Ghost in the PLC
Designing an Undetectable Programmable Logic Controller Rootkit via Pin Control Attack

Ali Abbasi[1] and Majid Hashemi[2]

[1] Distributed and Embedded Systems Security Group, University of Twente, The Netherlands,
{a.abbasi}@utwente.nl
[2] QuarksLab, France
mhashemi@quarkslab.com

Abstract. Input/Output is the mechanisms through which embedded systems interact and control the outside world. Particularly when employed in mission critical systems, the I/O of embedded systems has to be both reliable and secure. Embedded system's I/O is controlled by a pin based approach. In this paper, we investigate the security implications of embedded system's pin control. In particular, we show how an attacker can tamper with the integrity and availability of an embedded system's I/O by exploiting cerain pin control operations and the lack of hardware interrupts associated to them.

Keywords: Pin, SoC, Exploit, Attack, PLC, Rootkit

1 Introduction

Embedded systems are widely used today in a variety of applications, such as consumer, industrial, automotive, medical, commercial and military. As such, they are often employed in mission critical systems that have to be both reliable and secure. In particular, it is important that their I/O (Input/Output) be stable and secure [1], as this is the way they interact with the outside world.

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

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

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

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

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

To demonstrate the feasibility of our attack technique, in Section 7 we describe the practical implementation of an attack against a Programmable Logic Controller (PLC) environment by exploiting the runtime configuration of the I/O pins used by the PLC to control a physical process. The attack allows one to reliably take control of the physical process normally managed by the PLC, while remaining stealth to both the PLC runtime and operators monitoring the process through a Human Machine Interface, a goal much more challenging than simply disabling the process control capabilities of the PLC, which would anyway lead to potentially catastrophic consequences. The attack does not require modification of the PLC logic (as proposed in other publications [2, 3]) or traditional kernel tampering or hooking techniques, which are normally monitored by anti-rootkit tools.

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

Finally, in Section 8.1 we discuss potential mechanisms to detect/prevent Pin Configuration exploitation. However, because the pin configuration does happen legitimately at runtime and the lack of proper interrupt notifications from the SoC, it seems non-trivial to devise monitoring techniques that are both reliable and sufficiently light-way to be employed in embedded systems.

2 Background

2.1 Attack Techniques

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

- Firmware modification attacks: in recent years, a number of firmware modification attacks against embedded devices have been researched and discussed. Cui et al. [4] demonstrated how the HP-RFU firmware update protocol can be exploited to allow adversaries to inject malicious firmware into HP printers. Traynor et al. [5] showed how to recursively compromise embedded devices and use them to create a network of malicious devices by manipulating their firmware. Wegner [6] demonstrated how to install a backdoor into Siemens office telephone communication devices by exploiting a vulnerability in their firmware verification system. Basnight et al. [7] illustrated that it is feasible to execute arbitrary code in a PLC by exploiting the firmware update feature, and finally, Peck et al. [8] showed how to exploit the Ethernet module of a PLC by uploading malicious firmware to it.
- Configuration manipulation attacks: these attacks allow an adversary to modify critical configuration parameters of an embedded device to force it to misbehave. For example, an anonymous security researcher with the nickname PT [9] demonstrated how to obtain access to a Private Branch Exchange (PBX), an embedded device used for telephone systems, by exploiting a vulnerability in the proprietary authentication protocol used by one vendor. A special case of configuration manipulation attacks concerns programmable devices, such as PLCs. PLCs can be programmed to control a physical process by following the logic specified by the user. In this case, the attack consists of uploading a malicious logic to alter the manner in which the process is controlled. Falliere et al. [10] reported that the Stuxnet malware was used to manipulate the logic of PLCs from a programming station to subvert part of the uranium enrichment process at Natanz (Iran). In [3, 2], McLaughlin et al. introduced two techniques for the dynamic generation of a malicious PLC control logic. To the best of our knowledge, the techniques proposed by McLaughlin et al. are, for the moment, limited in their practical applicability and have never been used in real-world attacks.
- Control-flow attacks: in general, this category of attacks consists of manipulating the execution flow of a running process. This is typically achieved by exploiting a stack/heap overflow or use-after-free vulnerability, which allows for the execution of arbitrary code by an adversary. Jump- and return-oriented programming (JOP and ROP) are considered to be control-flow attacks. Recent research has illustrated the possibility of control-flow attacks in embedded devices. For example, Beresford [11] presented multiple protocol vulnerabilities in Siemens PLCs that can allow an adversary to perform a remote code execution attack. Wightman demonstrated that Schneider Elec-

tric PLCs are vulnerable to buffer overflow attacks [12, 13]. Heffner [14–16] presented multiple memory corruption vulnerabilities in home routers.
Although several techniques have been proposed to detect or prevent control-flow attacks on general IT systems, this class of attacks remains one of the most dangerous. Effective countermeasures that are simultaneously applicable in the domain and not circumventable by adversaries have yet to be developed. For example, Schuster et al. [17] evaluated several detection techniques for control-flow attacks [18–20] and claimed that attackers can bypass them using the code sequence within the executable modules of the target program. Davi et al. [21] introduced several techniques for bypassing detection techniques for control-flow attacks in multiple system security products [18, 20, 22]. Specifically, they showed not only that adversaries can find sufficient ROP gadgets within a program's binary code but also that by using long loops of NOP gadgets, they can create a long gadget chain and thereby break detection mechanisms for control-flow attacks.

2.2 Detection Techniques

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

- Firmware integrity verification: verifying the integrity of firmware allows one to detect or prevent firmware modification attacks. Such verification can be performed by the host when storing new firmware or at runtime.
  Adelstein et al. [23] introduced a firmware-signing method that consists of a "certifying compiler" for firmware. The compiler allows the firmware to be verified at runtime by checking certain properties of the execution flow, memory and stack integrity in the firmware. Zhang et al. [24] introduced IOCheck, a framework to verify at runtime the integrity of firmware and the I/O configuration of computer I/O peripherals. After a (assumed trusted) BIOS boot, IOCheck leverages the System Management Mode of x86 CPU architectures to perform integrity checks that can be either executed at random polling intervals or driven by specific events. Finally, Duflot et al. [25] introduced NAVIS, a framework for the detection of firmware integrity manipulation in the memory of a network card by inspecting the memory accesses performed by the NIC processor against a model of expected behavior based on the memory layout profile of the adapter. A memory access that is outside the NIC memory profile is interpreted as an attempt to manipulate the NIC firmware.
- Memory verification: these techniques verify the integrity of executable code in memory at runtime. The most common technique for memory verification is attestation, which is used for low-power embedded devices. Attestation is a challenge-response technique that allows an external application (the verifier) to verify the integrity of (parts of) the state of a system (the prover)

against malicious modifications. Attestation techniques typically require the availability of dedicated hardware (e.g., a Trusted Platform Module). However, because of the practical limitations in embedded devices, certain works have focused on the development of pure software-based attestation techniques.

Seshadri et al. [26] introduced SWATT, a software-based attestation technique that can remotely verify the runtime memory contents of embedded devices and discover malicious modifications. SWATT uses a challenge-response protocol to remotely control the memory content of the embedded devices.

LeMay et al. [27] proposed an ad hoc static kernel for smart meters that can cryptographically sign every new firmware version uploaded to a device. The signature is sent to the verifier to attest that the current (and previous) firmwares loaded on the smart meter are legitimate and integer.

Armknecht et al. [28] introduced a framework for evaluating the security of software-based remote attestation techniques. The authors discussed the security properties of common basic cryptographic functions, such as pseudo-random number generators (PRNGs) and hash functions, when used for attestation purposes. They also discussed the possibility of leveraging time as a verification parameter to strengthen the security of an attestation scheme.

In an approach different from that of memory attestation frameworks, Cui et al. [29] proposed a new host-based deployment mechanism for embedded devices running operating systems, which they called a Symbiotic Embedded Machine or symbiote. The mechanism is specifically designed to inject intrusion detection functionality into the firmware of such devices and to verify the integrity of its executable parts. A symbiote is a code structure embedded in a piece of firmware that can closely co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. The symbiote is embedded in a randomized fashion to protect itself from removal, and the execution context of the symbiote is separated from that of the operating system to make it more resistant against adversaries. The authors demonstrated the deployment of a symbiote in the Cisco IOS firmware, with a low performance penalty and without an impact on the router's functionality. Symbiotes cannot continuously monitor the entire firmware but rather set specific watchpoints and monitor certain executable locations of the firmware.

– Control-flow integrity: the vast majority of control-flow hijacking attacks operate by exploiting memory corruption bugs, such as buffer overflows, to control an indirect control-flow transfer instruction in the vulnerable program, most commonly a function pointer or return address. CFI mechanisms counter control-hijacking attacks by ensuring that the control flow remains within the control-flow graph (CFG) intended by the programmer.

