Co-Guard: Guarding Safety-Critical Embedded Devices in Emergencies

Christian Niesler[0000−0002−8589−5231], Christian Scholz[0009−0003−2601−2202], Nils Hannappel[0009−0002−9764−8200], and Lucas Davi[0000−0002−7322−2777]

paluno, University of Duisburg-Essen
Universitätsstraße 2, 45141 Essen, Germany
{christian.niesler, christian.scholz, nils.hannappel, lucas.davi}@uni-due.de

Abstract. Malware launched on safety-critical systems is an increasingly utilized attack strategy. Most security measures focus on stopping attacks, but few consider what happens after a system is compromised. These defenses often depend on extra monitoring devices and stable networks, which may not be available during emergencies. To address this issue, we developed *Co-Guard*, a local safety system that uses the microcontroller's coprocessor. *Co-Guard* can stop physical damage by rebooting devices to a safe state without requiring network connectivity. The coprocessor is part of the device and can monitor sensor data to detect tampering. We evaluate *Co-Guard* with two case studies: a water purification system modeled after the Israeli water infrastructure attack and a smoke detector. These studies show that our approach is practical and effective in protecting safety-critical devices from manipulation.

Keywords: Co-Processor · Safety & Security · Embedded Systems · Industrial Control Systems

1 Introduction

In recent years, cyberattacks have escalated from primarily targeting data confidentiality to directly threatening human safety and critical infrastructure. High-profile incidents such as the Triton [31] malware attack against a Saudi petrochemical facility and intrusions into water treatment [28] and dam control systems [16] illustrate how adversaries can manipulate industrial control logic to cause physical harm or large-scale disruption. These cases underscore that protecting safety-critical infrastructure is no longer just an operational concern but a central security challenge.

Safety-critical systems, typically modeled as cyber-physical systems (CPS), control essential physical processes ranging from industrial robotics and power generation to water treatment and building safety. Increasingly, smaller networked devices—such as connected vehicles, drones, and smart sensors form integral parts of these infrastructures. Although such systems are designed with rigorous safety engineering, they remain exposed to security vulnerabilities. As Lutz et al. [40]

observed, failures in safety-critical software often originate from programming mistakes, human error, or flawed processes. Today, external attackers can exploit the same weaknesses to induce dangerous behavior even when no immediate design error is apparent.

Traditional defenses like control-flow integrity (CFI) and data-flow integrity (DFI) are difficult to deploy on embedded platforms that must meet strict real-time constraints and operate with limited memory and CPU resources. Recent work, including Kage [20], Silhouette [68], and other related protections [48, 65], leverages hardware features to strengthen resistance against exploitation on common architectures. However, these mechanisms primarily focus on preventing initial compromise. Once an attacker has gained control, there is often little support for constraining or recovering from malicious actions, such as gradually altering chemical concentrations in a water plant or suppressing alarms in safety systems [28, 42].

Complementary efforts such as *IoTMon* [19], *IoTGuard* [14], and *SCADMAN* [1] monitor device behavior and process integrity from the network side, typically using centralized analysis backends. While effective in well-connected environments, these frameworks assume reliable communication channels and continuous connectivity to monitoring infrastructure. In practice, harsh environmental conditions, targeted denial-of-service attacks, or large-scale disasters can disrupt networks [22, 67], isolating devices precisely when safety guarantees are most needed. Under such circumstances, safety-critical nodes must remain capable of autonomously detecting and mitigating dangerous states, without relying on external oversight.

To address these limitations, this work introduces *Co-Guard*, a security architecture that treats physical safety as a first-class security objective for embedded systems operating under unstable network conditions. *Co-Guard* leverages existing on-chip hardware—such as auxiliary coprocessors available on many microcontrollers—to monitor safety-relevant signals and invariants in real time, independently of the main application CPU. By offloading monitoring and decision logic to this dedicated hardware, *Co-Guard* can continuously track process variables (for instance, chemical concentrations or sensor readings) without introducing prohibitive overhead into the main control loop. When a violation of predefined safety thresholds is detected, *Co-Guard* can force the system into a safe state, for example, by triggering an emergency shutdown or rebooting into a known-good firmware configuration, thereby limiting the attacker's ability to cause physical harm.

Contributions.
In summary, this paper makes the following contributions:

- It presents *Co-Guard*, a framework that enhances the safety of embedded systems in unstable or degraded environments by leveraging off-the-shelf hardware components, such as coprocessors, for independent safety monitoring.
- *Co-Guard* integrates with existing firmware and RTOS-based systems while imposing low runtime and memory overhead, making it suitable for resource-constrained, hard real-time devices.

