effects on our physical world. Therefore, if the I/O of a PLC is compromised, this constitutes a much more significant security and safety risk with respect to other non-critical embedded system. For this reason, PLCs represent one of the most attractive targets for I/O attacks.

In this section we focus on the design of I/O attack. First, we summarise the host-based detection mechanisms taken into account during the attack design and the techniques used to evade them. Second, we analyse the attack itself in more detail. The goal of this chapter is to help

us having a better understanding of the attack, and to provide a detailed threat model on which our defense design will be based on.

Pin Control Attack has been designed to be different from previous attack techniques, and to circumvent the off-the-shelf host-based detection systems. The authors of I/O attack have identified two main defensive mechanisms whose properties makes them more easily applicable to PLCs. Here we want to generalise and consider them in our analysis, clarifying how and when the I/O attack becomes applicable and interesting for attackers. We also show how these defenses are circumvented by the attack, describe the available techniques. We do not repeat here the detailed description made in [1]. We only want to have basic concepts definition, in order to be able to focus on our further analysis and implementation of the attack.

3.2.1 Defense analysis

At this point, one could ask why descending to the lowest level possible to conduct an attack, when a lot of easier techniques (e.g. function hooking) may achieve equivalent results. The point here is that many producers of embedded systems and PLCs already started to move towards a more security-aware production. As considered in [1], in order to be applicable to PLCs, a defensive mechanism should have at least the following properties:

- low CPU overhead: CPU resources are limited in embedded systems such as PLCs, which typically have hard real-time constraints;
- designed to run on an operating system: almost all of the modern PLCs have a real-time OS running on it;
- no hardware modification required: many solutions are hardware-based [29,40–42,44], making them not easily applicable;
- no virtualisation required: the majority of embedded systems, like PLCs, does not support virtualisation; thus solutions like [33] are less attractive.

Given these considerations, we can list some of the applicable defenses:

- **SEM**: the Symbiote detection system described in [34], which protects the kernel from code modification attacks;
- **Autoscopy Jr.**: the lightweight intrusion detection system proposed in [38], effective against control flow manipulation inside the kernel.

The above detection systems still have some shortcomings, leveraged by Pin Control Attack to remain stealth. First, they are based on a comparison with golden references (the symbiotes and the TLL, respectively) taken from a subset of the entire kernel space. If the attacker is able to find a portion of the kernel space which has not been considered by the detection system, then it can be circumvented. For example, Autoscopy Jr. can be bypassed if the attacker is able to craft a duplicate of the kernel functions it needs. If a duplicate function is used instead of the original version, Autoscopy Jr. does not throw an alert, because this function is not listed in the TLL at all. Second, the references used by both defenses are *static*, which means that they are based on previous analysis of the immutable (static) portion of the system. Thus, an attacker could still use dynamic memory, whose content changes during run-time, to conduct its own attack. In the case of SEM, only static code regions (which are immutable) are taken into account, so any malware placed in dynamic memory will not be detected. Autoscopy Jr., instead, considers the function pointers used inside the kernel, including the ones inside the System Call Table and other similar tables. Therefore, it is able to detect only *data hooks* but not *code hooks*; in other words, any direct code modification (e.g. of the kernel text) is still possible.

Anyway, these two defenses may be used in combination, thus providing a host-based IDS able to detect both code and data hooks. Even if they are deployed together, however, attacks

through dynamic memory and without function hooking are still possible, since monitoring dynamic memory is a far more complex issue. One of the attack implementations (see Section 3.3) leverages exactly this kernel dynamic memory to reach its purpose.

Host-based detection mechanisms like the ones discussed above will likely be deployed into many commercial products within the next years [49, 50], raising the bar for attackers. Therefore, I/O level attacks will become more and more interesting as long as classical techniques (e.g. function hooking) are overcome.

3.2.2 Threat model

In our work, we assume that the system is protected at least by the HIDSs described in Section 3.2.1, or equivalent. Hence, 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 HIDSs. 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 already discussed in [1], many ICS-CERT advisories have shown that PLCs have vulnerabilities that could lead to malicious code execution [13–18], and they may affect PLC runtime software as well. Thus, obtaining the same PLC runtime privilege level is feasible on real systems.

As in [1], 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 [4], 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 [10, 11] 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 which do not need an a priori knowledge of the I/O mapping anymore.

3.2.3 Pin Control Attack

Based on the previous assumptions, Abbasi et al. [1] crafted Pin Control Attack, which is capable of manipulating the physical process controlled by a PLC without requiring any firmware or logic modification. The idea is that the attacker operates at the lowest level possible, targeting the interaction between the firmware and the I/O, as shown in Fig. 3.8. For this reason, the attack is also known as I/O attack. Despite its crucial function in embedded systems, I/O hardware architecture and I/O drivers are currently designed without taking into account any concept of security, assuming that I/O is reliable. Unfortunately, given the properties inferred in Section 3.1 from a generic hardware architecture, this is often not the case.

As described in [1], the attack is designed to evade off-the-shelf HIDSs presented in Section 3.2.1. In particular, it leverages kernel dynamic memory (in one implementation variant) or it uses a simple user code (in a second variant). Both variants are able to circumvent the above detection mechanisms.

The authors of [1] have found at least two different attack vectors: *pin configuration* and *pin multiplexing*. Both kind of attacks leverage the SoC properties described in Section 3.1. The attack is performed by misusing the control registers used to program the I/O peripherals, writing malicious values into them. If the attack targets I/O configuration registers, it is a pin configuration attack, otherwise, if the targeted registers are related to the multiplexing state of the pins, then it is a pin multiplexing attack. In some architectures, I/O configuration and multiplexing is achieved by using the same set of registers (e.g. different bits of the same register may have different purposes).

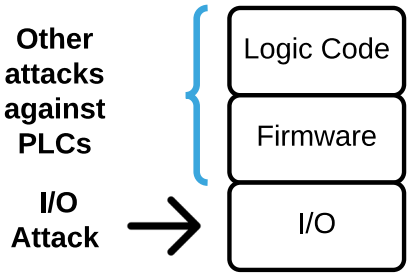


Figure 3.8: I/O level attack, from [1]

In both cases, the main principle behind the attack is something that they called *memory illusion*, which is represented in Fig. 3.9. In a system equipped with a Memory Management Unit (MMU), that is the most common scenario for PLCs, each process cannot directly access to physical address space. Every memory access is mediated by the MMU, which creates an abstraction called *virtual address space*, or *virtual memory*. When a process wants to access physical memory, it has to request a mapping to the system between physical address space and virtual address space. In this way, every process has its own "virtual view" of the physical memory. This is valid also for I/O memory, which is part of the physical memory. Once a mapping has been established, the I/O can be managed by accessing the mapped virtual memory. Thus, as Fig. 3.9 outlines, an attacker

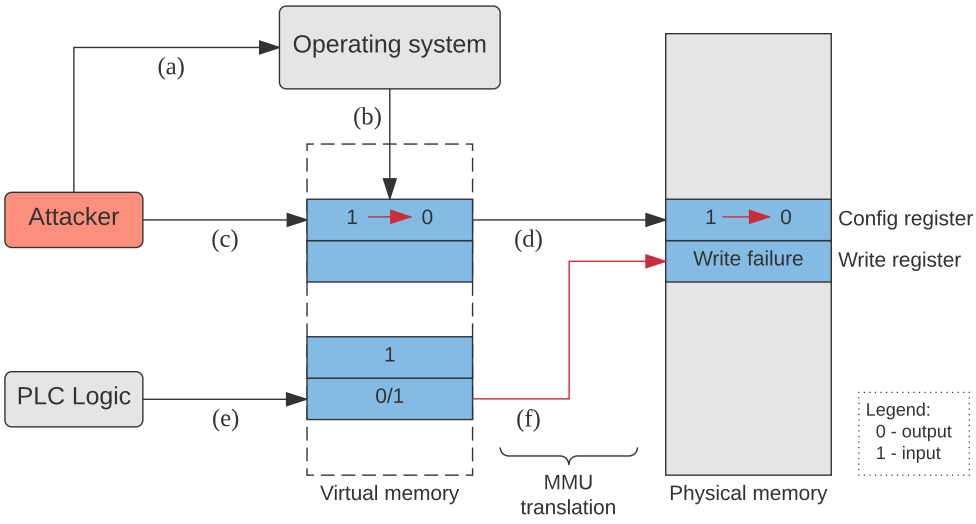


Figure 3.9: Attack memory illusion

can do the following: first, it has to request a mapping for I/O memory (a); once a virtual mapping is provided by the operating system (b), the malicious process is able to alter the pin configuration (e.g. change an output pin to input) through virtual memory (c). This change is immediately reflected into physical memory via MMU translation (d). When the PLC logic, which is a normal process in the system, tries to update the value of the pin through virtual memory (e), the target address of the write instruction arrives to the MMU, which translates it to the corresponding physical address. Unfortunately, since the pin has been changed to input, the execution of the instruction does not actually affect physical memory (f). Therefore, the write silently fails, but no signal error is reported back to the CPU, and from the process virtual perspective everything went correctly. As explained later in Chapter 4, the fact that the virtual address of the instruction actually reaches the MMU will be a crucial mechanism for the design of our defense.

As reported in [1], only by leveraging the configuration of pins they crafted an attack which is

able to do the following:

- reconfigure an I/O pin as input when the PLC logic is attempting to write;
- reconfigure an I/O pin as output and write a malicious value when the PLC logic is attempting to read.

In this way, the attacker can disable specific output operations and feed the PLC logic with malicious input at the same time. In other words, the whole PLC logic is considered as a black box. Instead of altering the logic code, the attack simply controls input and output values by means of the above operations. Whether this approach can lead to useful results (for the attacker) is highly dependent on the actual PLC logic and, of course, on the attacker purpose. In Section 3.3.1 we show that this mechanism may be very interesting for the attacker, and we also discuss about some hardware limitations of this approach.

Moreover, we imagine that many other attack possibilities may exist. An attacker can tamper with the I/O configuration in many other ways depending on the specific I/O peripheral and on its features (e.g. event detect, interrupts, clock signals, etc.), and the effects of the attack are unpredictable. If the attacker can analyse the reference manual of the target SoC (mostly publicly available) and it has enough competence and time, then its fantasy becomes the only limit to I/O level attacks. To give an idea of what such an attack can do, imagine a simple SoC pin which can be multiplexed between a memory controller and a PWM controller. If this pin is actually connected to an external memory, thus multiplexed to a memory controller, a pin multiplexing attack can lead to dangerous signals sent from a PWM controller to a memory module, which may burn the memory. Furthermore, if a subset of pins is multiplexed to a specific controller (e.g. I2C, SPI, DMA, etc.), the control registers of the device can be targeted as well, obtaining an indirect effect on the corresponding I/O pins.

On the other side, from our analysis on a real PLC, we found that another attack vector is viable at a different abstraction level than hardware configuration. Commonly, the I/O peripherals are controlled by operating system drivers, which in turn are used from user space applications like the PLC runtime. As we show in Section 3.3.2, an attacker can target these drivers, or even these applications, to obtain equivalent effects without dealing with the underlying hardware and I/O registers directly.

3.3 Attack Implementation

In this section we first describe the available implementation presented in [1], which was the starting point of our analysis. Next, we introduce a possible attack implementation on a real PLC. Both implementations are designed for Linux-based systems, and they have two different variants: kernel side and user side. Each variant imposes different requirements for the attacker, which will be discussed in more detail in the next sections.

3.3.1 Raspberry Pi

The system used for our experiments with the first implementation is a *Raspberry Pi 1 Model B* based on an ARMv6 architecture, running the following kernel:

```
Linux raspberrypi 4.4.22+ #912 Mon Sep 26 19:00:13 BST 2016 armv6l GNU/Linux
```

The Raspberry Pi 1 features a Broadcom System on Chip named *BCM2835*. With reference to the BCM2835 documentation [51], we set up the system I/O by connecting a button as input, an LED and a servo motor as outputs. Button and LED are accessed via 2 GPIO interfaces, one configured as input and one as output, respectively. To connect the micro servo, we needed an external PWM controller. The controller is attached to the system through the I2C interface, which requires two

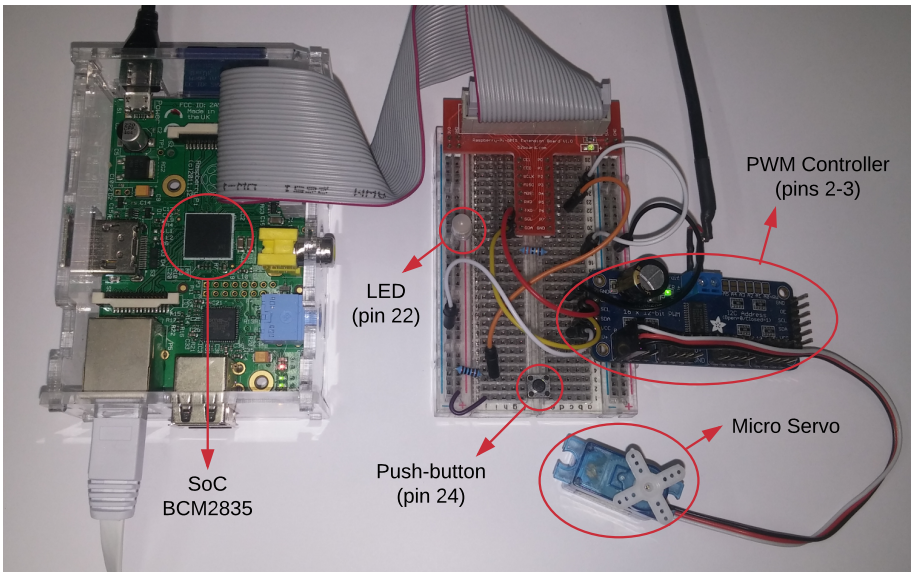


Figure 3.11: Raspberry Pi system used for experiments

Pin	Multiplexing	Configuration	Connected to
24	GPIO	Input	Button
22	GPIO	Output	LED
2	I2C SDA	-	PWM controller (servo)
3	I2C SCL	-	PWM controller (servo)

Table 3.1: I/O configuration of our Raspberry Pi testing system

pins: one for the data line, *Signal DAta* line (SDA), and one for the clock line, *Signal CLock* line (SCL). The resulting system on which we conducted our experiments is shown in Fig. 3.11, and its I/O configuration is summarised in Tab. 3.1.