  In the context of CFI approaches for embedded devices, Reeves et al. [30] introduced a host-based intrusion detection system for embedded devices that leverages a built-in kernel tracing framework to identify control-flow anomalies in syscalls. The system is constructed by learning, for each mon-

itored syscall, a list of known good source addresses. During detection, the system checks that when a certain syscall is invoked, the source of the call is on the safe list. Although its detection capabilities are limited, the approach also imposes a limited overhead on the system, which makes it suitable for being deployed in embedded devices such as those used in power grids (RTUs, IEDs and PLCs). Other CFI approaches for embedded devices are hardware-assisted. Special hardware modifications to the devices are needed to support the proposed approaches. The authors motivate the need for hardware modifications by citing the limited processing capabilities of embedded devices or the lack of features required by existing CFI approaches (e.g., memory management units or execution rings) in simpler, low-cost, embedded devices. Abad et al. [31] introduced a hardware-assisted CFI system for embedded devices. The system employs a dedicated hardware component to compare the control flow of the embedded device firmware at runtime to the CFG. The graph is constructed by decompiling the binary of the application to be protected. However, the method proposed for constructing the CFG does not consider indirect control-flow transfers (e.g., indirect function calls); therefore, the approach is incomplete and prone to certain types of control-flow attacks in which the adversary manipulates the control flow of the target application by changing the values of data memory areas. For example, the adversary may first spray the heap memory with shellcode instructions and then overwrite the value of a function pointer to point to a random heap address, which may contain the shellcode. Francillon et al. [32] proposed a hardware-assisted protection mechanism for AVR microcontrollers against control-flow attacks. The mechanism consists of separating the stack into a data stack and a control stack. The control stack is hardware-protected against unintended or malicious modifications (i.e., those not performed by call or ret instructions). Finally, Davis et al. [33] proposed a hardware-assisted CFI scheme that uses the hardware to confine indirect calls. This CFI scheme is based on a state model and a per-function CFI labeling approach. In particular, the CFI policies ensure that function returns can only transfer control to active call sides (i.e., return landing pads of currently executing functions). Furthermore, indirect calls are restricted to target the beginning of a function, and finally, behavioral heuristics are used to address indirect jumps.

3 Detection Mechanisms Applicable to PLCs

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

– Designed for embedded devices that run modern operating systems: there is a group of embedded devices, called low-powered embedded devices, that do not have an operating system (OS). Devices that run microcontroller-based

processors (such as AVR or ATMEL) can be considered as low-powered embedded devices. However, most of the PLCs have a *real* OS. Therefore we only consider approaches that target embedded devices with a *real* OS.
- No hardware modification: the performance limitations make it difficult to introduce a complete host-based security mechanism for PLCs. Most of the solutions described in Section 2 attempt to overcome these limitations by first considering hardware modifications of the embedded devices, thus making those solutions less attractive.
- Not dependent to virtualization: majority of embedded processors do not support virtualization. Therefore, any implementation which is purely based on virtualization can not be considered as solid solution for embedded systems.

Based on the parameters, we can identify two host-based detection mechanisms that, unlike most of the techniques described in Section 2.2, possess both desired qualifications. These host-based detection systems are Autoscopy Jr. [30] and Doppelganger [29]. For the sake of completeness, we use a third condition, namely CPU overhead, for applicable host-based detection systems for PLCs. In embedded devices, high CPU overhead is an important issue for host-based detection systems. Large CPU overhead makes host-base defenses for embedded systems less practical since the CPU resources in such systems are very limited. We consider the same threshold achieved by the authors of Autoscopy Jr. [30] as an acceptable overall CPU overhead. Considering the CPU overhead limit as a new condition, leads to the same selected defensive solutions since both satisfy this new requirement.

These solutions are practically applicable, and because of their practical approach, they were adopted in the industry immediately upon their introduction [30, 34]. However, these approaches also exhibit certain weaknesses. Understanding these weaknesses assist us in designing better host-based solutions for embedded devices.

- Autoscopy Jr.: Autoscopy Jr. is a kernel control-flow monitoring system that searches for control-flow anomalies caused by function hooking in the kernel [30]. Autoscopy Jr. incurs only 5% CPU overhead, which is a significant achievement for a host-based detection system for embedded devices. Autoscopy Jr. specifically searches for kernel attacks in which the malicious code manipulates a function pointer. When a process calls a function with a manipulated pointer, the call is diverted to the malicious function instead of a legitimate one. The malicious function can then decide either to never call the original function or to call the legitimate function with a manipulated input. Autoscopy Jr. operates in two phases:
  1. Learning phase: In the learning phase, Autoscopy Jr. installs a character device driver that allows it to access the kernel memory by invoking `ioctl()`. Next, Autoscopy Jr. uses the device driver to monitor direct and indirect function calls and their corresponding return addresses. Afterward, it saves the return addresses of these functions, with certain

runtime information (such as function arguments), to a data structure called the Trusted Location List (TLL). It then uses the TLL during the detection phase.

2. Detection phase: During the detection phase, Autoscopy Jr. uses the previously installed device driver to monitor function calls. When a function that is listed in TLL is called, Autoscopy Jr. verifies the function address against the TLL entry for the same function. If the function address is not found in the TLL, it generates an alert.

– Doppelganger: Doppelganger is a host-based intrusion detection solution for embedded devices. It can detect both kernel- and application-level attacks in embedded devices. Doppelganger first analyzes the firmware of the embedded device to detect live code regions therein. Live code regions are executable parts of the firmware. Once Doppelganger detects the executable area of the memory, it randomly inserts its symbiotes (watchpoints) into the detected live code areas. Doppelganger symbiotes contain a CRC32 checksum of the randomly selected live code regions.

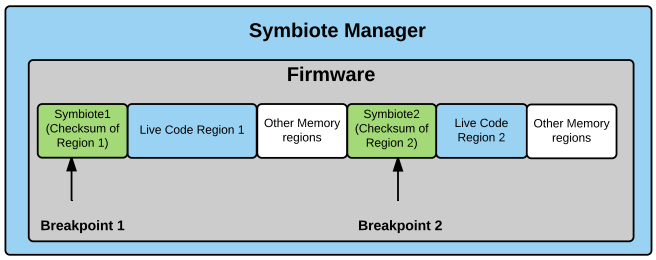

Fig. 1. Structure of embedded device firmware controlled by Doppelganger

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

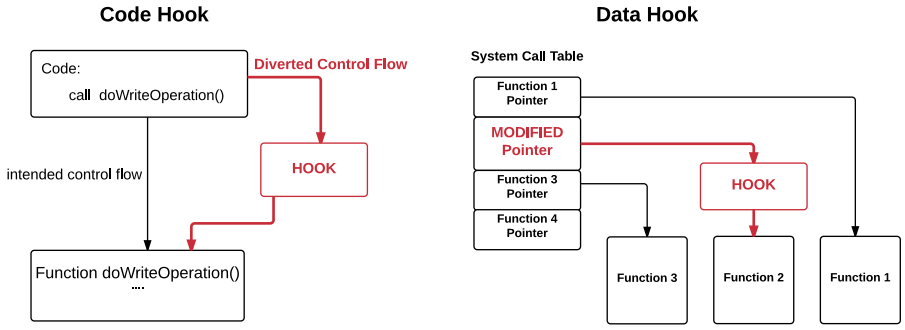

Fig. 2. Typical function hooking

4 Methodology for Evading Defensive Mechanisms

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

- Static referencing: Both Autoscopy Jr. and Doppelganger use static references to verify the execution flow or the integrity of an executable code region. Static referencing is comparable to signature-based approaches. If an attacker avoids the explicitly defined references, he can evade detection.
  The static references in Autoscopy Jr. are the entries of the TLL. In Doppelganger, the static references are the symbiotes. None of these references can be modified during runtime. Autoscopy Jr. requires an additional learning phase to add more entries to the TLL, and Doppelganger requires the recreation of the firmware to insert additional symbiotes. These requirements limit the capabilities of both Doppelganger and Autoscopy Jr.: if an attacker inserts malicious code into locations that are not considered among the static references, then this malicious code can not be detected.
- Function hooking: In general, there are two types of function hooks: *code hooks* and *data hooks* [35, 36]. Both types of hooks are illustrated in Figure 2. In code hooking, an attacker can divert function calls by modifying executable parts of the kernel, such as the *.text* section. If the attacker wishes to hook the function call `doWriteOperation()`, as illustrated in Figure 2, he modifies the executable instructions that call `doWriteOperation()` to instead call its hook.
  In data hooking, the attacker does not manipulate executable instructions; instead, he modifies the function pointers in the System Call Table (or other

similar tables, such as the System Service Dispatch Table) to call their hooks. The System Call Table consists of pointers to system call functions. If an attacker modifies a function pointer and that function is then called by a process, the OS calls the *hook function* instead of the original function.