- By monitoring physical device behavior and safety invariants rather than solely digital state, *Co-Guard* can detect security incidents that manifest as unsafe physical conditions and intervene before they lead to damage.
- The framework supports timely emergency responses, including controlled shutdowns and reboots into non-compromised configurations, thereby preserving the safety and integrity of critical operations.
- *Co-Guard* is inherently scalable: each device can perform autonomous safety monitoring on its local coprocessor, reducing dependence on centralized infrastructure.
- The design is instantiated on widely deployed platforms, including the ESP32 microcontroller family, where *Co-Guard* runs on the built-in coprocessors used in many IoT deployments, and on constrained coprocessor-like units such as the PIO blocks of the RP2040, demonstrating applicability even to minimal hardware offload engines.
- A proof-of-concept evaluation on a water purification system and a smoke detector shows that *Co-Guard* can detect and counteract real attacks: after compromising and manipulating the devices to allow harmful chemical concentrations or to ignore fire conditions, *Co-Guard* identifies the resulting safety violations and reboots the systems into a clean state before substantial physical harm occurs.

2 Background

To ensure the safety of embedded devices, it is vital to understand how these systems operate and interact with their surrounding environment. Embedded devices are typically designed for specific, well-defined functions and operate within constrained hardware and software environments. They integrate sensing, computation, and communication capabilities to process information in real time and respond to environmental changes. This section outlines the key components and mechanisms of embedded systems, forming the basis for later discussions on enhancing security through safety-oriented design principles.

Sensor Measurements. Sensors transform continuous physical signals into digital data that microcontrollers can process. This conversion is handled by Analog-to-Digital Converters (ADCs), which translate analog signals into binary representations. In many embedded systems, agent-based architectures [55] are employed to monitor the environment and dynamically adjust performance parameters in response to changing conditions.

General Purpose Input/Output (GPIO). Microcontrollers use General Purpose Input/Output (GPIO) pins [8] to interface with sensors, actuators, and other external hardware components. These pins can be configured as inputs or outputs and frequently support communication protocols such as UART [37] or I2C [41]. Through GPIO interfaces, embedded systems enable flexible, efficient data exchange with peripheral devices.

Communication Protocols. Protocols such as I2C [41], UART [37], and SPI [2] are fundamental to ensuring reliable data communication among microcontrollers,

sensors, and actuators. Each protocol offers distinct advantages—such as trade-offs in speed, wiring complexity, and scalability—making them suitable for a wide variety of embedded applications. Understanding these protocols is essential for developing robust inter-device communication in resource-constrained systems.

Memory Protection Units (MPUs). Memory Protection Units (MPUs) are instrumental in enforcing memory access restrictions and safeguarding sensitive data. By segmenting memory into distinct regions with predefined access rules, MPUs help prevent unintended data corruption and strengthen overall system security [39, 9].

Real-Time Operating Systems (RTOS). In safety-critical contexts, real-time processing ensures predictable and timely task execution. Real-Time Operating Systems (RTOS) employ deterministic scheduling algorithms that guarantee deadlines are met, thereby enhancing system reliability and maintaining consistent performance under stringent timing constraints. [59]

Coprocessors. Coprocessors complement the main processing core by offloading specialized computational or input/output tasks. They often operate in low-power or deep-sleep modes [32], helping embedded systems reduce energy consumption and extend operational lifespan. Depending on the architecture, coprocessors may maintain separate memory or share resources with the primary CPU while communicating over a common system bus.

3 Problem Statement & Challenges

Preventing damage arising from compromises of safety-critical devices constitutes a significant challenge in modern embedded systems. Although these systems are typically equipped with software intended to guarantee safe operation, such assurances hold only as long as the underlying platform remains uncompromised. Once an attacker obtains control, the device's behavior can be altered or halted, potentially resulting in severe physical consequences. For instance, malicious manipulation of chemical levels in a water purification system could endanger thousands of individuals. This context motivates the need for an independent monitoring mechanism capable of detecting and constraining malicious behavior in a timely manner.

Insufficiency of existing monitoring strategies. Existing approaches, such as intrusion detection systems (IDSs) [6], monitor systems for anomalous behavior and report suspected compromises. However, most IDS are deployed on external components and primarily observe network traffic [6]. In contrast, the proposed approach integrates monitoring directly into the device and employs a coprocessor to identify violations with higher precision. Rather than merely reporting irregularities, the system enforces active countermeasures, such as stopping or restarting the device when safety violations are detected. Purely network-based IDSs cannot generally restart devices and depend on stable network connectivity, which may not be guaranteed in safety-critical environments [6].