To enable PLC capabilities on such a system, we installed a PLC runtime on it. In particular, we used *CODESYS Control for Raspberry Pi SL* provided by 3S-Smart Software Solutions GmbH [52]. Through the CODESYS Development System [53], we designed the PLC logic shown in Fig. 3.12, whose code is executed for each scan cycle. We chose a 10 ms scan cycle interval, however, since this PLC runtime is not real-time, the timing may be affected by small errors ($\approx 50\,\mu s$). At each scan cycle, the logic reads from pin 24 (push-button), and writes the outputs accordingly. The motor is driven with a sinus wave which makes it going forward and backward in the interval $[-45°, 45°]$. If the button is not pressed, which corresponds to value 1 (high), the LED is toggled every $2s$ and the motor completes a round in 2 s. If the button is pressed, that is value 0 (low), both LED and motor frequencies are multiplied by 4, and the new round time becomes 500 ms. The timing of the write operations, which is a multiple of the scan cycles, is controlled by software counters embedded into the logic. Thus, while the input is high the counters are updated at normal speed; while the input is low they get updated 4 times faster. Furthermore, the logic does not write to the outputs at each scan cycle, but only when their values change (e.g. the LED pin is written only every 2 s). This is typically different from the normal behaviour of a real-time PLC, as discussed in Section 3.3.2.

Given the system above, a possible implementation of Pin Control Attack can target input or output pins, having different effects. By re-configuring an output pin as input, it is possible to disable the PLC logic write operations, leading to a Denial of Service (DoS) condition. More interesting, by changing an input pin into output and writing its own values, an attacker can actually control the behaviour of the outputs. In our case, if the attacker is able to intercept PLC

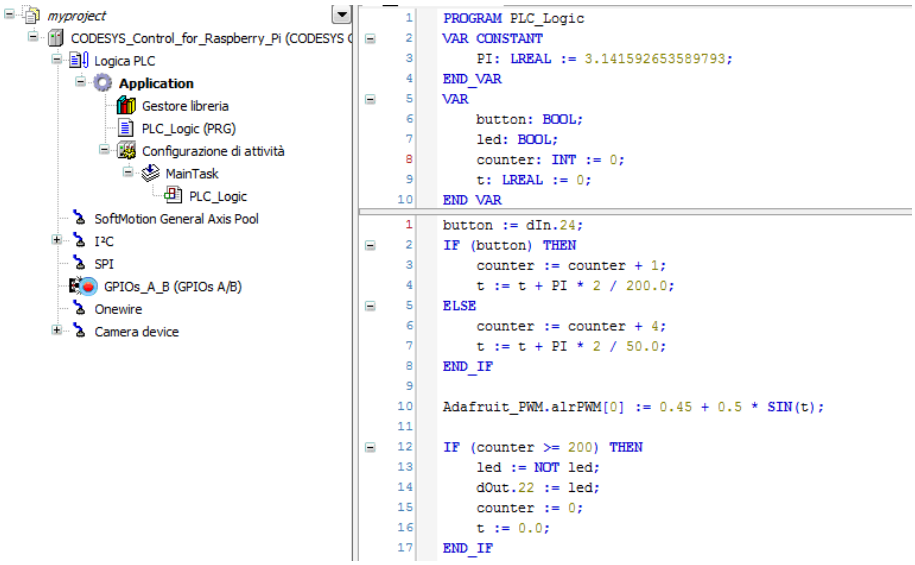


Figure 3.12: PLC logic loaded into Raspberry Pi

logic read operations and write its own values, he can actually decide the frequency of the outputs. For instance, as demonstrated by our attack experiments, if the attacker feeds the logic with 0 and 1 alternatively at each scan cycle, the round time of the outputs becomes the average between 2 s and 500 ms, factually having a frequency that is not even programmed into the logic. But there is a limitation to this attack approach, implied by the architecture described in Section 3.1.1 and confirmed by the experiments: the read operation is not reliable if the pin is configured as output pin. In our case, if we reconfigure pin 24 as output, the malicious value written into the write register is not always the same value read by the PLC logic. That is probably because the voltage on the pin is actually driven by two entities at the same time, the external sensor (the button) and attack code, so it is not deterministic whether, at the time of reading the pin level register, the voltage is considered a 1 or a 0.

The authors of Pin Control Attack described two different ways of intercepting read (or write) operations, and they implemented them into the following attack variants:

1. a Linux loadable kernel module which is able to intercept logic read and write operations by leveraging the *Debug subsystem*;
2. a user code which maps the physical I/O memory, or re-use PLC logic mapping, and manually watches the I/O values to synchronise itself with the PLC logic operations.

Once the target operation has been caught, the attack modifies the I/O configuration according to its purpose. Both implementations are able to evade the HIDSs analysed in Section 3.2.1, because they reside into kernel and user dynamic memory, respectively, and they do not alter any control flow. Here follows a brief description of these two possible implementations, focused on the findings of our experiments. For a more detailed analysis of the original version of the attacks refer to [1].

Kernel module

This variant requires root privileges, in order to be able to insert a loadable module into the kernel. The kernel module makes use of debug registers, available in many embedded architectures, to catch PLC logic operations. A debug register takes a virtual address, and optionally a process context, and raises a CPU exception when the address is accessed within the given context. The access type may be read, write or execute: the first two are intended for data addresses, the last one for instruction addresses. Thus, to use a debug register, the attacker needs to know the mapped virtual address used by the PLC logic to read/write the I/O pins. These addresses are typically

mapped by the PLC runtime before starting the logic, and can be easily obtained by looking into the `/proc/<pid>/maps` file, which lists the current mappings owned by a process identified by `pid` (process id). As discussed in Section 3.2.2 the attacker needs to know the running PLC logic, and the mapping between the I/O pins and the physical process he wants to tamper. With reference to our target system, if the attacker knows that the input button is connected to pin 24 and wants to intercept a read operation, then he can get the physical address of the read register from the SoC documentation [51] (`0x20200034`), and finally the virtual address from the current PLC runtime mappings (e.g. `0xb6f40034`). With this information, he can install a debug exception handler by setting a debug register: the malicious handler will be called whenever the PLC logic tries to read from that address. Inside the handler, the attacker can decide whether to change the pin configuration and which value wants to feed the PLC logic with. The use of debug registers gives the attacker a great timing accuracy on read and write operations, but the overhead may be detectable by power consumption-based intrusion detection systems.

User code

This version does not need root privileges as the previous one. Since it is executed from user space, to have access to the I/O physical memory it needs to either request a new mapping to the kernel or use the existing mapping of the PLC runtime. In both cases, the requirement is to have the same privilege level of the PLC runtime, and this may be obtained by exploiting a code execution vulnerability affecting the CODESYS PLC runtime [14, 15]. The requirement needed to obtain the PLC runtime mapping is the same as the kernel module variant. To intercept read and write operations, the attacker cannot use debug registers, because the access to the debug subsystem is mediated by the kernel `ptrace` API. Therefore, this version can only monitor the value of an output pin to get the relative time, and also infer timing information of other pins by using its knowledge of the PLC logic. In our case, if the attacker monitors the pin 22, when its value changes he knows from the logic that at the same time the button input has been read, and the servo motor has just finished a round. This information can be used to set relative timers, which will be activated on the next I/O events. If the target system has real-time capabilities, as almost all real PLCs do, this technique can be even more accurate than our experiments in a non real-time environment. Moreover, its overhead is much less than the corresponding debug register one, and it may not be easily detectable.

3.3.2 PLC

As done for Raspberry Pi, we analysed a real PLC and considered the attack possibilities on this system as well. The provided PLC is a *PFC200 750-8202* model from *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 [54]. As discussed in Section 3.1.2, different kind 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*, 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. 3.13.

The PLC runtime environment is the *e!RUNTIME* provided by Wago, which is always 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. 3.14. 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

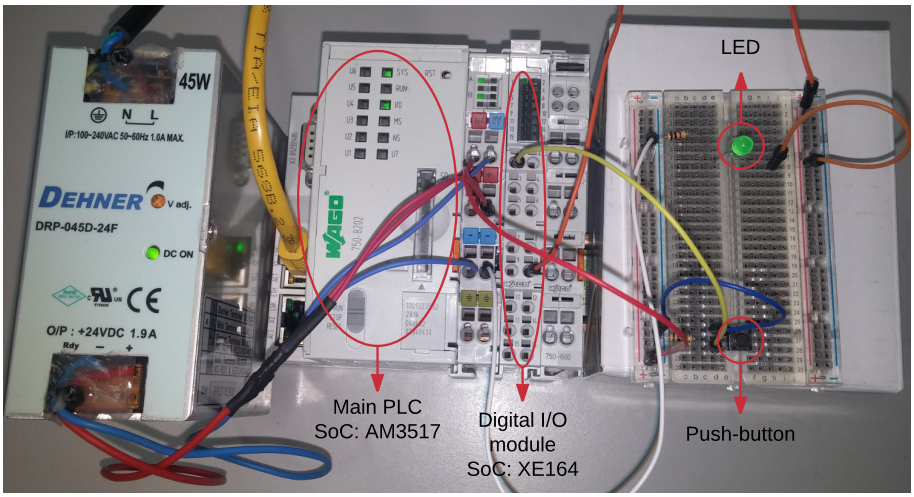


Figure 3.13: Wago PLC system used for experiments

and the I/O pins of the external module. Differently from the Raspberry Pi system behaviour, 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, already reported in Section 3.1.2, and the Wago implementation.

```
PROGRAM PLC_PRG
VAR
    counter: INT := 0;
    led: BOOL := FALSE;
    button: BOOL := TRUE;
END_VAR
```

```
IF (button) THEN
    counter := counter + 1;
    IF (counter = 50) THEN
        counter := 0;
        led := NOT led;
    END_IF
ELSE
    led := FALSE;
END_IF
```

Variable	Mapping	Channel	Address	Type	Default Value	Unit	Description
		_IN	%IB1	BYTE			Input Channels
Application.PLC_PRG.button		_IN	%IX1.0	BOOL	FALSE		Digital input
		_IN	%IX1.1	BOOL	FALSE		Digital input
		_IN	%IX1.2	BOOL	FALSE		Digital input
		_IN	%IX1.3	BOOL	FALSE		Digital input
		_IN	%IX1.4	BOOL	FALSE		Digital input
		_IN	%IX1.5	BOOL	FALSE		Digital input
		_IN	%IX1.6	BOOL	FALSE		Digital input
		_IN	%IX1.7	BOOL	FALSE		Digital input
		_OUT	%QB0	BYTE			Output Channels
		_OUT	%QX0.0	BOOL	FALSE		Digital output
Application.PLC_PRG.led		_OUT	%QX0.1	BOOL	FALSE		Digital output
		_OUT	%QX0.2	BOOL	FALSE		Digital output
		_OUT	%QX0.3	BOOL	FALSE		Digital output
		_OUT	%QX0.4	BOOL	FALSE		Digital output
		_OUT	%QX0.5	BOOL	FALSE		Digital output
		_OUT	%QX0.6	BOOL	FALSE		Digital output
		_OUT	%QX0.7	BOOL	FALSE		Digital output

Figure 3.14: PLC logic loaded into Wago PLC

PLC e!RUNTIME analysis

For our research we used the following materials and tools:

- **Board Support Package** (BSP) provided to us by Wago [54], which allows to build and configure a complete disk image of the system running inside the PLC. It essentially consists of: the kernel source code with all the patches applied by Wago, the libraries and the applications used into the system. For most of the Wago libraries, only the binary code is provided.
- **AM35x manual**: the Technical Reference Manual of AM35x processor, from [55].
- **strace** tool [56], useful to analyse the behaviour of the PLC runtime, in particular the system calls used for accessing the I/O.
- **kprobes** kernel feature [57], used to intercept function calls related to the I/O from kernel space.