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

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

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

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

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

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

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

5 Pin Control in Embedded Systems

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

5.1 Pin Multiplexing

Embedded SoCs usually employ hundreds of pins connected to the electrical circuit. Some of these pins have a single defined purpose. For example, some only provide electricity or a clock signal. Since different equipment vendors with diverse I/O requirements will use these SoCs, the SoC manufacturer produces its SoCs to use a certain physical pin for multiple mutually exclusive functionalities, depending on the application [37]. The concept of redefining the functionality of the pin is called Pin Multiplexing and is one of the necessary specifications of the SoC design [38, 39]. For example the SoC in Figure 3 has multiplex pins for JTAG/SPI, SPI/GPIO, MMC/GPIO and I2C/GPIO. In Figure 3 each multiplex pins, gives a vendor options to choose between those two functionalities. Regarding the interaction of the Pin Multiplexing with OS, it is recommended by SoC vendors to only multiplex the pins during the startup since there is no interrupt for multiplexing. However the user still can multiplex a pin at runtime and there is no limitation on that.

5.2 Pin Configuration

Embedded SoC I/Os (e.g. ARM, MIPS or PowerPC) are controlled with a pin based approach and must be configured otherwise these can not function prop-

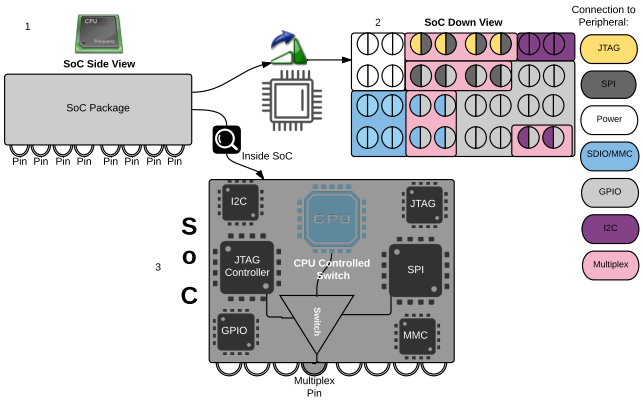

Fig. 3. Part 1 shows an SoC from the Side view. Part 2 shows the design of multiple multiplex pins with different I/O peripheral. Part 3 shows how SoCs peripherals are located inside a SoC and how one multiplex pin is connected to two peripheral.

erly. The configuration can be divided to two groups: Configuration at boot-time and configuration at runtime.

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

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

5.3 How the OS configures pins

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

Once the initial multiplexing setup is finished, the start-up configuration of the pins initiates. Once again, the configuration starts by mapping the physical address of registers related to pin configuration at run-time to virtual addresses. However, this time a device driver performs the task. After mapping, the driver will write appropriate configuration detail to those virtual addresses that eventually get mapped to the physical configuration registers in the SoC.

5.4 Security concerns regarding Pin Control

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

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

6 Pin Control Attacks

In this section, we describe a new type of attack that targets PLCs based on the methodology described in Section 4. A pin control attack basically consists

of misusing the pin control functionalities of the embedded system at runtime. An attacker can either block communications with peripherals, cause physical damage to them or manipulate values read or written to/from a peripheral by a legitimate process.

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

To cause physical damage, an attacker can use a combination of Pin Configuration and Pin Multiplexing. For example, a pin which can be multiplexed between an MMC (memory controller) and a PWM (Pulse-width modulation) controller can be targeted to cause physical damage. If the pin is multiplexed to be used as an MMC controller, an attacker can multiplex the pin to connect it to a PWM controller and modify the PWM controller to push a significant pulsing electrical current toward the memory controller causing the memory to burn. An alternative option would be writing to Configuration and Multiplexing registers consecutively. Some of the Pin Configuration and Multiplexing registers are either located in NVRAM or EEPROM. Both of these memories have finite write cycle life, usually between 100 to 100K. Once the write operation life cycle of such memories ends, the SoC will be useless since it will not be any more a non-volatile memory. An attacker can just write to this memories consecutively. With this technique one can brick any existing embedded SoC within a day.

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

Finally the novelty of our attack lies in the fact that to manipulate the physical process we do not modify the PLC logic instructions or firmware [3, 2, 7, 11, 4]. Instead, we target the interaction between the firmware and the PLC I/O. This can be achieved without leveraging traditional function hooking techniques and by placing the entire malicious code in dynamic memory (in rootkit version of the attack), thus circumventing detection mechanisms such as Autoscopy Jr. and Doppelganger. Additionally, the attack causes the PLC firmware to assume that it is interacting effectively with the I/O while, in reality, the connection between the I/O and the PLC process is being manipulated.

6.1 Threat Model

For pin control attack we consider three requirements which the attacker must satisfy. These three requirements are the followings:

- *PLC runtime privilege:* we can envision an attacker with an equal privilege as the PLC runtime process which gives her the possibility to modify the pin configuration registers. Since the PLC runtime can modify such configuration registers, having equal privilege means that an attacker can also modify those registers. In recent years, multiple research has shown that the PLCs from multiple vendors such as Siemens [40], [11], ABB [41], Schneider Electric [42], [12], Rockwell Automation [43], and WAGO [44] are vulnerable to system-level code execution via the memory corruption vulnerabilities. Therefore, we can argue that getting equal privilege as PLC runtime is not a farreaching assumption.
- *Knowledge of the physical process:* we also assume that the attacker is aware of the physical process on the plant. Attacking critical infrastructure is usually carried out by state-sponsored attackers and usually such actors study their targets before launching their attack. For example, as reported [45] in Stuxnet [10] case, the attackers were very well aware of the physical process in their target plant. Therefore, we can assume that it is feasible for other state-sponsored attackers to study their target physical process and be aware of it.
- *Knowledge of mapping between I/O pins and the logic:* we assume that the attacker is aware of the mapping between the I/O pins and the logic. The PLC logic might use various inputs and outputs to control its process; thus, the attacker must know which input or output must be modified to affect the process as desired. The mapping between I/O pins and logic is already available in the PLC logic and therefore, an attacker can access to it within the PLC without any limitation. Additionally, the works presented by McLaughlin et al. [3], [2] can be used to discover the mapping between the different I/O variables and the physical world. Thus we can argue that it is reasonable to assume the attacker can be aware of the mapping between I/O pins and the logic.

7 A pin control attack in practice

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

PLCs play a significant role in the industry since they control and monitor industrial processes in critical infrastructures [46]. For this reason, the successful exploitation of a PLC can affect the physical world and, as a result, can have serious consequences for the safety of equipment and human life [1]. For example, an adversary may manipulate the value of tank pressure sensors in a pressure sensitive boiler thus leading to the explosion of the boiler, or, similarly

to Stuxnet, change the frequency of variable speed drives of centrifuges in a uranium enrichment facility, leading to damage of the centrifuge cascades. Consequently, one of the main objectives when attacking a PLC is to manipulate the physical process by intercepting the signals received from sensors and altering the ones sent to the actuators controlled by the PLC in such a way that the PLC has no way to tell that his communication with the I/O is being manipulated. This can be achieved by pin control exploitation without leveraging traditional function hooking or kernel data structure modification techniques [47].

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

1. For write operations: if the PLC software is attempting to write a value to an I/O pin that is configured as output, the attacker reconfigures the I/O pin as input. The write operation will not succeed, but the PLC software will be unaware of this.
2. For read operations: if the PLC software is attempting to read a value from an I/O pin that is configured as input, the attacker can reconfigure the I/O pin as output and write the value that he wishes to feed to the PLC software in the reconfigured pin.

We implement this strategy in two variants of the attack. In the first variant we assume the attacker has root access to the PLC. In the second variant we assume the attacker the same access level of the PLC software.

However, before discussing the technical implementation of the attack, we discuss in more details how a PLC generally works, and how the specific environment in which we built our attack is set up.