Remote attestation represents another class of techniques that interact with devices to verify their state and regulate their operation. This method, however,

similarly relies on dependable network conditions. The proposed approach, by contrast, operates entirely locally on microcontrollers equipped with a coprocessor and does not require network connectivity or hardware modifications, using the coprocessor for continuous real-time monitoring. While some platforms provide a trusted execution environment, such as *TrustZone* on ARM [49], many microcontrollers, particularly legacy devices, do not support this feature. Furthermore, deploying *TrustZone* typically entails greater complexity than leveraging an existing coprocessor, whereas the latter offers a comparatively straightforward and effective solution, especially for real-time workloads. Coprocessors are connected to general-purpose input/output (GPIO) pins and can perform independent measurements.

Real-Time Requirements. Many safety-critical applications, including power generation facilities, water purification plants, and industrial production systems, are subject to stringent real-time constraints. Such systems frequently rely on *off-the-shelf* embedded hardware and are designed for extended operational lifetimes, sometimes spanning several decades. Even minor modifications to hardware or software may necessitate a comprehensive and costly safety re-certification process. Conventional security mechanisms are often difficult to adopt in this context because they can interfere with deterministic timing behavior. For example, techniques such as Control Flow Integrity (CFI) may introduce non-negligible timing overhead and variability, which is unacceptable for real-time or safety-critical applications [34, 36, 54].

Requirements in Safety-Critical Systems. Safety-critical systems exhibit several domain-specific challenges. Historically, these systems were often isolated from external networks, such that engineering efforts primarily targeted *program faults, human errors, or process flaws* [40]. Safety engineering, therefore, emphasizes the modeling and validation of system-specific safety requirements, employing dedicated methods and formal models to ensure compliance [44]. Two central aspects of such systems are real-time analysis and robustness under unreliable communication conditions, as demonstrated by Masrur et al. [45]. The increasing connectivity of safety-critical systems to external networks introduces additional concerns, including the avoidance of defective software specifications [34] and the mitigation of security vulnerabilities. Recent security incidents involving the ROS2 [17] robotic operating system illustrate that safety-critical systems are not inherently less vulnerable to attacks than conventional IT systems.

Challenges for a Real-Time Safety System in a Constrained Environment. Although monitoring network activity is one possible defensive strategy, embedded devices are frequently deployed in environments with intermittent or unstable connectivity. Consequently, the device should be able to monitor its own behavior without relying on external communication channels. Any additional security mechanisms must preserve real-time properties and either execute with predictable timing or operate in parallel with the primary control logic. For safety-critical applications, a comprehensive safety model is typically specified and implemented on the device [47]. Violations of this model can therefore be

interpreted as indicators of a potential compromise. In summary, *Co-Guard* is designed to address the following challenges:

C1 Eliminate reliance on external networks, which may be unstable or unavailable during emergencies.

C2 Satisfy real-time and safety requirements, ensuring predictability and availability.

C3 Perform independent sensor measurements to mitigate the risk of manipulated sensor data.

C4 Provide local remediation mechanisms, such as secure restarts or safety-oriented shutdowns, to prevent physical harm.

4 Threat Model

We consider an embedded device executing a typical real-time operating system (RTOS), such as FreeRTOS [29]. The device operates in a real-time environment and therefore must exhibit predictable behavior. Infrequent reboots are acceptable and are assumed not to cause physical damage or endanger human safety. The RTOS manages all tasks and ensures that all timing constraints and deadlines are met [59], provided the device remains uncompromised.

The device is deployed in an environment with network connectivity that may be unstable and subject to periodic failures. We assume that the embedded device includes an unused coprocessor, such as the ESP32 from Espressif [23], or hardware with comparable capabilities, such as the RP2040 (*Raspberry Pi Pico*) [52]. The primary function of the device is to control or measure a physical process, and it is connected to actuators and sensors to interact with the physical environment. In the event of a compromise, the device may cause physical damage or pose safety risks. However, a restart or emergency shutdown is assumed to transition the device into a safe state with respect to the controlled physical process.

We assume a remote adversary who can obtain control over the embedded device, for example, by exploiting vulnerabilities that enable remote code execution. The attacker can manipulate actuators and sensors or inject falsified sensor data over the network. Furthermore, the adversary may transmit fabricated sensor readings on the physical bus. The coprocessor is locked during the boot phase or requires a privilege level that the attacker cannot obtain, thereby ensuring that privileged hardware registers remain protected. This assumption is consistent with prior work [10, 57], where memory protection mechanisms such as memory protection units (MPUs) or memory management units (MMUs) are used to prevent code injection and unauthorized firmware modification.

Physical attacks, such as hardware replacement or direct tampering with the device, are considered out of scope.