We report the findings of our analysis as follows. With reference to the architecture described in Section 3.1.2, we discovered that 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 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. From strace, we found that 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. The operations performed at each scan cycle are outlined in the scheme of Fig. 3.15, which shows

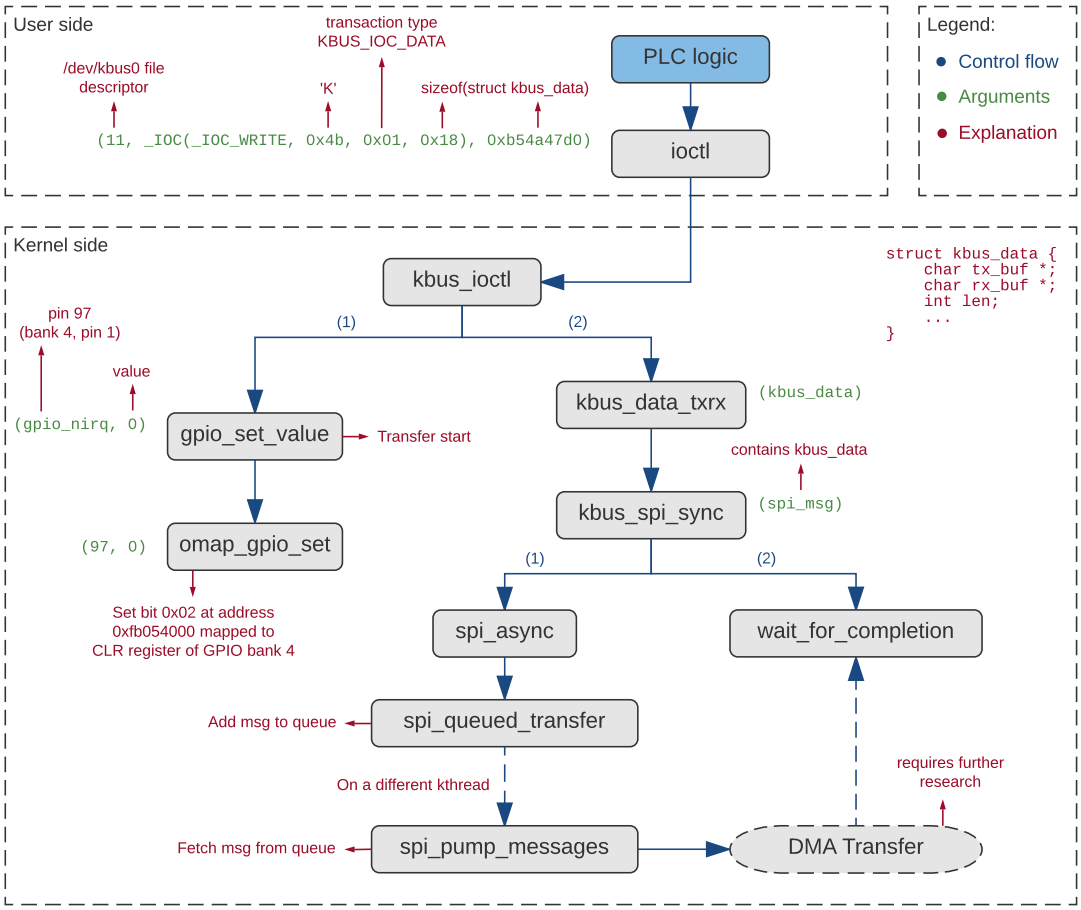


Figure 3.15: PLC I/O operations executed for each scan cycle

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 synchronisation (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 behaviour of a DMA transfer and its corresponding IRQ handler, but that goes beyond the purpose of this master thesis. 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. This behaviour gives us a hint to build another type of attack. Our findings about possible attacks are described in the next section.

PLC attack

One of the first attack attempts we made was hooking the `kbus_ioctl` function and tampering the values inside the buffer. The effect was more powerful as the one achieved by the original I/O attack, because we were able to control both inputs and outputs. Anyway, this approach required function hooking, which is forbidden by our threat model (Section 3.2.2). A second attempt was multiplexing the SPI data pins to GPIO, to prevent the SPI controller to write to its I/O pins.

This condition, unfortunately, led to an indefinite wait of e!RUNTIME, noticeable by the operator running the e!COCKPIT software. However, a better understanding of the SPI hardware protocol may allow the attacker to send fake data to the I/O module, through a technique known as *GPIO bit bang*. The bit banging technique consists of emulating in software all the operations made by a hardware controller (e.g. SPI) on I/O pins.

After having a good knowledge of the KBUS driver, we finally crafted three different attacks able to break the I/O communication without *e!RUNTIME* noticing anything and without hooking any function. The first two attacks are about pin configuration and pin multiplexing. From the previous analysis, we know that the pin 97, multiplexed as GPIO and configured as output, is used by KBUS to send the *transfer start* signal. If we either multiplex or configure the pin in a different way, we are able to prevent e!RUNTIME from reading and writing values to the I/O. This implementation demonstrates that I/O attack is perfectly applicable to real PLCs, even though it is still less powerful with respect to the original version described in Section 3.3.1. In fact, our version enables the attacker to perform only one of the operations defined for Pin Control Attack: disabling the output. Actually, since read and write are performed within the context of the same SPI transfer, also the input gets disabled at the same time. However, this is irrelevant because the PLC logic cannot write to output in any case, independently from the input. As discussed before, whether this operation may allow the attacker to alter the physical process, or not, is highly variable case by case. For instance, in our target system, we proved that by repeating the attack every 50 ms with a trivial script, we were able to modify the logic behaviour: the LED output started blinking every 50 ms instead of the original 500 ms. Furthermore, we found another interesting attack vector, which leverages the interrupt line used by the DMA controller and the `write` interface provided by the KBUS driver. Our experiments show that if this interrupt is masked while the PLC logic is running, the system is not able to read/write the I/O, thus obtaining again the same effect as the previous implementation. Although it is still very simple, this kind of attack, which we can call *IRQ Configuration Attack*, has some similarities with Pin Control Attack, such as the stealthiness and the hardware-level target. The PLC runtime is not able to notice our attacks, no failures or errors are reported and no function hooks are required. For both attack versions, the CPU overhead depends on the actual implementation, since the attacker can decide at which frequency disabling/enabling the I/O. Generally speaking, the overhead is very low, because only one operation is needed to do/undo the attack, respectively, which is typically translated into two or three processor instructions.

As discussed above, we believe that more sophisticated attacks are possible, since the basic hardware concepts described in Section 3.1 are still valid for a real PLC. The only difference is that, in this particular case, I/O pins are used as part of more complex protocols (SPI, DMA, etc.) and they are multiplexed to the corresponding controllers inside the SoC. Therefore, to understand how each pin is used and to obtain more elaborated attacks (e.g. faking data via GPIO bit bang), further research effort is required.

To conclude our PLC analysis, we briefly describe the implementation of our attacks on Wago PLC. Pin configuration and pin multiplexing attacks are basically the same as for Raspberry Pi system. They can be executed in both kernel and user variants, requiring the corresponding privilege level as discussed in Section 3.3.1. One important difference here is that e!RUNTIME does not have its own mapped virtual addresses in user space, because the I/O is managed in kernel space by the KBUS driver. Thus, we distinguish two cases:

- **kernel-side attack**: the attacker can either remap the I/O physical addresses taken from the AM3517 manual [55], or re-use the addresses already mapped by the pin control subsystem;
- **user-side attack**: the attacker can only request a new mapping to the kernel, because no other available mappings exist in user space.

Re-using already mapped addresses is theoretically feasible, because the control registers are actually mapped into the kernel, as reported by the output below.

```
root@PFC200-4106BA:~ cat /proc/iomem
[...]
```

```
48002030-48002267 : pinctrl-single
48002a00-48002a5b : pinctrl-single
[...]
```

However, the attacker needs to know the kernel virtual addresses (e.g. by looking at page tables) in order to have access to the mapped registers.

Finally, for our IRQ configuration attack, we implemented it in two different variants as well: kernel side and user side.

Kernel side This version requires root privileges, since it is designed as a loadable kernel module. The module leverages the `disable_irq` function provided by the Linux kernel, which requires the interrupt line number to disable. Thus, the attacker needs to know the target system model, but no knowledge of the PLC logic nor of the I/O mapping is required, because the system uses the same configuration independently from the current logic. We found the interrupt line number by inserting a kernel probe for the `disable_irq` function, and triggering the PLC runtime to disable the interrupts while loading a new logic.

User side This version requires at least the same privilege level of the PLC runtime, who is able to enable/disable the interrupt with the following operations (from strace):

```
[...]
04:09:46.008222 open("/dev/kbus0", O_RDWR) = 11
[...]
04:09:46.009237 write(11, "\x01\x00\x00\x00", 4) = 0
[...]
04:09:46.070565 write(11, "\x00\x00\x00\x00", 4) = 0
[...]
```

Writing a 1 into the `/dev/kbus0` file disables the interrupt, while a 0 enables it. We can implement the attack by emulating these PLC runtime operations from user space. The effect of this second variant is the same as the kernel module one, but it requires less privilege and it could be exploited through a remote code execution vulnerability of the PLC runtime, which is again based on CODESYS [14, 15].

Chapter 4

Attack detection

This chapter proposes a possible countermeasure to I/O level attacks. First, it describes the overall design of our detection system, showing the approaches to tackle the attack at different levels and explaining our design choices. Second, it contains a detailed report of the implementation for Linux kernel.

4.1 Design

We report the design phase below, starting from a preliminary analysis of the problem to a discussion of different possible solutions. Then, a detailed description of the defense architecture concludes the section.

4.1.1 Preliminary analysis

The main goal of our defense is to detect and hinder I/O attacks while being able to not consume the limited resources inside the target system, and to not affect real-time operations. The timing of the operations is fundamental in systems like PLCs, on which the minimum delay may alter the physical process.

We make the same assumptions contained in the threat model of Section 3.2.2. Therefore, we assume that the system, and our defense as well, is protected by the previously discussed HIDSs. Starting from the analysis reported in Chapter 3, 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 paper:

- **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 unauthorised modification may have direct or indirect effects on the controlled process. In particular, we are interested into pin control registers, which directly determine the behaviour 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 described in Section 3.1, protecting I/O configuration is not straightforward because the hardware lacks any protection mechanism related to the I/O subsystem. For instance, the SoC might generate a trap for each modification to I/O registers. However, this approach would not be reasonable, because of the huge overhead imposed to any I/O access. Alternatively, a trap could be generated only when an I/O access is malicious. For example, to prevent pin configuration attack, a trap signal might be delivered to the CPU when the internal signal of a pin is different from the external one, meaning that the pin is misconfigured. A similar approach may be used for pin multiplexing, by checking the internal signal connected to the multiplexed device. Unfortunately, since these approaches would require significant hardware modifications, with subsequent updates of all the SoC drivers, they are very unlikely to be applied.

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 analysed the possible software-based solutions. In order to choose the best strategy against Pin Control Attack, we compared the following approaches, as proposed in [1]:

1. monitoring I/O configuration to detect changes;
2. monitoring the use of debug registers;
3. monitoring the mapping requests targeting I/O configuration;
4. monitoring performance overhead;
5. using a trusted execution environment.

These approaches are not mutually exclusive, and may be used simultaneously to get a higher protection level. In this work, we decided to design and implement the first three approaches in combination, because they directly protect the three different resources that the attacker may use. Thus, they represent a good minimum set of countermeasures capable of raising the bar for the attacker (see Section 4.1.2). Our solution, however, may be extended to include the last two approaches as well, to cover some existing limitations (see Section 4.3). Before describing our design phase, we briefly discuss these two further protections in the following sections.

Monitoring performance overhead

A detection system based on performance monitoring may be useful to add a further protection, especially against the attack variant which uses debug registers. In general, such a monitor would be also useful for other kind of attacks targeting PLCs, because these systems cyclically performs a limited amount of well-known operations. This simplifies the detection of any deviation from the standard behaviour. In our case, a performance monitor would be helpful to cover those attacks that the first three strategies are not able to detect, although, as discussed in the following sections, those are very limited cases. Deploying a performance monitoring system, anyway, does not require very high effort because it can leverage *Hardware Performance Counters* (HPC), nowadays available in almost all the SoC processors. The most challenging part would be, of course, integrating

the monitor with the PLC runtime. When a new logic is uploaded to the PLC, a burst of operations are executed by the runtime to check the new logic code, apply the new configuration and start the logic. These extra operations, of course, must not be labeled as malicious. Furthermore, once a new logic started, it may perform operations that are different with respect to the previous one; therefore, the monitor should be able to recognise this change and update its statistics as well.

Trusted execution environment

A Trusted Execution Environment (TEE) is a particular set of hardware and software components providing security features, such as isolated execution, integrity of Trusted Applications (TAs), and integrity and confidentiality of TAs assets [58]. The TEE technology is based on the concept of partitioning a computing system into Secure and non-Secure world, where the code running into non-Secure world cannot access the Secure partition. A lot of effort has been put into the standardisation of the TEE, and some commercial solutions already exist, such as Intel Trusted Execution Technology (TXT) [59] and ARM TrustZone [28]. Since embedded systems are our main target, we provide a brief description focused on the ARM TrustZone implementation. In this technology, the physical memory is partitioned into Secure and non-Secure regions, and the processor core is divided into Secure and Non-secure virtual cores. The virtual core is distinguished by the NSTID (Non-Secure Table IDentifier) bit associated with the current instruction, while each instruction or data address (i.e. each bus transaction) is marked with an NS bit. The protection of the Secure world is guaranteed by checking all the accesses to memory or peripherals. The Non-secure core can only access Non-secure memory regions, while the Secure world can use both Secure and Non-secure addresses. This partitioning is parallel and independent from Supervisor/User modes available on the CPU. Therefore, each world has its own supervisor and user mode as well. To improve the performance of this architecture, TLBs and caches may support the NS attribute for each entry as well. This enables Secure and Non-secure entries to co-exist avoiding TLB and cache flushes on every switch between the two worlds. The switching between Secure and Non-secure world is managed through a specific Secure Monitor Call (SMC) instruction, which changes the core mode into Monitor Mode. The code executed into monitor mode is always Secure, and it basically performs the context switch between the two virtual cores. If a Secure process is loaded, NSTID bit is set accordingly. Since the Secure world may contain a whole parallel micro-kernel with trusted user applications, it may be possible to deploy our detection system inside the trusted domain, protecting the defense itself from defense-aware attackers. However, the overhead imposed by a trusted execution environment may be unacceptable for embedded systems with real-time constraints like PLCs. Hence, the impact of this solution still needs to be investigated.