7.1 PLC operations

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

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

After pin configuration, the PLC runtime executes the instructions in the logic in a loop (the so-called program scan). In a typical scenario, the PLC

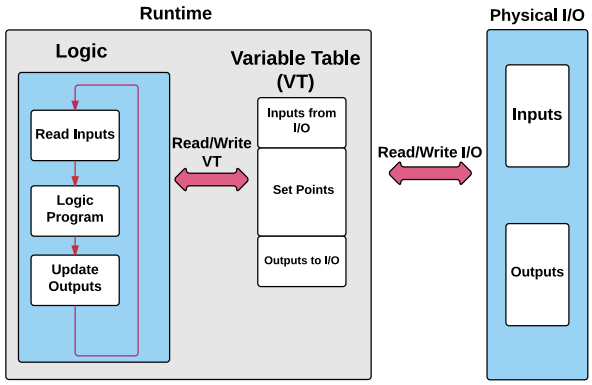

Fig. 4. Overview of PLC runtime operation, the PLC logic and its interaction with the I/O

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

A PLC typically comprises separate digital and analog inputs and outputs. Because PLCs are digital systems, they cannot control analog input and output without additional hardware components. Digital-to-Analog Converters (DACs) for analog outputs and Analog-to-Digital Converters (ADCs) for analog inputs form part of the analog interface of a PLC. These components read or write analog values by converting them to or from digital outputs or inputs to allow the PLC to interact with its analog interfaces. The DACs and ADCs are not separate components of the PLC but rather an integral part of the PLC circuit board. One can argue that the basis of I/O interaction in PLCs is digital. Analog control is simply a conversion of digital signals into analog signals or analog signals into digital signals.

7.2 Environment setup

Target device and runtime To mimic a PLC environment we choose two platforms. First a complex set of I/O modules in a Raspberry Pi 1 model B. We used a Raspberry Pi, because of the similarity in CPU architecture, available memory, and CPU power to a real PLC. The Raspberry Pi 1 uses a Broadcom BCM2835 single-core processor with a clock speed of 700 MHz. The Raspberry

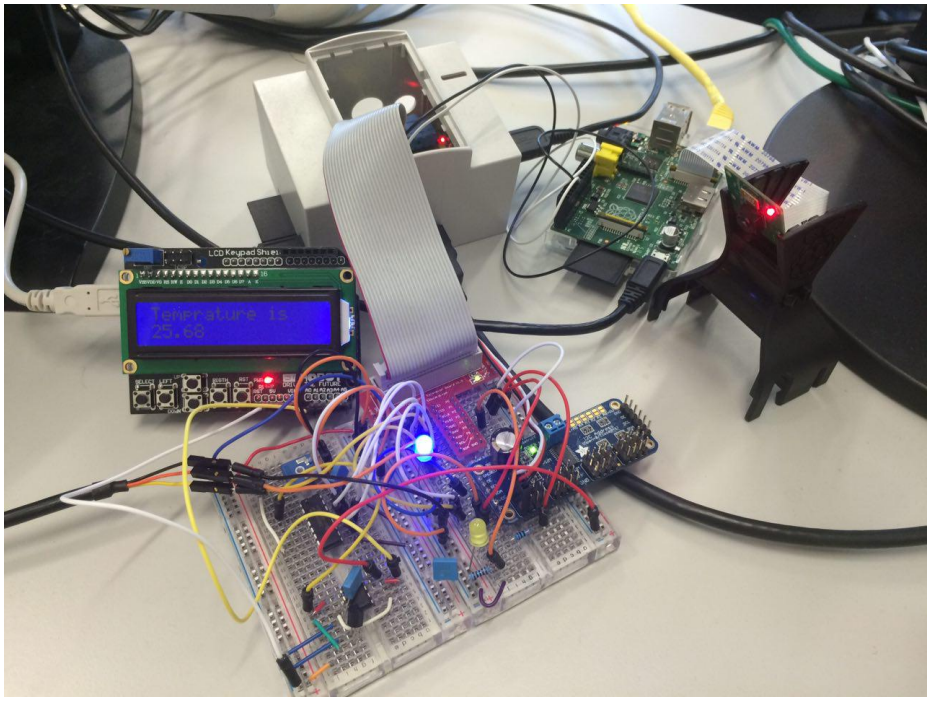

Fig. 5. Terget Raspberry Pi 1 running Codesys runtime, connected to multiple I/O interfaces

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

For the second target platform we use Wago 750-8202 with a 8-Channel Digital Input/Output Module 24 V DC (WAGO 750-1506).

For PLC runtime, we use the Codesys platform (both Codesys and e!Cockpit). Codesys is a PLC runtime that can execute ladder logic or function block languages on proprietary hardware and provides support for industrial protocols such as Modbus and DNP3. Currently, more than 260 PLC vendors use Codesys as the runtime software for their PLCs [44]. Figure **??** depicts our first target platform connected to multiple analog and digital I/Os via ADC and DAC controllers. Figure **??** depicts our second target platform (a real industrial PLC) connected to a digital I/O controlling an LED.

The logic and the physical process for the Raspberry Pi

We use pins 22 and 24 of the Raspberry Pi (and) to control our physical process. In our logic, we declare pin 22 as the output pin and pin 24 as the input pin. In the physical layout, our output pin is connected to an LED and our input

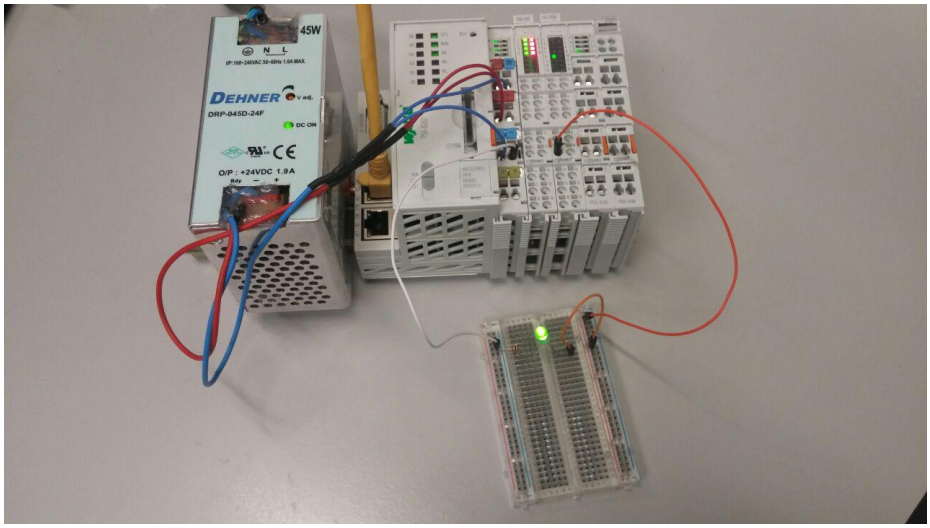

Fig. 6. Wago 750-8202 Test Platform running Codesys runtime (e!Cockpit variant)

pin is connected to a button. The electrical current becomes disconnected when the button is pressed and is reconnected when the button is released. From the perspective of the runtime, if no one is pressing the button, the input pin has a value of TRUE, whereas if someone is pressing the button, the pin has a value of FALSE.

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

The logic and the physical process for the Wago 750-8202 PLC

We use pins 0 of the Wago PLC to control our physical process. In our logic, we declare pin 0 of the 750-1506 digital I/O module as the output pin. In the physical layout, our output pin is connected to an LED.

From the perspective of the runtime, the runtime change the state of the pin 0 TRUE, for 100 times of PLC scan cycle and switch it back to FALSE.

The described logic is illustrated in Figure 7. According to our logic, the LED turns on or off every 1 seconds (our Scan cycle).

Interaction between the virtual I/O registers and the runtime To understand how the Codesys runtime interacts with the pins, we briefly describe how virtual I/O registers operate in a BCM2835 processor (the one used in the Raspberry Pi 1).

input : State of $In.24$
output: State of $Out.22$

Main Logic;
while *True* **do**
  read input;
  while *input True* **do**
    switch_state(output, five seconds);
    //states are High or Low.
  end
  if *input False* **then**
    hold the state of the output;
  else
    go to first while;
  end
end

Algorithm 1: Logic for our representative physical process in a —Raspberry Pi 1

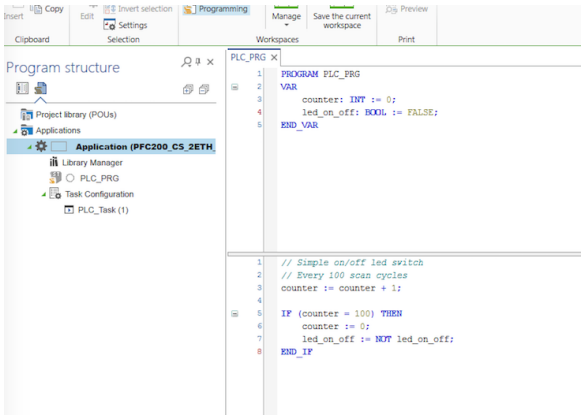

Fig. 7. Logic for our representative physical process for Wago 750-8202 Test Platform

The virtual I/O registers are simply I/O address ranges in the processor memory that perform different types of I/O operations. The operation types, the memory sizes, and the physical addresses of the three virtual I/O registers related to read, write, and I/O configuration operations in the BCM2835 processor are summarized in Table 1.