5 Design

Embedded devices typically regulate physical processes by acquiring sensor measurements, processing data, and issuing commands to external actuators [55]. Owing to their tight coupling with the physical environment, these devices are subject to safety constraints that adversaries may violate, potentially resulting in physical damage or harm. Given the limited computational resources available on such platforms, contemporary resource-intensive security mechanisms are often impractical. Instead, safety violations, because they manifest as direct, measurable effects on the physical process, can serve as reliable indicators of attack and can be detected and mitigated accordingly.

Unreliable network conditions can render remote monitoring ineffective (see challenge **C1**). Moreover, remote defenses may introduce nontrivial delays before shutdown, which can be insufficient to prevent damage in time-sensitive scenarios (see **C2**). Accurate assessment of system state necessitates independent sensor measurements (see **C3**). Finally, the device must be capable of either transitioning to a safe shutdown state or rebooting into a known-good configuration to stabilize the controlled physical process (see **C4**).

We adopt a local security monitoring and remediation strategy to address these challenges. The approach leverages common embedded real-time architectures that include a coprocessor [32]. In particular, sensors, memory, the main CPU, and the coprocessor often share a common bus [32], as illustrated in Figure 1.

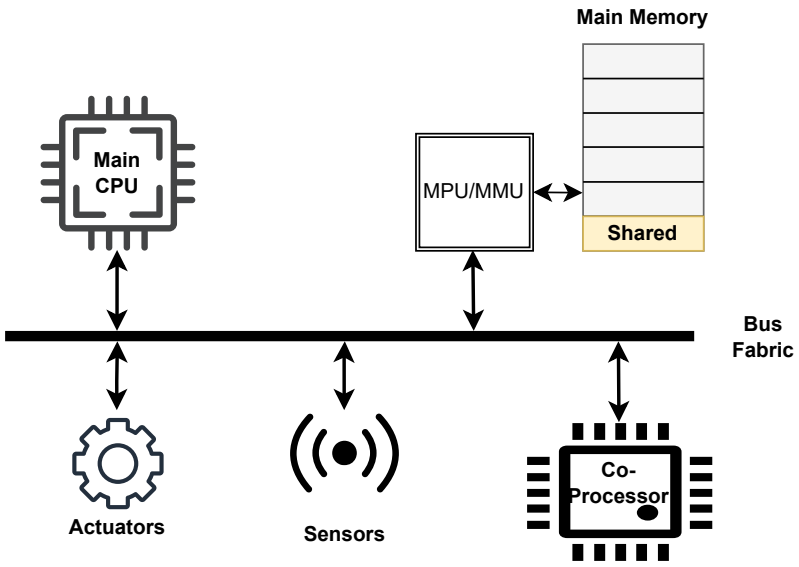

Fig. 1. Architectural Overview of the Components required by *Co-Guard*.

As discussed in Section 2, the coprocessor functions as an independent unit with either dedicated or shared memory. Commonly used for I/O handling, it can operate autonomously and perform independent sensor measurements. An MPU can prevent application code from accessing memory assigned to the coprocessor, thereby maintaining isolation (see Section 4).

Building on this prevalent coprocessor-centric architecture, *Co-Guard* fulfills the identified requirements without necessitating specialized hardware modifications. Off-the-shelf microcontrollers equipped with coprocessors suffice to realize the design, as demonstrated in our implementation (see Section 6) and evaluation (see Section 7). The coprocessor operates independently of the main CPU and does not rely on network connectivity, addressing challenge **C1**. It executes in parallel with the main CPU, enabling continuous monitoring for safety violations without disrupting primary control tasks, thereby addressing challenge **C3**. Upon detecting a safety violation, the coprocessor can trigger an emergency shutdown or reboot the device into a secure state (**C4**). Some platforms, such as the ESP32, provide secure boot features [23], ensuring that the device restarts with authentic, uncompromised firmware. An emergency restart can be initiated through software mechanisms (e.g., a watchdog timer) or via GPIO control by connecting a coprocessor-driven GPIO to the device's *Reset* pin.

Monitoring for safety violations in embedded systems is intrinsically challenging. *Co-Guard* must detect violations with high fidelity while avoiding excessive false positives. The coprocessor is central to validating that the device's responses to safety threats are correct and timely. For example, in a water purification system, a sudden increase in a chemical concentration may legitimately trigger a mitigation action; failing to account for this response could lead to spurious violation reports and unnecessary reboots. The specific monitoring logic and response policies are application-dependent. Certain applications, such as a smoke detector (see Evaluation 7.1), may require relatively simple checks, whereas others, such as water purification (see Evaluation 7.1), demand more sophisticated decision-making capabilities on the coprocessor.