4.1.2 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 can refer to our detection system as *Pin Control Defense*. The system is designed to run as part of the operating system, thus having kernel privilege level. Its overall architecture is shown in Fig. 4.1. 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.

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

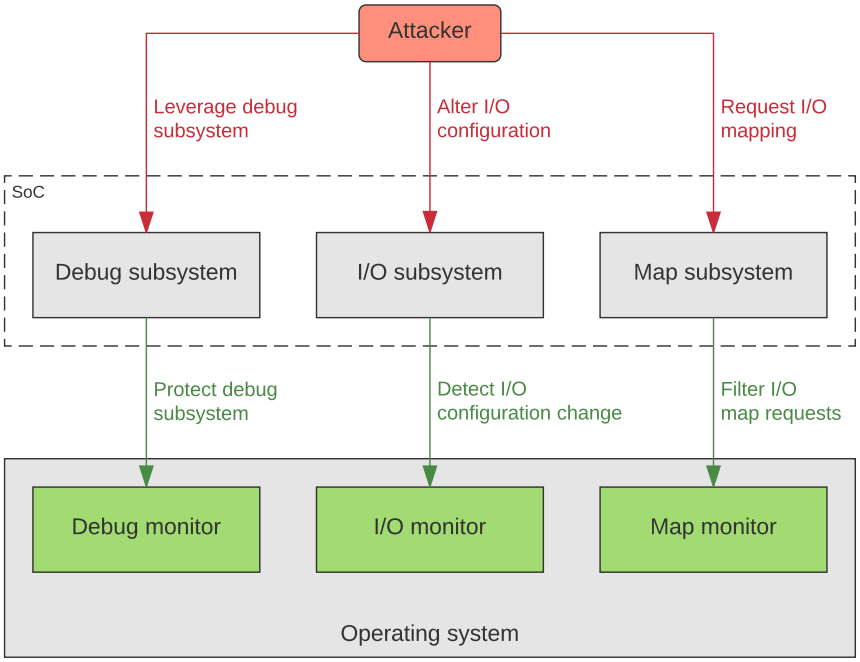


Figure 4.1: Pin Control Defense general architecture

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 customisable, the actual effects depends on their configuration. The following sections describe in more detail the design and the role of each module.

4.1.3 I/O monitor

The I/O monitor plays the most critical role into our detection system. Its 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 behaviour. As discussed in Section 3.1.2, 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 behavioural 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 behaviour 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 behavioural 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 behavioural 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 analyse 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 analysing the PLC software it is possible to dynamically obtain the required information from the running logic;
- **PLC runtime vendor collaboration**: if the PLC software 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 (see our approach in Section 4.2.2). 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.

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 synchronise itself with the monitor loop, it has enough time to alter $\left\lfloor \frac{50\,\text{ms}}{10\,\text{ms}} \right\rfloor = 5$ consecutive I/O operations and then restore the configuration back before getting noticed. Smaller intervals, instead, may cause too much performance overhead. We provide our experimental results in Chapter 5.

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.

4.1.4 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. To understand this problem, we need to distinguish the following two cases: the debug support can be either enabled or disabled into the kernel itself. If it is enabled, the attacker may simply leverage the system interface to use debug registers from kernel space. If debug support is disabled, an attacker who gains kernel level access (as in kernel module version of Pin Control Attack) can always re-enable them at run-time by inserting its own debug exception handler into the OS interrupt vector table (see [60,61]). However, this kind of attack is already covered by the defenses assumed in our threat model, because it is a data hooking technique. Thus, we designed the following monitor strategy, that is required only if debug support is enabled into the operating system:

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 I/O 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 behavioural 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.

4.1.5 Mapping monitor

As discussed in Section 4.1.1, 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. We distinguish at least two techniques to re-use existing virtual addresses:

- exploit a remote code execution vulnerability on the PLC runtime owning the addresses;
- set debug register on PLC runtime virtual address (the debug handler will have access to the process virtual addresses).

We exclude the second technique, because it is already detectable by the debug monitor. The first one, instead, can only be counteracted by detecting the control flow attack itself. Thus, the aim of this monitor is not to detect addresses re-using, which is out of our scope, but 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 summarise the strategy of the map monitor as follows:

1. define a trusted behavioural 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 behaviour 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 behavioural model of step 1 should describe if and how the mapping interface is used by the PLC runtime. We analysed the behaviour 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 (see implementation in Section 4.2 for more details):

- 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 Section 3.2.2, 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 behaviour. 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.

4.2 Implementation

We describe here our implementation of the above strategies, discussing the engineering problems encountered and the adopted solutions. Since the authors of the attack presented their work as "Ghost in the PLC" [1], we called our defense prototype implementation *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 be easily extendable 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.

4.2.1 Implementation architecture

The implementation architecture is based on the general one described in Section 4.1.2. Thus, we implemented I/O, debug and map monitors. Each one of them has been divided into two main parts, following the template method pattern: the main strategy and the sub-actions implementation. The aim of this separation is to minimise the effort needed to deploy our defense into different systems, thus improving its portability. In particular, we considered the following variables: each target system may have its own System on Chip, firmware and PLC runtime. During the design part, we defined high-level monitor strategies, which allowed us to provide an abstraction adaptable to any Linux-based system, independently from these variables. The resulting architecture is shown in Fig. 4.2, which associates each specific role to the corresponding source file(s) (in red). Furthermore, the compilation of our module can be parameterised in different ways, as shown by the green labels in the figure. Each green label, associated to a rectangle container in the figure, corresponds to a compiler flag that affects the compilation of the enclosed portion. The compilation is managed by a `Makefile`, in which all the required flags are defined. The `Makefile` needs to know the location of the target Linux kernel source directory.

In particular, each monitor can be enabled or disabled by means of the following flags:

```
IO_MONITOR_ENABLED
DR_MONITOR_ENABLED
MAP_MONITOR_ENABLED
```

If a monitor is disabled, it will not be included into the compilation at all, reducing the final binary size. This may be useful to exclude a particular monitor that is not needed by the target system (e.g. DR monitor if debug support is not enabled into the kernel, see discussion in Section 4.1.4).

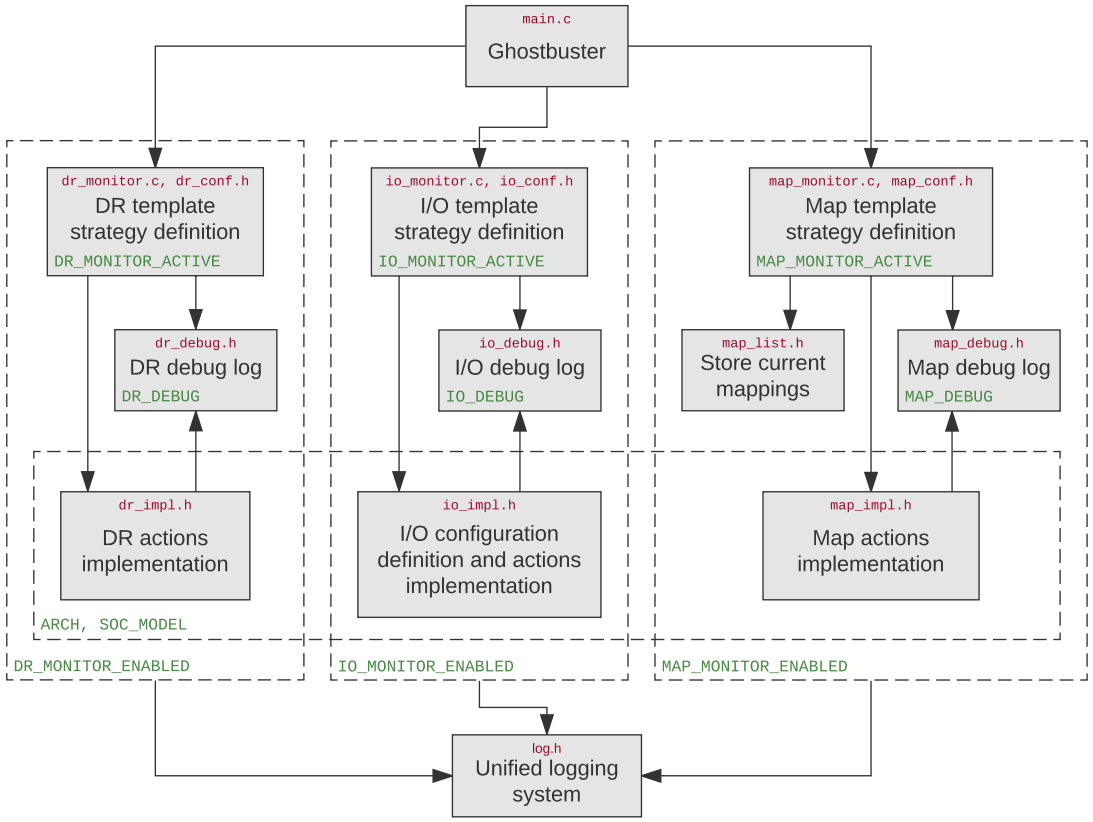


Figure 4.2: Ghostbuster implementation architecture

When a monitor is enabled, its execution mode may be controlled by one of the following flags, respectively:

```
IO_MONITOR_ACTIVE
DR_MONITOR_ACTIVE
MAP_MONITOR_ACTIVE
```

If one of these flags is not defined, the corresponding monitor is compiled in *passive* mode: it will only reports about events, without taking any further action. Otherwise, the monitor is *active* and it will include its specific reactions to the events. If a monitor is disabled, the corresponding active flag is ignored.

The messages reported by each monitor may be extended by enabling the following debug flags:

```
IO_DEBUG
DR_DEBUG
MAP_DEBUG
```

If a debug flag is defined, the corresponding monitor will print out its complete state at each event detection.

Finally, the `ARCH` and `SOC_MODEL` variables are used to select a specific implementation for the enabled monitors. An implementation is identified by an architecture name and a SoC model (e.g. `arm`, `BCM2835`). Given this information, the implementation files will be automatically included from the `<ARCH>/<SOC_MODEL>/` sub-directory. We chose to directly include the implementation part into header files, instead of external compilation units (`.c` files), to allow the usage of `inline` functions. The code of an inline function is directly included into the caller, and it does not need

an explicit function call. This may cause some code duplication, which we tried to minimise, but it achieves better performance, especially for those functions that are called many times (e.g. in the main loop of a monitor).

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 (see Section 4.2.5 for more information). The target systems used to test our defense are exactly the same as the ones described for the attack part in Chapter 3, 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*.

4.2.2 I/O monitor

This monitor is responsible for protecting I/O configuration memory from malicious usage. The monitor interface includes the I/O configuration data type, modeled as a set of memory blocks by means of the `io_conf_t` structure:

```
typedef struct {
        const void** addrs; // Set of block base addresses
        const unsigned* sizes; // Size of each block in bytes
        const unsigned blocks; // Number of blocks
        const unsigned size; // Total size in byte
} io_conf_t;
```

This model takes into account the fact that I/O registers may be located at different addresses, resulting in a non-contiguous I/O memory.

With reference to the BCM2835 manual [51] and to the target configuration defined in Section 3.3.1, 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, by filling in the `io_conf_t` global structure:

```
#define IO_BLOCKS               1 // Registers are contiguous: one block needed
#define PIN_CTRL_BASE           ((void*)0x20200000) // Pin control base address
#define PIN_CTRL_SIZE           24 // 6 regs * 4 bytes each
#define __IO_STATE_TOTAL_SIZE   24 // Total size in bytes

static const void* bcm2835_io_addrs[IO_BLOCKS] = { PIN_CTRL_BASE };
static const unsigned bcm2835_io_sizes[IO_BLOCKS] = { PIN_CTRL_SIZE };

static const io_conf_t phys_io_conf = {
        .addrs = bcm2835_io_addrs,
        .sizes = bcm2835_io_sizes,
        .blocks = IO_BLOCKS,
        .size = __IO_STATE_TOTAL_SIZE
};
```

Of course our implementation is only a prototype, and many other registers may be included into the I/O configuration (e.g. event detect, edge detect, I2C control registers, etc.).

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

- **pinmux registers**: they are initialised at boot time, and never change during run-time;
- **pinconf registers**: they are initialised 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. Both codes are shown in Algorithm 1, and they are independent from any specific architecture or SoC model.