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

1. Input/Output Mode register: this register sets pins to input or output mode. Each pin in this register has three bits of space. The first bit of each pin's bit space defines whether the pin is used for input or output. If the first bit for a pin is set to 0, the pin is an input pin. If the first bit is set to 1, the pin is an output pin. The Codesys runtime cannot change the value of a pin in

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

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

input mode; it can only read from it. Even if Codesys writes a value to an input-mode-enabled pin, the operation has no physical effect and is ignored by the processor. However, the Codesys runtime can change the value of a pin when the pin is in output mode. The input/output mode address range begins at offset 0x2020000 and ends at address 0x20200002. For pin 22, bits 6 to 8 of address 0x20200002 are used.

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

We illustrate how the Codesys runtime interacts with virtual I/O registers by describing a write operation for pin 22 (see Figure 8). Read operations follow a similar procedure. During the startup of the Codesys runtime, the OS maps the addresses of the I/Os into the Codesys Thread Local Storage (TLS). Once the I/Os are mapped, the mapped I/O addresses are recorded in the page table and in a cache that is called the Translation Lookaside Buffer (TLB). The page table is a structure in which the OS maintains the list of mapped physical addresses and their corresponding virtual addresses for the process. The page table can become so large that searching it can be time consuming for the OS. To avoid this scenario, the OS also uses a cache for the page table, the TLB.

After our logic is uploaded to the Raspberry Pi, the Codesys runtime evaluates the I/O configuration specified in the logic to determine which pins are designated as input or output. Pins 22 and 24 are designated as output and input respectively. Because pin 22 is declared as an output pin in our logic, Codesys

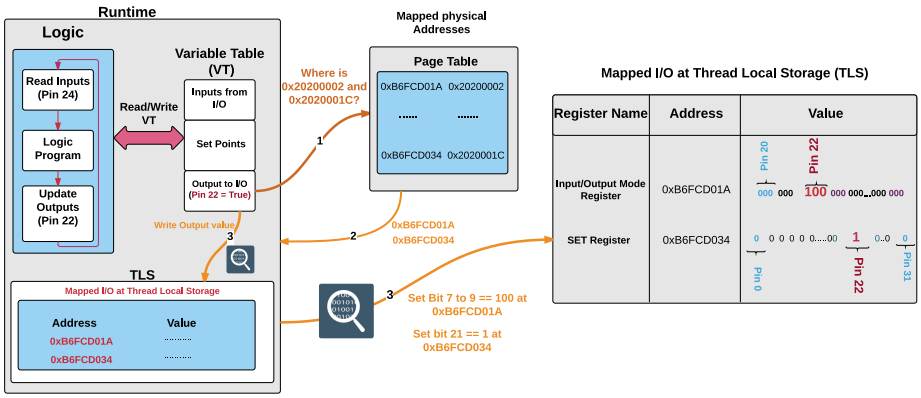

Fig. 8. A search for the address of pin 22 and a write operation to it

performs a write operation in the Input/Output Mode register and set bits 6 to 8 of offset 0x20200002 to 100. To execute this write operation, the Codesys runtime asks for the virtual address of 0x20200002. The OS looks up the virtual address of 0x20200002 (mapped address of 0x20200002) in the TLB and page table. In our environment, this address was located as expected in the TLS memory area of the Codesys runtime with value 0xB6FCD01A. Once Codesys has retrieved the virtual address, it writes to it and continues operations. Codesys then assumes that the I/O configuration sequence was successful and that the I/O pins are ready to use.

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

7.3 Attack implementation with root access

To be able to accurately tamper the control flow set in the logic we must be able to intercept each read and write operation of the Codesys runtime. However, if we were to use conventional function hooking techniques (e.g hooking the Codesys functions responsible for reporting the I/O status) or modifying the integrity of the code in the PLC (e.g. modifying the codes of the runtime or kernel data structure), most control flow and code integrity security mechanisms would be able to detect and block our attempts. Therefore, we leverage

the processor debug registers for interception. Debug registers were introduced to assist developers in analyzing their software, and all new processors with various different architectures (ARM, Intel, and MIPS) have such registers. These registers allow setting hardware breakpoints to specific memory addresses. Once an address that is in a debug register is accessed by a process, the processor interrupt handler is called and customized code can be executed.

Unfortunately, for PLCs running a modern OS with a memory management unit, setting debug registers requires root access. This can be achieved either by leveraging default passwords, through a control-flow attack against the PLC runtime, or through a firmware modification attack [4, 7, 8, 5]. Regarding default passwords, several reported vulnerabilities suggest that some PLCs provide shell access with default root passwords [48, 49]. An attacker could just log in to these devices using the default password and execute the attack. Using a control-flow attack, the attacker can gain access at the same level as the PLC process. As described earlier, previous research has revealed that various PLCs run their runtime as the root user by default [11, 44]. In case that the PLC runtime is vulnerable to control-flow attack but is not running as root, the attacker needs a privilege escalation vulnerability to gain root access to the PLC. Finally, by installing rogue firmware into the PLC, the attacker can infect every binary in the PLC. This can give the attacker complete leverage over the PLC operating system.

We also assume that the attacker knows the physical process (by the mean of reading the logic exist in the PLC) and is aware of the mapping between the I/O pins and the logic. The PLC logic might use various inputs and outputs to control its process; thus, the attacker must know which input or output must be modified to affect the process as desired. The work presented by McLaughlin et al. [3, 2] can be used to discover the mapping between the different I/O variables and the physical world.

As the first stage of our attack, we set the mapped I/O addresses to the debug register and intercept every write or read operation of the Codesys runtime. When the PLC runtime wants to read from or write to the I/O pins, the processor halts the process and calls the attacker hardware-based interrupt handler. The handler performs the I/O manipulation by exploiting the pin configuration functionality as discussed earlier. Figure 9 depicts this process.

For our experiment, we implemented the attack in an LKM (Loadable Kernel Module). LKMs have access to kernel space, even if this is not a requirement. Once the module is loaded, it checks the CPU information of the machine and matches it against a hard-coded list of CPUs and their I/O memory ranges. As mentioned above, the I/O addresses of the BCM2835 processor begin at 0x20200000. Using this information, the LKM looks in the OS page table for a process ID and a virtual address that are mapped to the physical address 0x20200000. Because target pins are known in advance (in our case, pin 22), the correct register address for the SET, READ, and Input/Output Mode registers can be easily calculated.

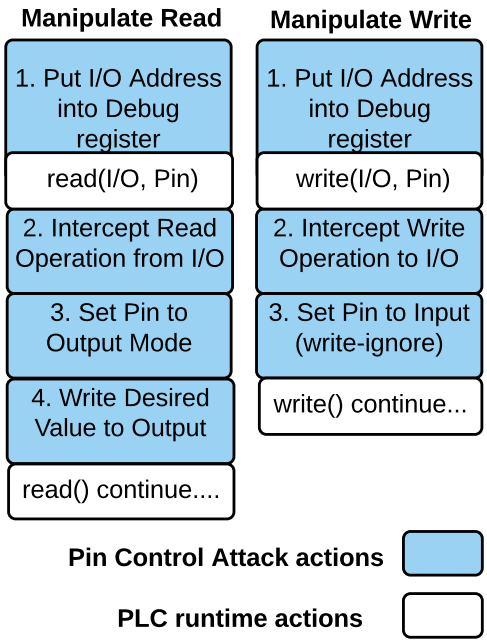

Fig. 9. Steps of the Pin Control for read and write manipulation

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

To manipulate read operstions on pin 24 the rootkit inserts the virtual address from the READ register associated with pin 24 into the debug register and installs its custom exception handler. Once Codesys accesses the virtual address from the READ register to read the value from the pin, the exception handler executes. The exception handler first sets pin 24 as an output pin by writing the values "100" to the related bits of the Input/Output Mode register and then writes the desired value for pin 24 into the SET register. The LKM then returns

control to the Codesys runtime, which will read the value written in the I/O register by the exception handler instead of the real value.

With this technique we were able to successfully alter the physical process described in Section 7.2. We modified the process by allowing the PLC to turn on and off the LED on pin 22 only every ten seconds instead of every five. Additionally, the read manipulation made the button in our physical layout ineffective. We could hold the last state of the LED by giving fake read input values to the runtime and make the runtime assume that someone pressed the electrical button while it was not the case.