The approach is intentionally generic. The coprocessor continuously acquires sensor measurements and, upon any reading exceeding a safety threshold, enters a heightened monitoring mode. This mode is essential for distinguishing actual safety violations from benign anomalies, measurement noise, or legitimate threshold excursions, and for ensuring appropriate countermeasures are implemented. For instance, an observed increase in water pollution may be genuine and trigger automatic remediation; in such cases, the absence of a required countermeasure—not the threshold exceedance itself—constitutes the safety violation.

6 Implementation

To evaluate the practicality and adaptability of the proposed concept, *Co-Guard* was implemented on four real-world microcontrollers developed by two semiconductor manufacturers. These implementations were validated in diverse scenarios,

including a water purification system and a smoke detector, thereby demonstrating the applicability and effectiveness of *Co-Guard* in real-world environments.

Espressif is a leading Chinese semiconductor company that manufactures widely adopted microcontrollers for IoT systems, including the ESP32, ESP32-S2, and ESP32-S3. These devices are popular due to their support for standard peripherals and communication protocols (I2C, UART, and SPI), as well as their integrated Bluetooth and Wi-Fi connectivity [62]. Their combination of high connectivity and cost efficiency makes them particularly suitable for cloud-based systems. All three ESP32 variants integrate a coprocessor, which is essential for implementing *Co-Guard*.

The Raspberry Pi Foundation, renowned for its commitment to open-source hardware [26], developed the RP2040 microcontroller. Since its release in early 2021, the RP2040 has gained substantial popularity [33], powering development boards such as the Arducam Pico4ML [3] and SparkFun's MicroMod RP2040 [58]. Although the RP2040 lacks a conventional coprocessor, it employs programmable programmable input-output (PIO) blocks [52] that can perform limited coprocessing tasks. While these PIO blocks provide significantly reduced functionality compared to the ESP32 coprocessors, they serve to illustrate the flexibility of our concept, which is applicable across architectures ranging from feature-rich to highly constrained designs.

6.1 Details on the Microcontrollers

The ESP32 (S1). Released in 2016, the ESP32-S1 remains a widely utilized microcontroller platform. Its coprocessor, situated within the real-time clock (RTC) domain, is based on an finite-state machine (FSM) architecture and is programmable in assembly using a custom instruction set. This instruction set streamlines operations such as sensor data acquisition via dedicated instructions for accessing analog-to-digital converter (ADC) units [23]. Implementing *Co-Guard* requires monitoring the actuator and sensor statuses. Although the coprocessor instruction set lacks direct access to all GPIOs, a mapping mechanism enables access to the GPIO registers through the RTC domain. The ESP32-S1 provides a reboot mechanism accessible via an Espressif API. However, since the coprocessor cannot directly invoke this API, we employ an alternative solution by connecting the enable (EN) pin to a GPIO pin, which, when pulled low, triggers a system reboot.

ESP32-S2 and ESP32-S3. The ESP32-S2 and ESP32-S3 microcontrollers closely resemble the ESP32-S1, particularly regarding their coprocessor architecture. Both feature two coprocessors: an FSM-based coprocessor, maintained for backward compatibility, and an additional RISC-V-based coprocessor. The supporting toolchain remains consistent across models. Similarly, both variants employ the same reboot mechanism as the ESP32-S1.

The RP2040 microcontrollers. In contrast to the ESP32 family, the RP2040 uses PIO blocks rather than a dedicated coprocessor. These blocks comprise eight programmable state machines with a compact instruction set, limited to direct interactions with GPIO pins. Despite these constraints, the RP2040 can

nonetheless support the proposed concept by reading sensor outputs and verifying countermeasure states through GPIO interactions. The device's reboot mechanism mirrors that of the ESP32 series, using a GPIO pin to initiate a hardware reset.

6.2 Programming Coprocessors

Espressif microcontrollers are typically programmed using the ESP-IDF framework [25], which enables application development in C. However, the coprocessor programming process differs among models. Figure 2 depicts the differences between their respective build processes. The ESP32-S1's coprocessor requires assembly programming via C macros, employing a specialized instruction set that facilitates operations such as ADC sampling and I2C communication. This design necessitates a custom C preprocessor to support instruction generation.

Although limited in capability, the RP2040's PIO blocks can also be programmed using a dedicated assembler, which compiles source files into C header files. These headers embed the generated program as opcode arrays, allowing seamless integration into higher-level C applications [52].