Algorithm 1 I/O monitor interface: main loop and detection handler

```
1:  function IOMainLoop()
2:      t ← monitor scan interval                          ▷ The monitor interval in ms
3:      C ← physical I/O configuration                     ▷ The defined I/O configuration
4:      T ← GetIOState(C)                                  ▷ Read trusted state from I/O registers
5:      loop
6:          for each block B ∈ C having index i do
7:              CheckIOState(B, T[i])                      ▷ Compare current and trusted state of block B
8:          end for
9:          if monitor should stop then
10:             return                                     ▷ Return if Ghostbuster is being stopped
11:         end if
12:         msleep(t)                                      ▷ Wait for next cycle
13:     end loop
14: end function

1:  function HandleIODetection(D)                          ▷ React to the detection D
2:      report about D
3:      if I/O debug enabled then
4:          dump entire I/O state
5:      end if
6:      if IsLegitimate(D) then
7:          UpdateIOState(D)                               ▷ Accept new configuration, update trusted state
8:      else
9:          report about I/O Attack
10:         if I/O monitor is active then
11:             RestoreIOState(D)                          ▷ Reject new configuration, restore trusted state back
12:         end if
13:     end if
14: end function
```

Each iteration of the main loop is executed every t ms, where the specific value of t is defined into the `IO_MONITOR_INTERVAL` flag. The loop terminates only if an external signal indicates that Ghostbuster is being stopped (see Section 4.2.5). The verification of the current I/O configuration is performed at block level against the golden reference, which is obtained by GetIOState 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 trusted. When a modification is detected by CheckIOState, the detection handler is called back and the monitor strategy is applied. Note that the handler may be called many times from the same call to CheckIOState, 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. These functions deal with the effective accesses to I/O registers, which depend on many factors. The implementation knows the way to access these registers, their size and their bits arrangement. In Algorithm 2 we show our implementation related to the BCM2835

Algorithm 2 I/O monitor implementation functions

```
1: function GetIOState(C)                                   ▷ Read current I/O state
2:     T ← ∅
3:     for each block B in C do
4:         for each register R ∈ B do
5:             T ← T ∪ ioread32(R)          ▷ Save value of register R into the trusted state
6:         end for
7:     end for
8:     return T
9: end function

1: function CheckIOState(B, T_B)          ▷ Verify block B against its trusted state T_B
2:     for each register R ∈ B having index i do
3:         C_R ← ioread32(R)                          ▷ Read current value of register R
4:         T_R ← T_B[i]                                 ▷ Trusted value of register R
5:         for each pin p ∈ R having index j do
6:             d ← C_R[j] ⊕ T_R[j]          ▷ Difference between current and trusted value
7:             if d ≠ 0 then
8:                 D ← {C_R, T_R, j, d}                 ▷ Fill in detection information
9:                 HandleIODetection(D)                    ▷ Call detection handler
10:            end if
11:        end for
12:    end for
13: end function

1: function IsLegitimate(D)
2:     if D is pin multiplexing then
3:         return false                  ▷ Pin multiplexing is not allowed at run-time
4:     end if
5:     if D is pin configuration then
6:         if new I/O configuration is conforming to PLC logic then
7:             return true                ▷ Change performed by the PLC runtime
8:         else
9:             return false                                          ▷ Malicious change
10:        end if
11:    end if
12: end function

1: function UpdateIOState(D)
2:     D.T_R[j] = D.C_R[j]   ▷ Update the trusted value according to new I/O configuration
3: end function

1: function RestoreIOState(D)
2:     iowrite32(D.T_R ⊕ D.d)          ▷ Restore I/O configuration to the trusted value
3: end function
```

SoC, where all registers are 32-bits wide.

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 the IsLegitimate function, which decides whether a configuration change

is legal or not. Based on our behavioural 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 Section 4.1.3, reported below.

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 [51]. 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 Section 4.2.3). 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. See Chapter 5 for more details about our results.

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 with the function listed in Algorithm 3, which accepts a pair of physical addresses $[s, e]$ representing the start and the end addresses of the given range. Since this function is needed by the MAP monitor independently

Algorithm 3 I/O monitor overlap function

```
1: function MapOverlapsIO(s, e)   ▷ Check if physical range [s, e] overlaps I/O configuration
2:     C ← physical I/O configuration                          ▷ The defined I/O configuration
3:     for each block B in C do
4:         if (B.s ≤ s ≤ B.e) ∨ (B.s ≤ e ≤ B.e) then          ▷ Overlap condition at block level
5:             return true
6:         end if
7:     end for
8:     return false
9: end function
```

from I/O monitor operations, we designed it to be available even if the I/O monitor is disabled. The I/O configuration, needed by MapOverlapsIO, is made available independently from the I/O monitor as well. 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.

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 behavioural 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 behavioural models which need to be analysed.
- **Debug subsystem**: given the engineering problem described later in Section 4.2.3, 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.

4.2.3 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 [62]:

```
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 these functions, we dynamically patched the kernel text replacing the prologue instructions of each function. In particular, they have the following prologue:

```
c00d3cb0 <register_user_hw_breakpoint>:
c00d3cb0: e1a0c00d  mov  ip, sp
c00d3cb4: e92dd800  push {fp, ip, lr, pc}
c00d3cb8: e24cb004  sub  fp, ip, #4
c00d3cbc: e24dd008  sub  sp, sp, #8
[...]

c00d3e30 <modify_user_hw_breakpoint>:
c00d3e30: e1a0c00d  mov  ip, sp
c00d3e34: e92ddff0  push {r4, r5, r6, r7, r8, r9, sl, fp, ip, lr, pc}
c00d3e38: e24cb004  sub  fp, ip, #4
c00d3e3c: e24dd00c  sub  sp, sp, #12
[...]

c00d3ce8 <unregister_hw_breakpoint>:
c00d3ce8: e1a0c00d  mov   ip, sp
c00d3cec: e92dd800  push  {fp, ip, lr, pc}
c00d3cf0: e24cb004  sub   fp, ip, #4
[...]
```

We replaced the prologue instructions with the following ones:

1. **return value**: if the function is not void (e.g. in the first two cases), we need to return a value to the caller. Thus, we move the value `-EACCES` (-13) into R0 by using the move-negate instruction: `mvn r0, #0xC`. `EACCES` is defined into the `errno-base.h` kernel header as `Permission denied`;
2. **branch**: jump back to the caller by branching to the link register: `bx lr`.

After the patch, the kernel text becomes the following:

```
c00d3cb0 <register_user_hw_breakpoint>:
c00d3cb0: e3e0000c  mvn  r0, #0xC
c00d3cb4: e12fff1e  bx   lr
[...]

c00d3e30 <modify_user_hw_breakpoint>:
c00d3e30: e3e0000c  mvn  r0, #0xC
c00d3e34: e12fff1e  bx   lr
[...]

c00d3ce8 <unregister_hw_breakpoint>:
c00d3ce8: e12fff1e  bx   lr
[...]
```

Once the user access to debug registers has been disabled, we activate the DR monitor to protect them from attackers who gain kernel access. As for the I/O monitor, we report the interface part and the implementation. The abstract strategy of the DR monitor is rather simple, and is listed in Algorithm 4. It is a simplification of the I/O monitor strategy, where any modification is considered

Algorithm 4 DR monitor interface: main loop and detection handler

```
1: function DRMainLoop()
2:     m_T ← mutex                                  ▷ Global mutex to protect trusted state
3:     t ← monitor scan interval                    ▷ The monitor interval in ms
4:     T ← GetDRState()                             ▷ Read trusted state from debug registers
5:     loop
6:         lock(m_T)
7:         CheckDRState(T)                          ▷ Compare current and trusted state of debug registers
8:         unlock(m_T)
9:         if monitor should stop then
10:            return                               ▷ Return if Ghostbuster is being stopped
11:        end if
12:        msleep(t)                                ▷ Wait for next cycle
13:    end loop
14: end function

1: function HandleDRDetection(D)                    ▷ React to the detection D
2:     if DR debug enabled then
3:         dump entire DR state
4:     end if
5:     report about I/O Attack
6:     if DR monitor is active then
7:         RestoreDRState(D)                        ▷ Restore debug register trusted state
8:     end if
9: end function
```

malicious. Since in our implementation the I/O monitor makes use of the debug subsystem, the DR monitor exposes the interface listed in Algorithm 5 to mediate any access to debug registers. By

Algorithm 5 DR monitor interface for I/O monitor

```
1: function SetDR(r)                                ▷ Set a debug register r
2:     lock(m_T)
3:     register_wide_hw_breakpoint(r)               ▷ Use kernel interface for debug registers
4:     T ← GetDRState()                             ▷ Update trusted state
5:     unlock(m_T)
6: end function

1: function ResetDR(r)                              ▷ Reset a previously set debug register r
2:     lock(m_T)
3:     unregister_wide_hw_breakpoint(r)             ▷ Use kernel interface for debug registers
4:     T ← GetDRState()                             ▷ Update trusted state
5:     unlock(m_T)
6: end function
```

using this interface, any modification of debug registers performed by the I/O monitor is excluded from the detection. This mechanism requires that the DR state is protected from concurrent access by a mutual exclusion mechanism.

All the details about debug registers are handled by the implementation part, shown in Algorithm 6. 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

Algorithm 6 DR monitor implementation functions

```
1: function GetDRState()                                   ▷ Read current I/O state
2:     T ← ∅
3:     D ← hardwaredebugregisters
4:     for each debug register r ∈ D do
5:         T ← T ∪ DRread(r)                    ▷ Save value of debug register r into T
6:     end for
7:     return T
8: end function
```

```
1: function CheckDRState(T)          ▷ Verify debug registers against trusted state T_B
2:     for each debug register r ∈ B having index i do
3:         C_r ← DRread(r)                              ▷ Read current value of register r
4:         T_r ← T[i]                                       ▷ Trusted value of register r
5:         d ← C_r ⊕ T_r                  ▷ Difference between current and trusted value
6:         if d ≠ 0 then
7:             D ← {C_r, T_r, r}                        ▷ Fill in detection information
8:             HandleDRDetection(D)                          ▷ Call detection handler
9:         end if
10:    end for
11: end function
```

```
1: function RestoreDRState(D)
2:     DRwrite(D.r, D.T_r)          ▷ Restore debug register to the trusted value
3: end function
```

them, and the detection is handled at register level. Each register is accessed through specific coprocessor 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.

Wago PLC version

To adapt the DR monitor for Wago PLC, the following engineering problem needs to be solved. 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 coprocessor instructions [55]. 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 technical problem, 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.

4.2.4 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. 4.3 summarises a significant sequence of operations performed via the mapping interface, assuming physical and virtual address spaces based on 4 kB pages. 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 (a). Second, he may use `mremap` to extend a current mapping, possibly causing overlap on an I/O address which was not mapped before (b). Third, he can re-map an already mapped virtual address to a different physical address, including a protected I/O address as well (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) (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. The definition of our data structure, which uses the list defined into the `<list.h>` kernel header, is reported below:

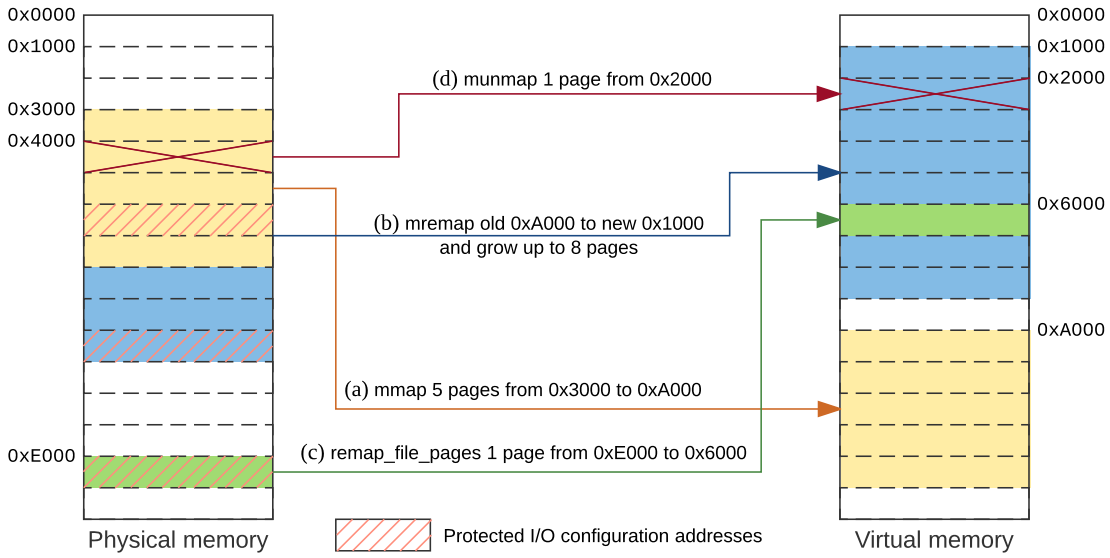


Figure 4.3: Accessing physical addresses through Linux mapping interface