Embedded devices typically have limited resources for the operations they execute. This is the case for PLCs as well. While in general performance overhead is not an issue for the attacker, it can be when a PLC controls processes that are time critical. If in such processes the performance overhead causes significant delay in the I/O speed, it can uncover the attack. For this reason we evaluated the performance overhead imposed by this attack on our selected hardware (Raspberry Pi model 1 B). Regarding CPU overhead, based on our evaluation, n rootkit based pin control attack on average incurs 5% CPU overhead for the manipulation of write operations and 23% CPU overhead for the manipulation of read operations. Read operation manipulation imposes a higher CPU load for two reasons. First, the PLC runtime environment reads the values from the I/O multiple times per second, thereby significantly increasing the CPU overhead, whereas for write operations, the number of I/O write operations depends only on the logic (in our case, every five seconds). Second, read manipulation requires two instructions (setting the pin to output mode and writing to it), whereas write manipulation requires only one instruction (setting the pin to input mode). Figure 10 depicts the CPU overhead incurred by the manipulation of read and write operations in a rootkit based pin control tack. The additional CPU overhead is not an important concern for the attacker, but it creates anomalies in the power consumption of the victimized device.

To understand the impact of an rootkit based pin control attack on control operations, we evaluated the I/O speed fluctuations in our selected setup (Raspberry Pi with Codesys runtime running our sample logic). Figure 11 depicts the fluctuation of the I/O speed with and without our rootkit implementation. On average the speed where our hardware could write to the I/O (without our rootkit) was 3.97 milliseconds. When the rootkit manipulates the I/O (intercept the I/O write operation and write the same value), the average speed of the I/O increased to 4.01 milliseconds.

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

7.4 Attack implementation without root access

The previous implementation of the attack allows to precisely tamper the control flow set in the logic. However such precision comes at the cost of the high privilege requirements and the non-negligible performance overhead due to the usage of

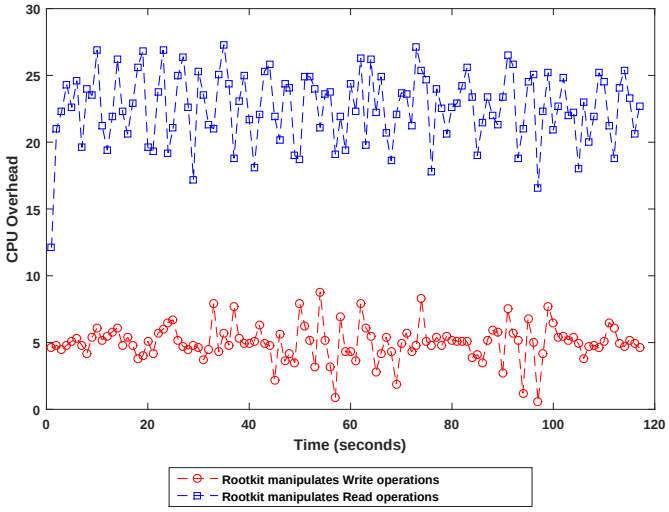

Fig. 10. CPU overhead in rootkit based Pin Control attack

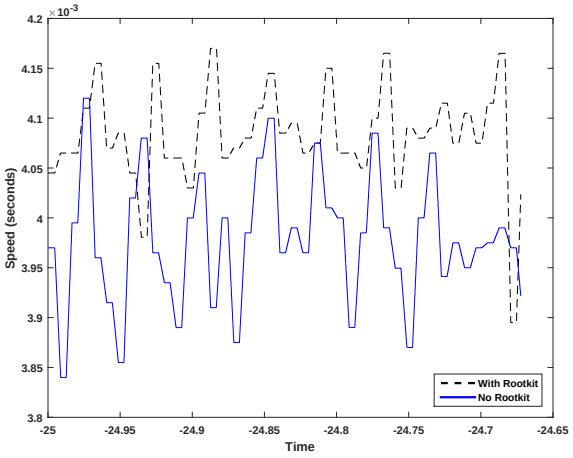

Fig. 11. I/O speed with and without rootkit

debug registers (which causes the call of hardware interrupt handlers of the SoC). By relaxing a bit our precise I/O modification requirement, we can create a second implementation of the attack which has better performance and does not require root/kernel access, while still being able to manipulate the physical process as desired.

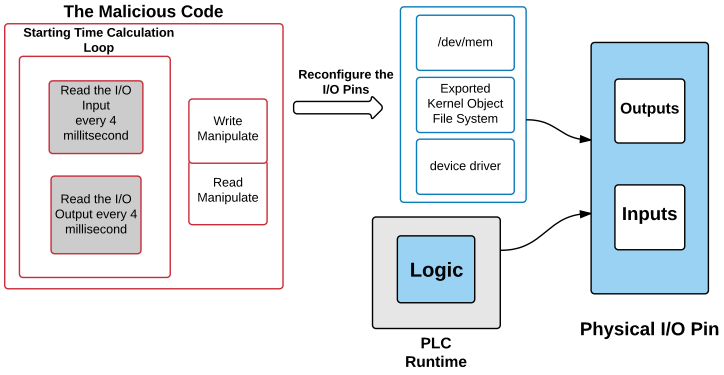

Fig. 12. Overview of pin configuration attack without root access using exported kernel objects, /dev/mem and a legitimate device driver call

The requirement for this second implementation is that the attacker has the same access privileges as the PLC runtime. This is achievable for example by exploiting a memory corruption vulnerability that allows code execution, such as a buffer overflow [43, 13, 11, 49]. A remote code execution vulnerability of such kind is known to be affecting the Codesys runtime [50]. Also, similarly to our previous implementation, we assume that our application already has access to the logic (since the logic will be inside the PLC) and knows the I/O mapping of the process.

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

Figure 12 depicts the steps our application takes to execute the attack. The application first checks whether the processor I/O configuration addresses are mapped in the PLC runtime. The list of all mapped addresses is system wide available in various locations for any user space application (e.g. via /proc/modules, or /proc/$pid/maps). If at any extraordinary circumstance the physical I/O addresses are not mapped, our application can map the I/O base address of our target platform at 0x20200000.

For manipulating write operations, the application needs to know a reference starting time. This is the relative time where the PLC runtime writes to the pin. While the application knows the logic and is aware that every five seconds there is a write operation to pin 22, it does not know at what second the last write operation happened. This can be easily found by monitoring the value of pin 22. Once the application intercepts the correct reference starting time, for every write operation in the logic it will carry out two tasks. First, right before the reference starting time (which is when the PLC runtime will start writing

its desired original value to the I/O) the application reconfigures the pin to input mode. The Codesys runtime then attempts to write to the pin. However, the write operation will be ineffective, since the pin mode is set to input. Our application then switches the pin mode to output and writes the desired value to it.

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

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

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

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

8 Discussions

As discussed in Section 6, the I/O attack cannot be detected by Autoscopy Jr. or Doppelganger. We verified this claim by testing the I/O attack against the current implementation of Autoscopy Jr. Unfortunately, the authors of Doppelganger were unable to provide us with their implementation for our test. Autoscopy Jr. does not detect this attack because no data hooking is performed in an I/O attack. We argue that Doppelganger is also unable to detect I/O attack because no modification of the static parts of the memory (e.g., code hooking) is performed in such an attack; instead, as described in Section 7, the entirety of the malicious code is loaded into the dynamic memory, which is not monitored by Doppelganger.

There are several characteristics of pin control attack that should be considered regarding their practical implementation. Therefore, in this section, we first discuss about why it is hard to detect pin control attack and then consider the hardware knowledge required in the rootkit version of the attack.

8.1 Detection of Pin Control Attack

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

1. *Monitoring the mapping of pin configuration registers*: an attacker needs to use the virtual addresses of the pin configuration registers to write to them. To do so, the attacker needs to either map the physical registers by herself or use already mapped addresses. In the first case, one could monitor the mapping of pin configuration registers by hooking critical mapping functions such as mmap() or ioctl(). However, an attacker could implement her own version of mmap() or ioctl(), thus bypassing the monitoring point. Additionally, in an embedded device that is already using the target I/O, an attacker can use the pin register address already mapped by the application or kernel.

2. *Monitoring the change of pin configuration registers*: one may detect our attack by monitoring the frequency at which pin configuration registers are changed. This may be challenging for two reasons. First, since changes in configuration registers do not generate interrupts, an attacker could be able to bypass monitoring mechanisms. For example, in order to avoid performance overhead, the value of configuration changes which must be checked in a loop, could be monitored with a certain frequency (e.g., every second). This would give the attacker a window of opportunity to modify the configuration within the checking window. Second, since pins get re-configured legitimately, it may be difficult to tell with reliable accuracy whether a sequence of of changes is legitimate or not.