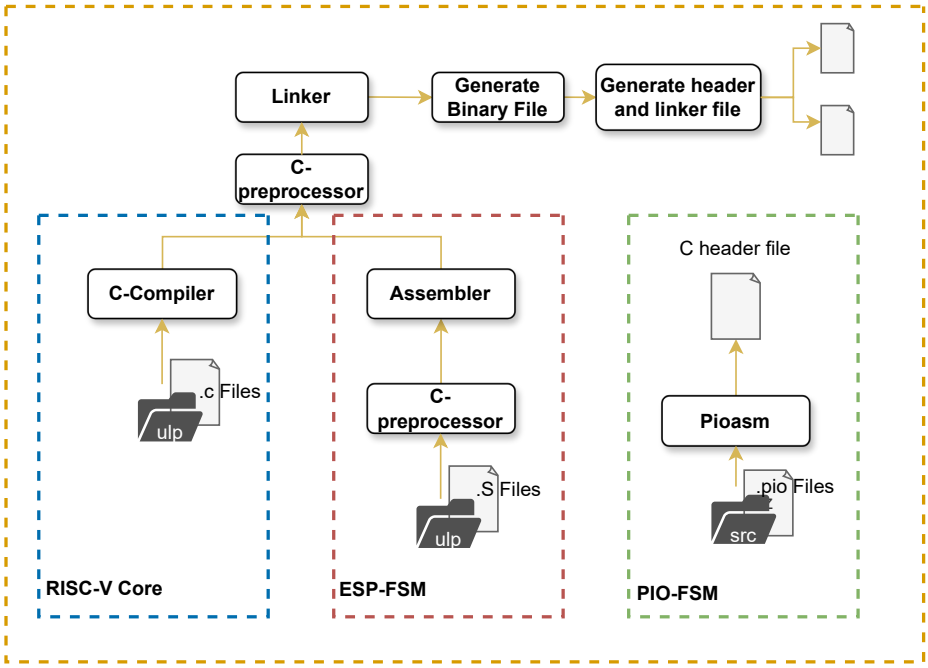

Fig. 2. Overview of the Coprocessor Build Chain [24, 52]

6.3 Safety Monitoring Implementation

Many microcontrollers integrate a coprocessor, though they differ notably in terms of programmability and internal architecture (FSM vs. RISC-V). This section presents the implementation of the safety monitoring system described in Section 5. The process comprises three primary stages: acquiring environmental data, verifying countermeasures, and initiating a reboot when safety violations are detected.

For the ESP32-S1, sensor readings are retrieved through the ADC unit, assuming that the sensor is connected to an analog-capable GPIO pin. These readings are then evaluated against countermeasure states by inspecting the corresponding GPIO pins. Implementing this procedure requires careful mapping between GPIO pins and the RTC domain to ensure correct signal access. In accordance with the design presented in Section 5, when an anomaly is detected, the system triggers a reboot by activating the designated GPIO macro responsible for output control.

The ESP32-S2 and ESP32-S3 enable a similar monitoring approach; however, their C-based development support facilitates easier interaction with both ADC units and GPIO pins through high-level APIs.

In contrast, the RP2040 implementation is inherently more constrained. Its PIO blocks cannot directly access peripherals such as ADC units, and instruction memory is highly limited. Consequently, environmental data can only be obtained from sensors that provide digital outputs (DOUT), often in addition to analog (AOUT) or bus interfaces (e.g., I2C lines SCL and SDA). Sensors such as the MQ2 gas detector [60] or the WAVESHARE obstacle detection sensor [64] provide digital outputs. Environmental parameters are thus read from GPIO states and stored in general-purpose registers for subsequent comparison against expected conditions. Countermeasure verification proceeds similarly, relying on evaluation of GPIO states. System reboots are triggered by enabling a designated output GPIO pin.

This implementation demonstrates that even limited coprocessing architectures, such as the RP2040, can execute the proposed safety monitoring algorithm.

7 Evaluation

To assess the practical applicability of the proposed concept, we evaluated it in two real-world scenarios: a water purification system and a smoke detector. Due to the limited functionality of its coprocessing units, the RP2040 microcontroller proved inadequate for the water purification scenario, as its hardware limitations hindered the implementation of safety-monitoring features for water treatment.

Beyond the case studies, the evaluation also examines the additional power consumption incurred by integrating the coprocessor. The water purification system was chosen to reflect a scenario analogous to the cyberattack on the Israeli water infrastructure [28]. At the same time, the smoke detector represents a ubiquitous safety-critical device found in both residential and industrial environments. Similar to prior work such as *OP-DFI* [66], which evaluated its approach using

PX4 drones [46, 63], our selected systems are highly safety-relevant, emphasizing water treatment and smoke detection.

7.1 Case Studies

Water Purification Systems are safety-critical infrastructures whose malfunction poses severe risks to public health and welfare. The provision of clean water is essential for both human and ecological well-being and underpins numerous economic activities. Water purification typically involves multiple processes, including adsorption (activated carbon filtering), disinfection (chlorine dosing), softening (removal of calcium and magnesium ions), and filtration (elimination of suspended particles) [21]. A key quality indicator is the Total Dissolved Solids (TDS) value, commonly expressed in milligrams per liter (mg/L) [18].