```
typedef struct {
        struct list_head pages; // Pointer to the rest of the list
        unsigned long paddr; // The page physical address
        unsigned long vaddr; // The page virtual address
        pid_t pid; // The process who requested the mapping
} page;
LIST_HEAD(page_list); // The list head
DEFINE_MUTEX(page_list_lock); // To protect the list from concurrent accesses
```

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 made of 4 hook functions having the same prototype of the system calls described above. 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. Our approach is feasible because does not use slow mechanisms such as kprobes [57], but it directly replaces the pointers into the system call table. Note that this mechanism is just a prototype, and it has some minor drawbacks that could be reduced. For instance, since our functions must be consistent with the original system calls, some code needs to be duplicated. In particular, we need to check the arguments used by the monitor in the same way the original system calls do (e.g. for page alignment). This could be avoided by including our monitor as part of the kernel code itself. Note that this improvement may be worth considering that the mapping system calls might be called many times during run-time, since they are not only used for physical memory but for any file/device mapping.

The abstract part of the MAP monitor, shown in Algorithm 7, is independent from the specific architecture. It makes use of the AddMapping, GetMappingPhysAddr, UpdateMapping, AlterMapping and DeleteMapping functions provided by the page list structure. We omit their implementation because it is straightforward and does not add anything to our discussion. PageAlign (and PageMask), PageShift and error, instead, refer to corresponding Linux kernel macros that deal with page alignment, page offset and error handling, respectively. For system calls that do not have the file descriptor as parameter, to check if a request refers to a physical memory mapping, and not to other files/devices, we look into our page list. If the given virtual address `addr` is not mapped into the list by the current `pid`, i.e. it does not map a valid

Algorithm 7 MAP monitor interface: hook functions for mapping system calls

```
1: function MyMmap(addr, len, prot, flags, fd, pgoff)                ▷ mmap (mmap2) hook
2:     pid ← current pid
3:     len ← PageAlign(len)                                  ▷ Align len to next page boundary
4:     if IsPhysMem(fd) then                                 ▷ If it is a physical memory mapping
5:         s ← PageShift(pgoff)                                        ▷ Start physical address
6:         e ← s + len                                                 ▷ End physical address
7:         if MapOverlapsIO(s, e) then                                 ▷ Use I/O monitor interface
8:             report about overlap
9:             if monitor is active then
10:                return −13                                          ▷ Permission denied
11:            end if
12:        end if
13:        r ← mmap2(addr, len, prot, flags, fd, pgoff)                ▷ Original system call
14:        if ¬error(r) then                                           ▷ Call succeeded
15:            AddMapping(s, e, r, pid)                ▷ Add mapped pages to the page list
16:        end if
17:        return r
18:    end if
19:    return mmap2(addr, len, prot, flags, fd, pgoff)                 ▷ Original system call
20: end function

1: function MyMremap(addr, oldLen, newLen, flags, newAddr)             ▷ mremap hook
2:     if addr is not page aligned then          ▷ mremap requires a page aligned address
3:         return mremap(addr, oldLen, newLen, flags, newAddr)         ▷ It will fail
4:     end if
5:     pid ← current pid
6:     oldLen ← PageAlign(oldLen)                ▷ Align old length to next page boundary
7:     newLen ← PageAlign(newLen)                ▷ Align new length to next page boundary
8:     s ← GetMappingPhysAddr(addr, pid)         ▷ Search page list for start physical address
9:     if ¬error(s) then                         ▷ If it is a physical memory mapping
10:        if newLen > oldLen then                                     ▷ Growing is dangerous
11:            e ← s + newLen                                          ▷ End physical address
12:            if MapOverlapsIO(s, e) then                             ▷ Use I/O monitor interface
13:                report about overlap
14:                if monitor is active then
15:                    return −13                                      ▷ Permission denied
16:                end if
17:            end if
18:        end if
19:        r ← mremap(addr, oldLen, newLen, flags, newAddr)            ▷ Original system call
20:        if ¬error(r) then                                           ▷ Call succeeded
21:            UpdateMapping(addr, oldLen, r, newLen, s, pid)  ▷ Update pages in page list
22:        end if
23:        return r
24:    end if
25:    return mremap(addr, oldLen, newLen, flags, newAddr)             ▷ Original system call
26: end function
```

physical address, then two possibilities exist. Either the virtual address is mapped to some other device (not included into the list by our mmap hook), or the process is sending a bad virtual address. In both cases we are not interested to the request, and we can let the original system call manage it.

The implementation part of the MAP monitor deals with the actual hooking procedure, which includes the following operations:

```
1: function MyRemapFilePages(addr, len, prot, pgoff, flags)        ▷ remap_file_pages hook
2:     addr ← PageMask(addr)                                  ▷ Align addr to previous page boundary
3:     len ← PageAlign(len)                                   ▷ Align len to next page boundary
4:     s ← GetMappingPhysAddr(addr, pid)                      ▷ Search page list for start physical address
5:     if ¬error(s) then                                      ▷ If it is a physical memory mapping
6:         s ← PageShift(pgoff)                               ▷ Start physical address
7:         e ← s + len                                        ▷ End physical address
8:         if MapOverlapsIO(s, e) then                        ▷ Use I/O monitor interface
9:             report about overlap
10:            if monitor is active then
11:                return −13                                 ▷ Permission denied
12:            end if
13:        end if
14:        r ← remap_file_pages(addr, len, prot, pgoff, flags)   ▷ Original system call
15:        if r == 0 then                                     ▷ Call succeeded
16:            AlterMapping(addr, s, len, pid)                ▷ Update pages in page list
17:        end if
18:        return r
19:    end if
20:    return remap_file_pages(addr, len, prot, pgoff, flags)   ▷ Original system call
21: end function
```

```
1: function MyMunmap(addr, len)                                ▷ munmap hook
2:     if addr is not page aligned then                        ▷ munmap requires a page aligned address
3:         return munmap(addr, len)                            ▷ It will fail
4:     end if
5:     s ← GetMappingPhysAddr(addr, pid)                       ▷ Search page list for start physical address
6:     if ¬error(s) then                                       ▷ If it is a physical memory mapping
7:         report about s being unmapped
8:         DeleteMapping(addr, len, pid)                       ▷ Delete pages from page list
9:     end if
10:    return munmap(addr, len)                                ▷ Original system call
11: end function
```

1. finding the system call table base address in kernel memory;
2. storing a copy of the original system call pointers to allow the abstract part to call them when necessary;
3. replacing them with pointers to the hook functions defined above.

Several technical details of these operations, which we omit in this report, depend on the actual architecture (e.g. find system call table, page write attribute, size of pointers, system call convention, etc.). The interesting part of the implementation is the function responsible for identifying whether a map request refers to physical memory or not. This is also dependent from the particular target system. For instance, the code listed below is related to the IsPhysMem implementation for our Raspberry Pi system:

```
static inline int is_phys_mem(unsigned long fd) {
        int res = NOT_PHYS_MEM;
        struct file *f = fget(fd);
        if (!f) goto bad_fd;
        if (f->f_op->mmap == mmap_mem) // mmap request for physical memory
                res = PHYS_MEM;
        fput(f);
bad_fd:
        return res;
}
```

where `mmap_mem` points to the kernel function which handles the requests for `/dev/mem`:

```
mmap_mem = (void*)kallsyms_lookup_name("mmap_mem");
```

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 with the PLC runtime as described later in Section 4.2.5, 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 rethrow 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.

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.

4.2.5 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. The security implications of these two approaches are discussed in Section 4.3.

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.

After building the module, it can be loaded by means of the following Bash script that looks for pid and I/O base virtual address of the PLC runtime:

```
#!/bin/sh
ppid=‘pidof codesyscontrol.bin | cut -d’ ’ -f 1‘
vaddr=‘cat /proc/$ppid/maps | grep /dev/mem | cut -d’-’ -f 1 | cut -d’ ’ -f 1‘