3. *Monitoring the use of debug registers*: one could argue that the usage of debug registers in our first implementation of the attack is something that can be easily detected. For example, by monitoring the processor debug registers one could detect the point in our first attack implementation in which we set a breakpoint to the access of the I/O registers. However, in our experiments we noticed that constantly monitoring all processor debug registers (i.e., with a loop) can cause a non-negligible CPU overhead. This makes this approach not very attractive for embedded systems, which are always resource constrained.

4. *Monitoring performance overhead*: because of the performance overhead imposed by our first implementation of the attack, we could employ approaches based on monitoring the power consumption of embedded systems, such as those proposed in [51] to detect the attack. However, how our second implementation demonstrates, other approaches can impose a significantly smaller CPU overhead, which would probably go unnoticed. In addition, other pin control attacks presented in Section 6 just require a single configuration or multiplexing operation, with practically no CPU overhead.

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

8.2 Hardware Knowledge

In Section 7, we used the physical addresses of the I/O pins to find their mapped virtual addresses. Moreover, in our rootkit implementation, we had knowledge of all physical I/O register addresses. However, this is not the case for all types of processors. For example, certain PLC processors are proprietary. In this case, an attacker needs to perform the additional step of determining the physical addresses of the I/O pins of his interest. However, this necessity does not stop state-sponsored attackers. Detecting the I/O addresses that are used in either drivers or applications is straightforward. Unix-based operating systems provide I/O address ranges in /proc/modules for kernel drivers or in /proc/$pid/maps (where $pid is the PLC runtime process ID) for applications for I/O mapping. Nevertheless, detecting the I/O register addresses is a complicated task. Again, attackers who wish to target PLCs to attack critical infrastructures will investigate their targets sufficiently to determine this information. One solution for obtaining this I/O register information is to first decompile the available PLC logic within the PLC memory and search for I/O read/write operations and then monitor the read/write operations involving the mapped addresses retrieved from the OS (e.g., /proc/modules or /proc/$pid/maps). An attacker can begin looking for the I/O input/output mode registers by monitoring the PLC runtime environment when it is starting up. Additionally, from the decompiled logic, the attacker can be aware of the timing of the cycle of read and write operations in a specified I/O memory range. By monitoring read/write operations in that memory area (e.g., using debug registers), the attacker can identify the I/O read/write registers.

9 Related Work

Various research works have addressed attacks against embedded systems. For example, a significant stream of work has explored the manipulation of embedded systems' firmware [4, 5, 7, 8]. Another relevant stream of work, instead, has explored memory corruption vulnerabilities in embedded systems [11, 13, 52]. However, to the best of our knowledge, we could not find other approaches discussing the security implications of the pin control system in embedded devices.

Part of our work bears some similarities with System Management Mode (SMM) rootkits [53–55] for X86 architectures. These rootkits tap the system I/O, similarly to what we did in our Pin Configuration attack. However, while the goals is similar, the way to reach the goal is different. For example, the modification of system I/O in SMM causes interrupts which need to be suppressed by SMM rootkits, typically by attacking kernel interrupt handlers. In our case, this operation is not needed due to the lack of interrupts for pin multiplexing and configuration.

Among the works focusing on protecting embedded systems, a stream of research focuses on firmware verification [23–25]. Another stream of research focuses on detecting kernel level attacks by monitoring syscall/function hooking techniques and kernel data structure manipulation [56–58, 29, 30].

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

10 Conclusions and Future Work

In this paper, we first looked into the current state of host-based detection techniques for embedded devices, with a particular focus on Programmable Logic Controllers. We found that current practical host-based intrusion detection techniques for embedded devices suffer from three major shortcomings. First, they completely ignore the control of dynamic memory when verifying memory contents. Second, they do not apply effective practical control-flow measures due to performance limitations. Finally, they mostly rely on static references to protect embedded devices. In the second part, we have proposed a new type of attack that leverages these weaknesses, and we have shown that it can be used by adversaries to manipulate the physical process in a way that the PLC runtime and the SCADA applications are unaware of the manipulation. This makes the attack interesting and relevant since current detection techniques are not effective against this new type of attack (Pin Control Attack) or any type of attack that exploits the weaknesses discussed in the Section 4.

We now plan to investigate in more detail the opportunities offered by the monitoring techniques we briefly sketched in Section 8.1. In particular, we will focus on the protection of dynamic memory and improving control-flow integrity in embedded devices. We believe that in any attack against embedded devices, control-flow integrity and dynamic memory verification measures will pose a notable hindrance to attackers, significantly reducing their success rate.

Acknowledgement We are immensely grateful to Prof. Sandro Etalle (Einhoven University of Technology), Dr. Emmanuele Zambon (Security Matters B.V), Prof. Thorsten Holz (Ruhr University Bochum), Marina Krotofil (Honeywell International, Inc.) and Jafar Haadi Jafarian (University of North Carolina at Charlotte) for their insights on the manuscript. We would also like to show our gratitude to the Linus Walleij, lead developer of the Linux kernel Pin Control Subsystem for discussion on the attack against PLCs and sharing his pearls of wisdom with us.

References

1. P. Koopman, "Embedded system security," *Computer*, vol. 37, no. 7, pp. 95–97, 2004.
2. S. McLaughlin and P. McDaniel, "SABOT: Specification-based payload generation for programmable logic controllers," in *Proceedings of the 2012 ACM Conference on Computer and Communications Security*, ser. CCS '12. New York, NY, USA: ACM, 2012, pp. 439–449.
3. S. E. McLaughlin, "On dynamic malware payloads aimed at programmable logic controllers," in *HotSec*, 2011.

4. A. Cui, M. Costello, and S. J. Stolfo, "When firmware modifications attack: A case study of embedded exploitation," in *NDSS*, 2013.
5. P. Traynor, K. Butler, W. Enck, P. McDaniel, and K. Borders, "Malnets: Large-scale malicious networks via compromised wireless access points," *Security and Communication Networks*, vol. 3, no. 2-3, pp. 102–113, 2010.
6. S. Wegner, "Security-analysis of a telephone-firmware with focus on backdoors," Bachelor's thesis, Ruhr-Universität Bochum, 2008. [Online]. Available: https://git.fabrik17.de/mrgitlab/embedded-multimedia/raw/437afd92da4b438f95fa3efad28564a9d0baffbd/Dokumentation/_thesis_template.pdf
7. Z. Basnight, J. Butts, J. L. Jr., and T. Dube, "Firmware modification attacks on programmable logic controllers," *International Journal of Critical Infrastructure Protection*, vol. 6, no. 2, pp. 76 – 84, 2013.
8. D. Peck and D. Peterson, "Leveraging ethernet card vulnerabilities in field devices," in *SCADA Security Scientific Symposium*, 2009, pp. 1–19.
9. pt, "Oops, I hacked my PBX. Why auditing proprietary protocols matters," *28th Chaos Communication Congress*, 2011.
10. N. Falliere, L. O. Murchu, and E. Chien, "W32. stuxnet dossier," *White paper, Symantec Corp., Security Response*, vol. 5, 2011.
11. D. Beresford, "Exploiting Siemens Simatic S7 PLCs," in *Black Hat USA*, 2011.
12. ICS-CERT, "Schneider electric modicon quantum vulnerabilities (update b)," 2014. [Online]. Available: https://ics-cert.us-cert.gov/alerts/ICS-ALERT-12-020-03B
13. R. Wightman, "Project basecamp at s4," *SCADA Security Scientific Symposium*, 2012. [Online]. Available: https://www.digitalbond.com/tools/basecamp/schneider-modicon-quantum/
14. Rapid7, "Linksys wrt120n tmunblock stack buffer overflow," 2014. [Online]. Available: http://www.rapid7.com/db/modules/auxiliary/admin/http/linksys_tmunblock_admin_reset_bof
15. ——, "D-link hnap request remote buffer overflow," 2014. [Online]. Available: http://www.rapid7.com/db/modules/exploit/linux/http/dlink_hnap_bof
16. O. S. V. D. (OSVDB), "D-link dir-605l wireless n300 cloud router captcha data http request parsing remote buffer overflow," 2012. [Online]. Available: http://www.osvdb.org/86824
17. F. Schuster, T. Tendyck, J. Pewny, A. Maaß, M. Steegmanns, M. Contag, and T. Holz, "Evaluating the effectiveness of current anti-ROP defenses," in *Research in Attacks, Intrusions and Defenses*, A. Stavrou, H. Bos, and G. Portokalidis, Eds. Springer, 2014, pp. 88–108.
18. V. Pappas, "kBouncer: Efficient and transparent ROP mitigation," 2012. [Online]. Available: http://www.cs.columbia.edu/~vpappas/papers/kbouncer.pdf
19. Y. Cheng, Z. Zhou, M. Yu, X. Ding, and R. H. Deng, "ROPecker: A generic and practical approach for defending against ROP attacks," in *Symposium on Network and Distributed System Security (NDSS)*, 2014.
20. I. Fratrić, "ROPGuard: Runtime prevention of return-oriented programming attacks," 2012. [Online]. Available: http://www.ieee.hr/_download/repository/Ivan_Fratric.pdf
21. L. Davi, D. Lehmann, A.-R. Sadeghi, and F. Monrose, "Stitching the gadgets: On the ineffectiveness of coarse-grained control-flow integrity protection," in *USENIX Security Symposium*, 2014.
22. Microsoft Corporation, "Enhanced mitigation experience toolkit," 2014. [Online]. Available: https://www.microsoft.com/emet