A proof-of-concept water purification prototype was implemented on the three ESP32 microcontrollers (S1, S2, and S3), focusing on maintaining a TDS concentration near 500 mg/L, as recommended by the U.S. Environmental Protection Agency [11]. The experimental setup comprised a TDS sensor module and a miniature water pump (Figure 3). The pump introduced distilled water to regulate the TDS level. For functional validation, the TDS concentration was artificially increased by adding salt to the tank, at which point the system autonomously activated the pump to restore safe levels.

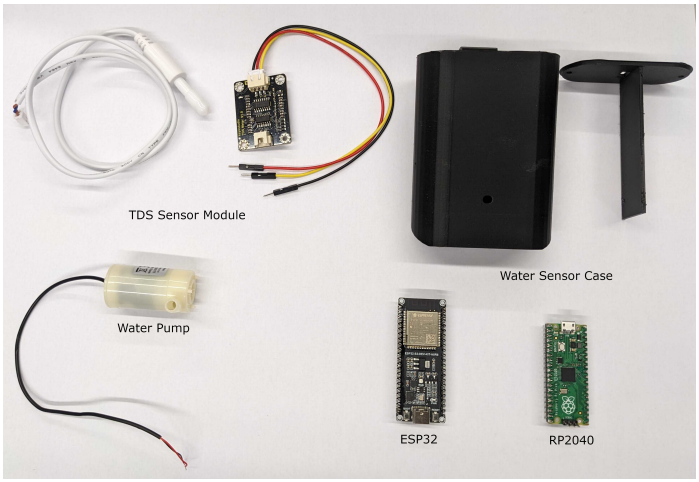

Fig. 3. Hardware Setup for the Water Purification Case Study.

The water purification prototype included a deliberately vulnerable web interface that adversaries could exploit to alter the TDS threshold to unsafe levels. In the absence of *Co-Guard*, elevated TDS readings were disregarded. Conversely, when *Co-Guard* was active, the system successfully detected unsafe

conditions and rebooted into a clean operational state, thereby mitigating the compromise.

This scenario is one of the more complex test cases for *Co-Guard*, as the monitoring logic must verify whether countermeasures have already been executed before triggering a reboot to avoid repeated restarts. Although the TDS metric was central to this study, the ESP32 family and the *Co-Guard* coprocessors can handle various sensors and interdependencies between sensor readings and actuator states (countermeasures).

Due to hardware constraints, such as the RP2040's lack of ADC access within its PIO blocks, a full implementation of *Co-Guard* for the water purification setup was infeasible. While this limitation could be overcome using an external ADC module connected to the GPIO pins, we refrained from making hardware modifications, as discussed in Section 4. Nonetheless, the RP2040 proved fully capable for the smoke detection case.

Smoke Detection Systems are essential in both residential and industrial safety contexts, where malfunctions can cause significant damage or loss of life. In many jurisdictions, including Germany, smoke detectors are mandatory [7]. These systems range from basic detectors capable solely of smoke detection to advanced multi-sensor devices that can identify gases such as carbon monoxide (CO), liquefied petroleum gas (LPG), methane, or propane. While simpler models merely activate an audible alarm, more advanced systems are often networked, providing coordinated alerts and source localization.

For this study, we implemented an interconnected smoke-detection prototype using an MQ2 gas sensor module [60], which can detect smoke and multiple gases (Figure 4). The prototype was deployed on both RP2040 and ESP32 platforms, illustrating *Co-Guard* 's adaptability to heterogeneous hardware architectures.

Several commercial products, such as the Shelly Plus Smoke [53], are based on ESP32-class microcontrollers and integrate support for smart home ecosystems, including Amazon Alexa and Google Home. Our proof-of-concept thus emulates commercially realistic use cases, demonstrating that *Co-Guard* can be integrated into existing market-ready devices.

The prototype was validated by generating artificial smoke to ensure correct alarm triggering. A deliberately vulnerable interface was introduced to allow attackers to suppress alarm functionality. After deploying *Co-Guard* on both the ESP32 and RP2040 smoke detectors, the system successfully detected attempts to suppress the alarm, initiated a reboot, and restored functional integrity. Upon reboot, the device correctly raised the alert when exposed to smoke.

These case studies collectively demonstrate that *Co-Guard* can be implemented on commercially available hardware and effectively mitigates attack-induced faults in safety-critical systems. In doing so, it ensures that devices such as smoke detectors continue to operate reliably even after attempted compromise, thereby safeguarding human life and property.