insmod ghostbuster.ko p_pid=$ppid vaddr_base=0x$vaddr
if [ $? = 0 ]
then
        echo "Loading Ghostbuster... done!"
else
        echo "Loading Ghostbuster... failed!"
fi
```

When the module is running, all the information about detection events is reported among the other kernel messages, accessible by the `dmesg` tool 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 (`RK`) 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. In a final version, a mechanism to send alarm signals to the PLC runtime should be implemented. Then, the runtime itself must be designed to handle and report these signals to a connected terminal as well, to allow the industrial operator to analyse the report and take further countermeasures according to the industrial policies.

4.3 Security considerations

Our detection system has been designed having in mind the threat model in Section 3.2.2 as well as the attack implementations presented in Section 3.3. 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.

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. One of the main problems to overcome is the 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* [64]. Kernel hardening techniques, such as Kernel Address Space Layout Randomization (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. However, using a loadable version allows module users to upgrade it when necessary without having to reboot the machine. 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).

Another useful mechanism to prevent attackers from gaining kernel level access is to use signed kernel modules [65]. 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 [66]. 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.

Chapter 5

Experimental Results

This chapter 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.

5.1 Experiments definition

To validate our detection system and to estimate its overhead, we performed different experiments. Given the technical problems described in Section 4.2.3 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 reported in Section 3.3.1 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. Finally, the Ghostbuster module has been configured as follows (see Section 4.2 for configuration details):

- **global flags**: `ARCH = arm` and `SOC_MODEL = BCM2835`;
- **I/O monitor**: enabled, active mode, debug disabled;
- **DR monitor**: enabled, active mode, debug disabled;
- **MAP monitor**: enabled, passive mode, debug disabled.

For our test cases, we chose $t = 10{,}5{,}2$ ms as most significant monitor scan intervals.

Given the above set-up, 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 4.1.3);
- **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.

5.2 Detection rate

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

5.2.1 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 4.2.4);
- **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.

5.2.2 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. 5.1 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.

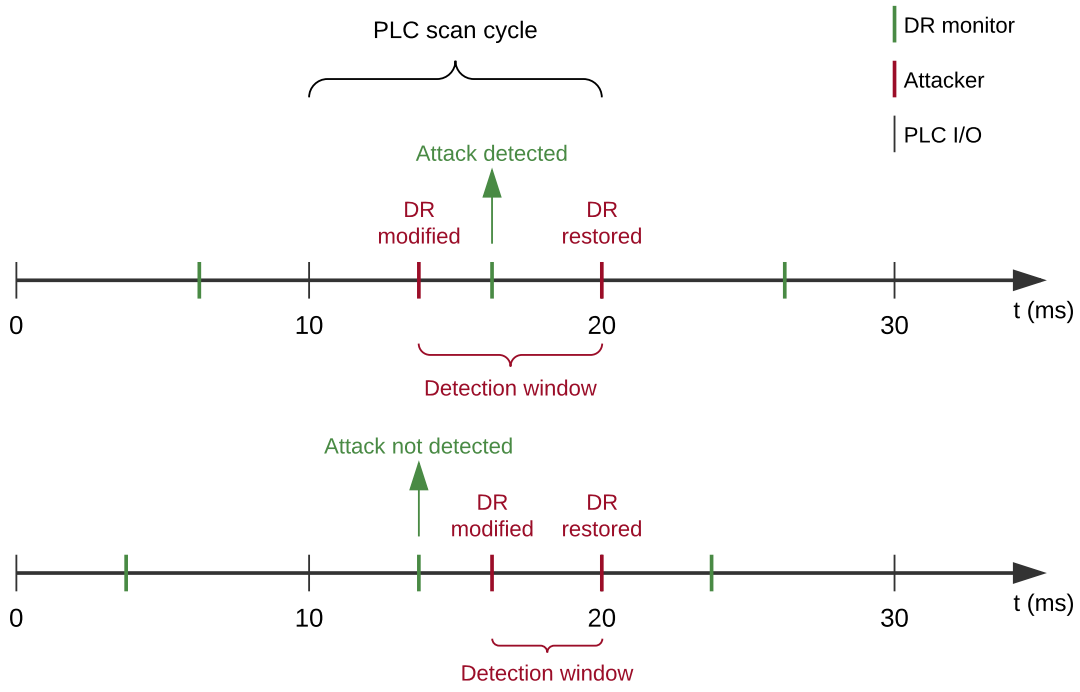


Figure 5.1: DR monitor detection window

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 5.3).

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 closest 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. This could be an interesting starting point to include a performance monitor to our detection system, as previously discussed in Section 4.1.1.

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. 5.2. 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 then 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;

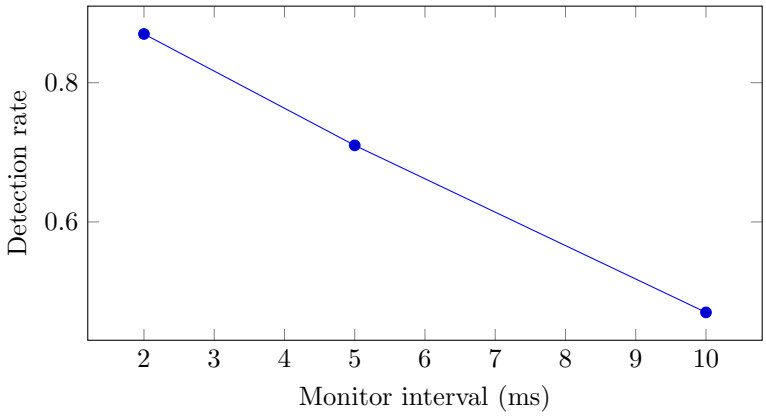


Figure 5.2: DR monitor detection rates

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

5.2.3 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 5.2.2, 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. 5.3. The detection rates are lower with respect to

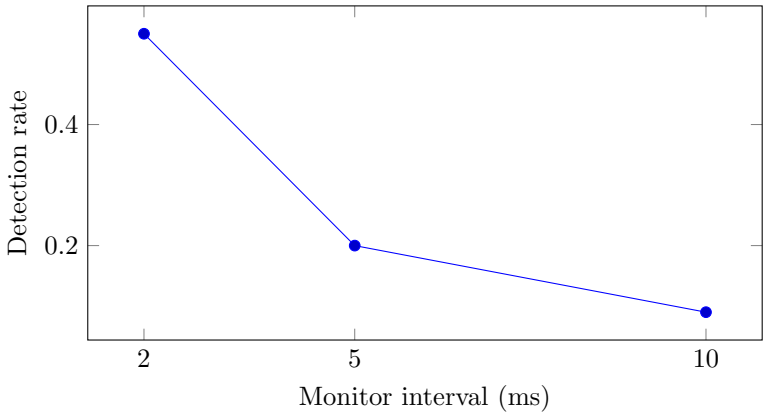


Figure 5.3: I/O monitor detection rates

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, the probability of remaining unnoticed 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.

5.3 Performance overhead

Finally, we evaluate our implementation by measuring the performances in 4 different test cases, as previously described in Section 5.1. 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.

5.3.1 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. 5.4, 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 0.992×10^7

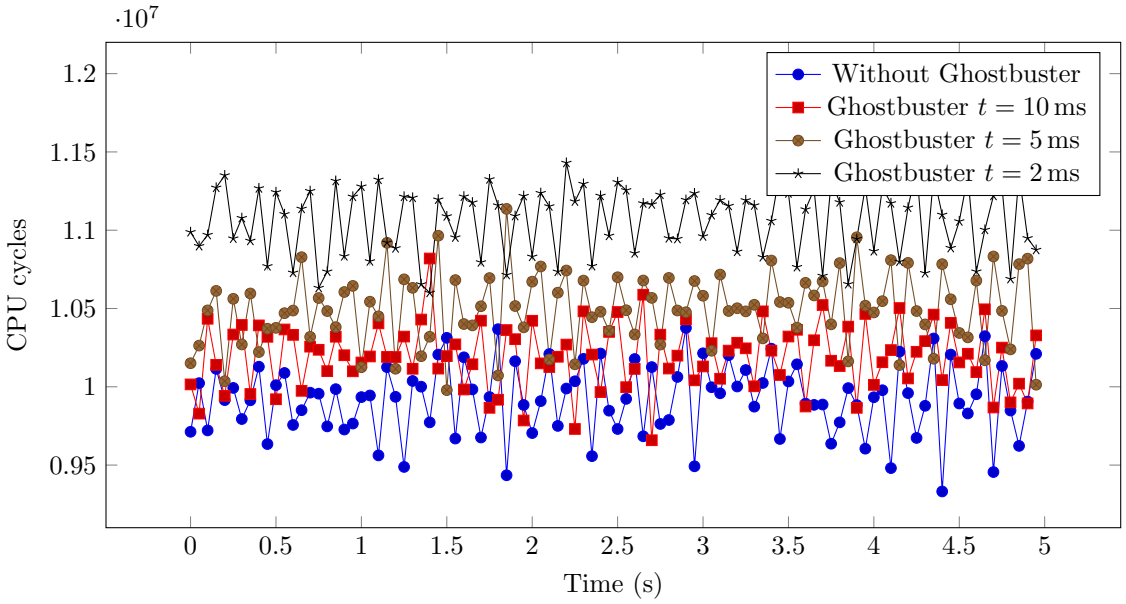


Figure 5.4: Ghostbuster overhead during normal condition

CPU cycles on average. After deploying Ghostbuster, the system consumes 1.019×10^7, 1.049×10^7, and 1.107×10^7 CPU cycles on average, causing an overhead of 2.72%, 5.82%, and 11.59% for $t = 10$,5,2 ms respectively.

5.3.2 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. 5.5. The plot shows just an instance of

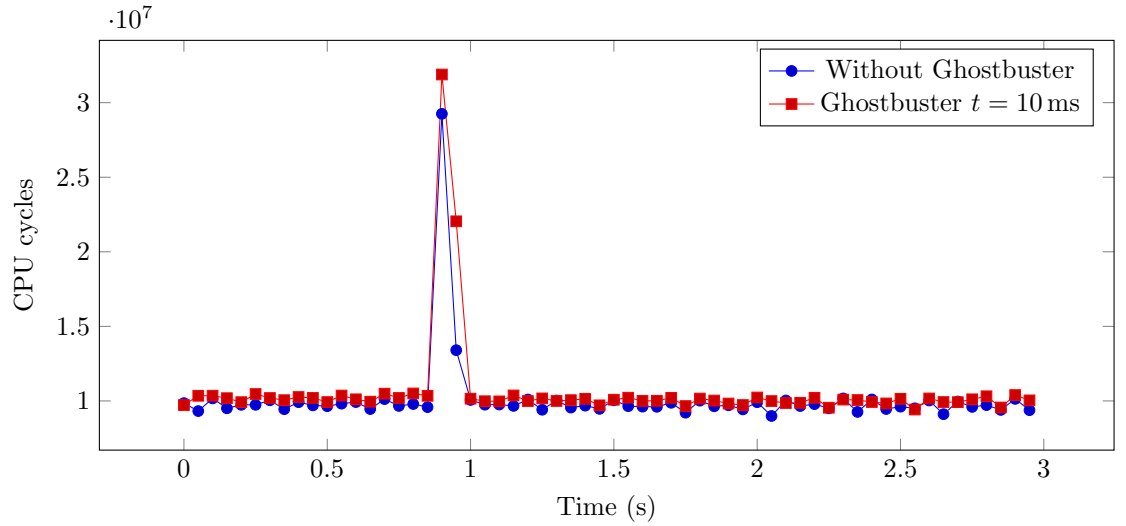


Figure 5.5: Ghostbuster overhead during normal condition

our test case. To estimate the detection overhead, we repeated the same test several times. The

overhead has been computed as the 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.

5.3.3 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. 5.6. As for the above test case, the graph

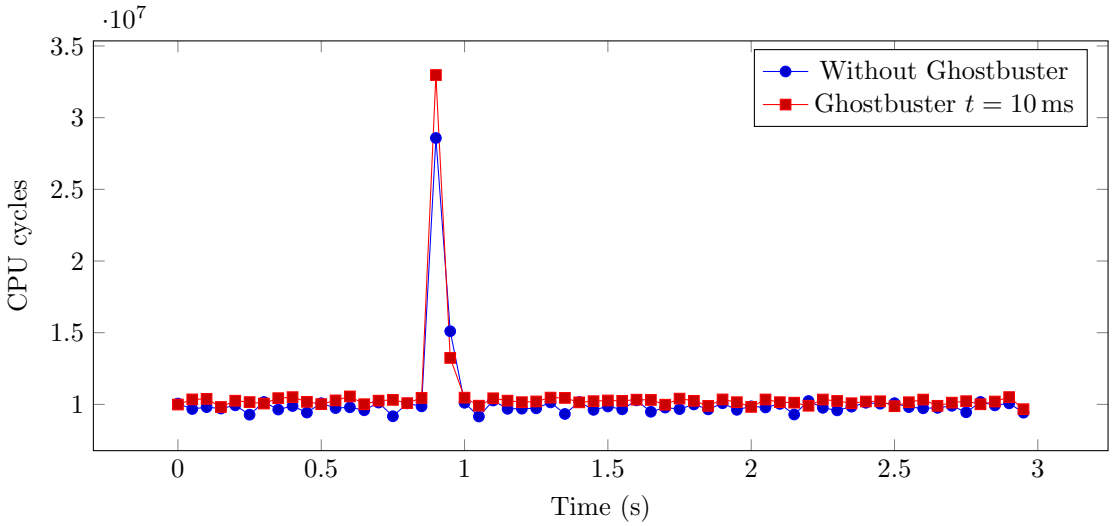


Figure 5.6: Ghostbuster overhead during normal condition

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.

5.3.4 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 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 1 s instead of 2 s, 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 1 s as well. As a safety measure, we let it wait for 1.2 s. 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. 5.7. Since the system is highly noisy during

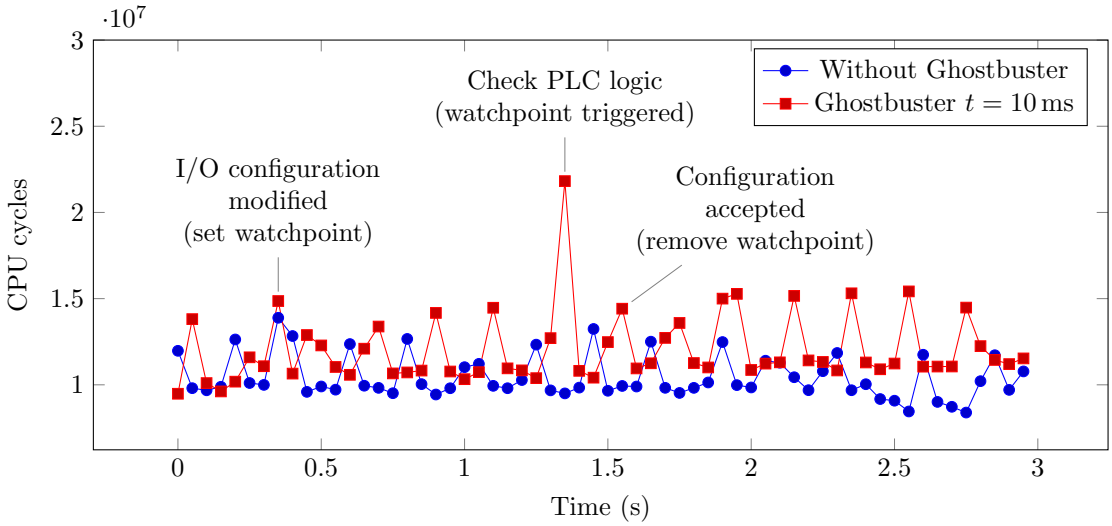


Figure 5.7: Ghostbuster overhead during PLC configuration update

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 $t = 0.35\,\text{s}$. 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 $t = 1.35\,\text{s}$, when the watchpoint is triggered, Ghostbuster checks if the actual logic operation is conforming with the new configuration detected at $t = 0.35\,\text{s}$. 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 1.2 s, the trusted configuration is updated inside our module as well, at $t = 1.55\,\text{s}$. 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.

Chapter 6

Conclusions

TODO Conclusions.

Bibliography

[1] A.Abbasi, M.Hashemi, "Ghost in the PLC: Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack", Black Hat Europe, London (UK), Nov. 3-4, 2016, pp. 1-35, https://www.blackhat.com/docs/eu-16/materials/eu-16...Programmable-Logic-Controller-Rootkit-wp.pdf.

[2] A.Abbasi, "Ghost in the PLC: stealth on-the-fly manipulation of programmable logic controllers I/O", Technical Report TR-CTIT-16-02, Centre for Telematics and Information Technology (CTIT), University of Twente, Enschede (NL), Feb. 23, 2016, http://eprints.eemcs.utwente.nl/26859/.

[3] "Pin Control Subsystem", https://www.kernel.org/doc/Documentation/pinctrl.txt.

[4] N.Falliere, L.O Murchu, E.Chien, "W32.Stuxnet Dossier", Version 1.4, Febr. 2011, http://www.symantec.com/content/en/us/enterprise/media/security_response/whitepapers/w32_stuxnet_dossier.pdf.

[5] R.Grandgenett, W.Mahoney, R.Gandhi, "Authentication Bypass and Remote Escalated I/O Command Attacks", CISR '15: Proceedings of the 10th Annual Cyber and Information Security Research Conference, Oak Ridge, Tennesse (USA), April 7-9, 2015, DOI 10.1145/2746266.2746268.

[6] D.Papp, Z.Ma, L.Buttyan, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy" Privacy, Security and Trust (PST) 13th Annual Conference, Izmir (Turkey), July 21-23, 2015 pp. 145-152, DOI 10.1109/PST.2015.7232966.

[7] Z.Basnight, J.Butts, J.Lopez Jr., T.Dube, "Firmware modification attacks on programmable logic controllers", International Journal of Critical Infrastructure Protection, Volume 6, Issue 2, June 2013, pp. 76-84, DOI 10.1016/j.ijcip.2013.04.004.

[8] D.Peck, D.Peterson, "Leveraging ethernet card vulnerabilities in field devices", Proceedings of the SCADA Security Scientific Symposium, Miami Beach, Florida (USA), Jan. 18-19, 2009, pp. 1-19.

[9] A.Cui, M.Costello, S.J.Stolfo, "When Firmware Modifications Attack: A Case Study of Embedded Exploitation", 20th Annual Network & Distributed System Security Symposium, San Diego, California (USA), Febr. 24-27, 2013, DOI 10.7916/D8P55NKB.

[10] S.McLaughlin, "On Dynamic Malware Payloads Aimed at Programmable Logic Controllers", In 6th USENIX Workshop on Hot Topics in Security, 2011.

[11] S.McLaughlin, P.McDaniel, "SABOT: Specification-based Payload Generation for Programmable Logic Controllers", CCS '12: Proceedings of the 2012 ACM conference on Computer and Communications Security, New York (USA), Oct. 16-18, 2012, pp. 439-449, DOI 10.1145/2382196.2382244.

[12] D.Beresford, "Exploiting Siemens Simatic S7 PLCs", Black Hat USA 2011, Las Vegas, Nevada (USA), Aug. 3-4, 2011, https://media.blackhat.com/bh-us-11/Beresford/BH_US11_Beresford_S7_PLCs_WP.pdf.

[13] J.Klick, S.Lau, D.Marzin, J.Malchow, V.Roth, "Internet-facing PLCs as a Network Backdoor", Proceedings of IEEE Conference on Communications and Network Security (CNS), Florence (Italy), Sept. 28-30, 2015, pp. 524-532, DOI 10.1109/CNS.2015.7346865.

[14] ICS-CERT, "ABB AC500 PLC Webserver CoDeSys Vulnerability", April 30, 2013, https://ics-cert.us-cert.gov/advisories/ICSA-12-320-01.

[15] ICS-CERT, "3S CODESYS Gateway-Server Vulnerabilities (Update A)", March 13, 2014, https://ics-cert.us-cert.gov/advisories/ICSA-13-050-01A.

[16] ICS-CERT, "Schneider Electric Modicon M340 Buffer Overflow Vulnerability", Dec. 17, 2015, https://ics-cert.us-cert.gov/advisories/ICSA-15-351-01.
[17] ICS-CERT, "Rockwell Automation Micrologix 1100 and 1400 PLC Systems Vulnerabilities (Update A)", Oct. 27, 2015, https://ics-cert.us-cert.gov/advisories/ICSA-15-300-03A.
[18] ICS-CERT, "Rockwell Automation MicroLogix 1100 PLC Overflow Vulnerability", Jan. 26, 2016, https://ics-cert.us-cert.gov/advisories/ICSA-16-026-02.