23. F. Adelstein, M. Stillerman, and D. Kozen, "Malicious code detection for open firmware," in *18th Annual Computer Security Applications Conference, 2002. Proceedings*, 2002, pp. 403–412.
24. F. Zhang, H. Wang, K. Leach, and A. Stavrou, "A framework to secure peripherals at runtime," in *Computer Security-ESORICS 2014*, M. Kutyłowski and J. Vaidya, Eds. Springer, 2014, pp. 219–238.
25. L. Duflot, Y.-A. Perez, and B. Morin, "What if you cant trust your network card?" in *Recent Advances in Intrusion Detection*. Springer, 2011, pp. 378–397.
26. A. Seshadri, A. Perrig, L. van Doorn, and P. Khosla, "SWATT: SoftWare-based attestation for embedded devices," in *2004 IEEE Symposium on Security and Privacy. Proceedings*, May 2004, pp. 272–282.
27. M. LeMay and C. Gunter, "Cumulative attestation kernels for embedded systems," *IEEE Transactions on Smart Grid*, vol. 3, no. 2, pp. 744–760, June 2012.
28. F. Armknecht, A.-R. Sadeghi, S. Schulz, and C. Wachsmann, "A security framework for the analysis and design of software attestation," in *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, ser. CCS '13, S. Merz and J. Pang, Eds. New York, NY, USA: ACM, 2013, pp. 1–12. [Online]. Available: http://doi.acm.org/10.1145/2508859.2516650
29. A. Cui and S. J. Stolfo, "Defending embedded systems with software symbiotes," in *Recent Advances in Intrusion Detectio: 14th International Symposiumn*, R. Sommer, D. Balzarotti, and G. Maier, Eds. Springer, 2011, pp. 358–377.
30. J. Reeves, A. Ramaswamy, M. Locasto, S. Bratus, and S. Smith, "Intrusion detection for resource-constrained embedded control systems in the power grid," *International Journal of Critical Infrastructure Protection*, vol. 5, no. 2, pp. 74–83, 2012.
31. F. Abad, J. van der Woude, Y. Lu, S. Bak, M. Caccamo, L. Sha, R. Mancuso, and S. Mohan, "On-chip control flow integrity check for real time embedded systems," in *2013 IEEE 1st International Conference on Cyber-Physical Systems, Networks, and Applications (CPSNA)*, Aug 2013, pp. 26–31.
32. A. Francillon, D. Perito, and C. Castelluccia, "Defending embedded systems against control flow attacks," in *Proceedings of the First ACM Workshop on Secure Execution of Untrusted Code*, ser. SecuCode '09. New York, NY, USA: ACM, 2009, pp. 19–26.
33. L. Davi, P. Koeberl, and A.-R. Sadeghi, "Hardware-assisted fine-grained control-flow integrity: Towards efficient protection of embedded systems against software exploitation," in *Proceedings of the 51st Annual Design Automation Conference*, ser. DAC '14. New York, NY, USA: ACM, 2014, pp. 133:1–133:6.
34. A. Cui, "Red ballon security." [Online]. Available: http://www.redballoonsecurity.com
35. Z. Liang, H. Yin, and D. Song, "HookFinder: Identifying and understanding malware hooking behaviors," in *Proceeding of the 15th Annual Network and Distributed System Security Symposium (NDSS'08)*, 2008. [Online]. Available: http://bitblaze.cs.berkeley.edu/papers/hookfinder_ndss08.pdf
36. S. Vogl, R. Gawlik, B. Garmany, T. Kittel, J. Pfoh, C. Eckert, and T. Holz, "Dynamic hooks: hiding control flow changes within non-control data," in *Proceedings of the 23rd USENIX conference on Security Symposium*. USENIX Association, 2014, pp. 813–828.
37. Kernel.org, "Pin control subsystem in linux." [Online]. Available: https://www.kernel.org/doc/Documentation/pinctrl.txt

38. P. Ghosh, P. S. Hira, and S. Garg, "A method to make soc verification independent of pin multiplexing change," in *Computer Communication and Informatics (ICCCI), 2013 International Conference on*. IEEE, 2013, pp. 1–6.
39. L. Walleij, "Pin control subsystem overview," in *Embedded Linux Conference*, 2012.
40. R. Spenneberg, M. Brüggemann, and H. Schwartke, "Plc-blaster: A worm living solely in the plc," *Black Hat Asia*, 2016.
41. ICS-CERT, "Abb ac500 plc webserver codesys vulnerability," 2013. [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-12-320-01
42. ——, "Schneider electric modicon m340 buffer overflow vulnerability," 2015. [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-15-351-01
43. ——, "Rockwell automation micrologix 1100 plc overflow vulnerability," 2016. [Online]. Available: https://ics-cert.us-cert.gov/advisories/ICSA-16-026-02
44. DigitalBond, "3S CoDeSys, Project Basecamp," 2012. [Online]. Available: http://www.digitalbond.com/tools/basecamp/3s-codesys/
45. R. Langner, "To kill a centrifuge: A technical analysis of what stuxnets creators tried to achieve," *Online: http://www. langner. com/en/wp-content/uploads/2013/11/To-kill-a-centrifuge. pdf*, 2013.
46. V. M. Igure, S. A. Laughter, and R. D. Williams, "Security issues in SCADA networks," *Computers & Security*, vol. 25, no. 7, pp. 498–506, 2006.
47. A. Baliga, V. Ganapathy, and L. Iftode, "Detecting kernel-level rootkits using data structure invariants," *Dependable and Secure Computing, IEEE Transactions on*, vol. 8, no. 5, pp. 670–684, Sept 2011.
48. DigitalBond, "WAGO IPC 758/870, Project Basecamp," 2015. [Online]. Available: http://www.digitalbond.com/tools/basecamp/wago-ipc-758870/
49. D. Beresford and A. Abbasi, "Project IRUS: multifaceted approach to attacking and defending ICS," in *SCADA Security Scientific Symposium(S4)*, 2013.
50. K. R. Wrightman, "Vulnerability Inheritance in PLCs," 2015.
51. C. A. Gonzalez and A. Hinton, "Detecting malicious software execution in programmable logic controllers using power fingerprinting," in *Critical Infrastructure Protection VIII*. Springer, 2014, pp. 15–27.
52. L. Davi, A. Dmitrienko, A.-R. Sadeghi, and M. Winandy, "Return-oriented programming without returns on arm," Technical Report HGI-TR-2010-002, Ruhr-University Bochum, Tech. Rep., 2010.
53. S. Sparks, S. Embleton, and C. C. Zou, "A chipset level network backdoor: bypassing host-based firewall & ids," in *Proceedings of the 4th International Symposium on Information, Computer, and Communications Security*. ACM, 2009, pp. 125–134.
54. S. Embleton, S. Sparks, and C. C. Zou, "Smm rootkit: a new breed of os independent malware," *Security and Communication Networks*, vol. 6, no. 12, pp. 1590–1605, 2013.
55. J. Schiffman and D. Kaplan, "The smm rootkit revisited: Fun with usb," in *Availability, Reliability and Security (ARES), 2014 Ninth International Conference on*, Sept 2014, pp. 279–286.
56. Z. Wang, X. Jiang, W. Cui, and P. Ning, "Countering kernel rootkits with lightweight hook protection," in *Proceedings of the 16th ACM Conference on Computer and Communications Security*, ser. CCS '09. New York, NY, USA: ACM, 2009, pp. 545–554.
57. V. Kuznetsov, L. Szekeres, M. Payer, G. Candea, R. Sekar, and D. Song, "Code-pointer integrity," in *USENIX Symposium on Operating Systems Design and Implementation (OSDI)*, 2014.

58. Y. Kinebuchi, S. Butt, V. Ganapathy, L. Iftode, and T. Nakajima, "Monitoring integrity using limited local memory," *Information Forensics and Security, IEEE Transactions on*, vol. 8, no. 7, pp. 1230–1242, July 2013.