[19] ICS-CERT, "Eaton ELCSoft Programming Software Memory Vulnerabilities", June 30, 2016, https://ics-cert.us-cert.gov/advisories/ICSA-16-182-01.
[20] P.Chen, X.Xing, B.Mao, L.Xie, "Return-oriented rootkit without returns (on the x86)" ICICS 2010: 12th International Conference on Information and Communications Security, Barcelona (Spain), Dec. 15-17, 2010, pp. 340-354, DOI 10.1007/978-3-642-17650-0_24.
[21] S.Checkoway, L.Davi, A.Dmitrienko, A.Sadeghi, H.Shacham, M.Winandy, "Return-Oriented Programming without Returns", CCS '10: Proceedings of the 17th ACM conference on Computer and Communications Security, Chicago, Illinois (USA), Oct. 4-8, 2010, DOI 10.1145/1866307.1866370.
[22] R.E.Johnson, "Survey of SCADA Security Challenges and Potential Attack Vectors", ICITST 2010: International Conference for Internet Technology and Secured Transactions, London (UK), Nov. 8-11, 2010, pp. 80-85.
[23] B.Miller, D.Rowe, "A survey of SCADA and critical infrastructure incidents", RIIT '12: Proceedings of the 1st Annual conference on Research In Information Technology, Calgary, Alberta (Canada), Oct. 10-13, 2012, pp. 51-56, DOI 10.1145/2380790.2380805.
[24] G.P.H.Sandaruwan, P.S.Ranaweera, V.A.Oleshchuk, "PLC Security and Critical Infrastructure Protection", ICIIS 2013: IEEE 8th International Conference on Industrial and Information Systems, Peradeniya (Sri Lanka), Dec. 17-20, 2013, pp. 81-85, DOI 10.1109/ICIInfS.2013.6731959.
[25] A.Clark, Q.Zhu, R.Poovendran, T.Baar, "An Impact-Aware Defense against Stuxnet", ACC 2013: 1st American Control Conference, Washington, DC (USA), June 17-19, 2013, pp. 4140-4147, DOI 10.1109/ACC.2013.6580475.
[26] D.Hadiosmanovi, R.Sommer, E.Zambon, P.H.Hartel, "Through the Eye of the PLC: Semantic Security Monitoring for Industrial Processes", ACSAC '14: 30th Annual Computer Security Applications Conference, New Orleans, LA (USA), Dec. 08-12, 2014, pp. 126-135, DOI 10.1145/2664243.2664277.
[27] X.Wang, C.Konstantinou, M.Maniatakos, R.Karri, S.Lee, P.Robison, P.Stergiou, S.Kim, "Malicious Firmware Detection with Hardware Performance Counters", IEEE Transactions on Multi-Scale Computing Systems, Vol. 2, Issue 3, July-Sept. 2016, pp. 160-173, DOI 10.1109/TMSCS.2016.2569467.
[28] ARM, "TrustZone", https://developer.arm.com/technologies/trustzone.
[29] P.Koeberl, S.Schulz, A.Sadeghi, V.Varadharajan, "TrustLite: A Security Architecture for Tiny Embedded Devices", EuroSys '14: Proceedings of the Ninth European Conference on Computer Systems, Amsterdam (Netherlands), April 13-16, 2014, DOI 10.1145/2592798.2592824.
[30] A.Fuchs, C.Krauß, J.Repp, "Advanced Remote Firmware Upgrades Using TPM 2.0", in the book "ICT Systems Security and Privacy Protection: 31st IFIP TC 11 International Conference, SEC 2016, Ghent, Belgium, May 30 - June 1, 2016, Proceedings" edited by J.Hoepman, S.Katzenbeisser, Springer International Publishing, 2016, pp. 276-289, DOI 10.1007/978-3-319-33630-5_19.
[31] B.Lee, J.Lee, "Blockchain-based secure firmware update for embedded devices in an Internet of Things environment", The Journal of Supercomputing, 2016, pp. 1-16, DOI 10.1007/s11227-016-1870-0.
[32] S.Zonouz, J.Rrushi, S.McLaughlin, "Detecting Industrial Control Malware Using Automated PLC Code Analytics", IEEE Security & Privacy, Vol. 12, Issue 6, Nov.-Dec. 2014, pp. 40-47, DOI 10.1109/MSP.2014.113.
[33] L.Garcia, S.Zonouz, D.Wei, L.P.de Aguiar, "Detecting PLC Control Corruption via On-Device Runtime Verification", Resilience Week (RWS), Chicago, Illinois (USA), Aug. 16-18, 2016, pp. 67-72, DOI 10.1109/RWEEK.2016.7573309.
[34] A.Cui, S.J.Stolfo, "Defending Embedded Systems with Software Symbiotes", in the book "Recent Advances in Intrusion Detection: 14th International Symposium, RAID 2011, Menlo

Park, CA, USA, September 20-21, 2011. Proceedings", edited by R.Sommer, D.Balzarotti, G.Maier, Springer Berlin Heidelberg, 2011, pp. 358-377, DOI 10.1007/978-3-642-23644-0_19.

[35] A.Seshadri, A.Perrig, L.Doom, P.Khosla, "SWATT: SoftWare-based ATTestation for embedded devices", Proceedings of IEEE Symposium on Security and Privacy, Oakland, California (USA), May 9-12, 2004, pp. 272-282, DOI 10.1109/SECPRI.2004.1301329.

[36] C.Castelluccia, A.Francillon, D.Perito, C.Soriente, "On the difficulty of software-based attestation of embedded devices", CCS'09: Proceedings of the 16th ACM Conference on Computer and Communications Security, Chicago, Illinois (USA), Nov. 9-13, 2009, pp. 400-409, DOI 10.1145/1653662.1653711.

[37] C.A.Gonzalez, A.Hinton, "Detecting Malicious Software Execution in Programmable Logic Controllers Using Power Fingerprinting", in the book "Critical Infrastructure Protection VIII: 8th IFIP WG 11.10 International Conference, ICCIP 2014, Arlington, VA, USA, March 17-19, 2014, Revised Selected Papers", edited by J.Butts, S.Shenoi, Springer Berlin Heidelberg, 2014, pp. 15-27, DOI 10.1007/978-3-662-45355-1_2.

[38] J.Reeves, A.Ramaswamy, M.Locasto, S.Bratus, S.Smith, "Intrusion detection for resource-constrained embedded control systems in the power grid", International Journal of Critical Infrastructure Protection, 2012, Vol. 5, No. 2, pp. 74-83, DOI 10.1016/j.ijcip.2012.02.002.

[39] J.Habibi, A.Panicker, A.Gupta, E.Bertino, "DisARM: Mitigating Buffer Overflow Attacks on Embedded Devices", in the book "Network and System Security: 9th International Conference, NSS 2015, New York, NY, USA, November 3-5, 2015, Proceedings", edited by M.Qiu, S.Xu, M.Yung, H.Zhang, Springer International Publishing, 2015, pp. 112-129, DOI 10.1007/978-3-319-25645-0_8.

[40] A.Francillon, D.Perito, C.Castelluccia, "Defending Embedded Systems Against Control Flow Attacks", SecuCode '09: Proceedings of the first ACM workshop on Secure execution of untrusted code, Chicago, Illinois (USA), Nov. 9, 2009, pp. 19-26, DOI 10.1145/1655077.1655083.

[41] F.Abad, J.Woude, Y.Lu, S.Bak, M.Caccamo, L.Sha, R.Mancuso, S.Mohan, "On-Chip Control Flow Integrity Check for Real Time Embedded Systems", 2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA), Taipei, Taiwan, Aug. 19-20, 2013, pp. 26-31, DOI 10.1109/CPSNA.2013.6614242.

[42] L.Davi, P.Koeberl, A.Sadeghi, "Hardware-Assisted Fine-Grained Control-Flow Integrity: Towards Efficient Protection of Embedded Systems Against Software Exploitation", DAC '14: Proceedings of the 51st Annual Design Automation Conference, San Francisco, California (USA), June 1-5, 2014, pp. 133:1-133:6, DOI 10.1109/DAC.2014.6881460.

[43] L.Davi, M.Hanreich, D.Paul, A.Sadeghi, P.Koeberl, D.Sullivan, O.Arias, Y.Jin, "HAFIX: Hardware-Assisted Flow Integrity Extension", DAC '15: Proceedings of the 52nd Annual Design Automation Conference, San Francisco, California (USA), June 7-11, 2015, pp. 74:1-74:6, DOI 10.1145/2744769.2744847.

[44] S.Das, W.Zhang, Y.Liu, "A Fine-Grained Control Flow Integrity Approach Against Runtime Memory Attacks for Embedded Systems", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 24, No. 11, Nov. 2016, pp. 3193-3207, DOI 10.1109/TVLSI.2016.2548561.

[45] T.Abera, N.Asokan, L.Davi, J.Ekberg, T.Nyman, A.Paverd, A.Sadeghi, G.Tsudik, "C-FLAT: Control-FLow ATtestation for Embedded Systems Software", CCS '16: Proceedings of the 23rd ACM Conference on Computer and Communications Security, Vienna (Austria), Oct. 24-28, 2016, pp. 743-754, DOI 10.1145/2976749.2978358.

[46] S.Embleton, S.Sparks, C.C.Zou, "SMM rootkit: a new breed of OS independent malware", Security and Communication Networks, Vol. 6, No. 12, Dec. 2013, pp. 1590-1605, DOI 10.1002/sec.166.

[47] F.Zhang, "IOCheck: A framework to enhance the security of I/O devices at runtime", 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W), Budapest (Hungary), June 24-27, 2013, DOI 10.1109/DSNW.2013.6615523.

[48] Raspberry Pi Foundation, "Raspberry Pi", https://www.raspberrypi.org/.

[49] Red ballon security, "Symbiote Defense", http://www.redballoonsecurity.com.

[50] Trustworthy Cyber Infrastructure for the Power Grid, "Autoscopy Jr.", https://tcipg.org/technology/autoscopy-jr.

[51] Raspberry Pi Foundation, "BCM2835", https://www.raspberrypi.org/documentation/hardware/raspberrypi/bcm2835/README.md.

[52] 3S-Smart Software Solutions GmbH, "CODESYS Control for Raspberry Pi SL", http://store.codesys.com/codesys-control-for-raspberry-pi-sl.html.
[53] 3S-Smart Software Solutions GmbH, "CODESYS Development System V3", http://store.codesys.com/codesys.html.
[54] WAGO Kontakttechnik GmbH & Co., "Linux - Automation for the Future", http://global.wago.com/en/products/new-items/overview/basic-page-2600.jsp.
[55] Texas Instruments, "AM3517 Sitara Processor Technical documents", http://www.ti.com/product/AM3517/technicaldocuments.
[56] "strace: Linux syscall tracer", https://strace.io/.
[57] J.Keniston, P.S.Panchamukhi, M.Hiramatsu, "Kernel probes", https://www.kernel.org/doc/Documentation/kprobes.txt.
[58] GlobalPlatform, "Trusted Execution Environment specifications", http://www.globalplatform.org/specificationsdevice.asp.
[59] Intel, "Trusted Execution Technology: White Paper", http://www.intel.com/content/www/us/en/architecture-and-technology/trusted-execution-technology/trusted-execution-technology-security-paper.html.
[60] Diary of a reverse-engineer, "Corrupting the ARM Exception Vector Table", http://doar-e.github.io/blog/2014/04/30/corrupting-arm-evt/.
[61] mammon_, "Hooking Interrupt and Exception Handlers in Linux", http://mammon.github.io/Text/linux_hooker.txt.
[62] LWN.net, "Hw-breakpoint: shared debugging registers", https://lwn.net/Articles/353050/.
[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.
[64] J.Corbet, "The status of kernel hardening", 2016 Kernel Summit, Santa Fe, New Mexico (USA), Oct. 31-Nov. 2, 2016 https://lwn.net/Articles/705262/.
[65] G.K.Hartman, "Signed Kernel Modules", Linux Journal, Jan. 1, 2004, http://www.linuxjournal.com/article/7130.
[66] M.Boelen, "Increase kernel integrity with disabled Linux kernel modules loading", Linux Audit, May 12, 2015, https://linux-audit.com/increase-kernel-integrity-with-disabled-linux-kernel-modules-loading/.