Power Consumption. Beyond functional validation, we also quantified the power consumption of *Co-Guard* across all tested microcontrollers. Energy efficiency is a crucial requirement in embedded systems [51]. As outlined in Section 4,

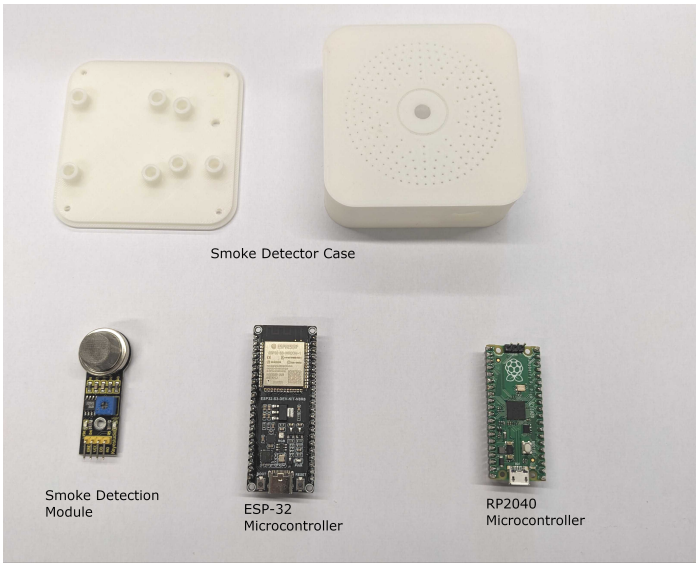

Fig. 4. Hardware Setup for the Smoke Detection Case Study.

the target devices include a dormant coprocessor, which *Co-Guard* repurposes for runtime monitoring. We measured total power draw with and without coprocessor activity over approximately ten minutes.

Figure 5 presents the experimental results. The ESP32-S1 exhibited the highest overhead, with a 9% increase during the smoke detector scenario. The ESP32-S2 and ESP32-S3 showed marginal increases of approximately 2%, while the RP2040 demonstrated only a 1% rise. In the water purification setup, the increase was 2% for the ESP32-S1 and ESP32-S2, and 1% for the ESP32-S3.

Overall, these results indicate that the additional energy cost incurred by *Co-Guard* remains modest, supporting its deployment in real-world embedded environments.

8 Security Considerations

In the previous sections, we demonstrated that *Co-Guard* can be deployed on a range of real-world safety-critical devices while ensuring reliable protection of the surrounding environment. In this section, the focus shifts to the security properties and limitations of the proposed approach.

Contemporary research typically distinguishes two primary attacker objectives: *Reconnaissance* and *targeted attacks* [30]. *Reconnaissance* refers to adversaries collecting information, often passively, to compromise confidentiality. A representative *Reconnaissance* attack is ScheduLeak [15]. Through *Reconnaissance*, an attacker attempts to infer a wide range of information, including not only software characteristics but also environmental and operational conditions. For

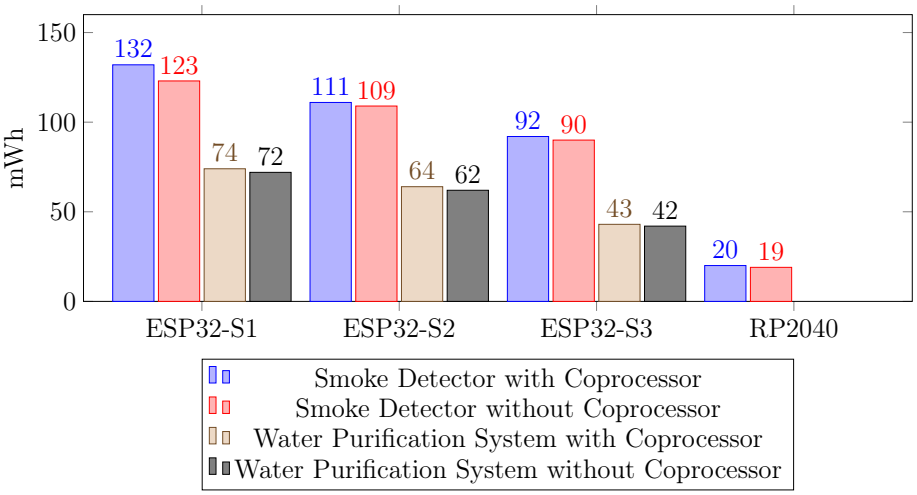

Fig. 5. Power Consumption Comparison with and without *Co-Guard*.

example, Liu et al. [38] recover engine speed by analyzing real-time task scheduling behavior. In contrast, targeted attacks primarily aim to manipulate device functionality and violate temporal constraints.

Many practical attacks follow the phases described in the Cyber Kill Chain, originally introduced by Lockheed Martin [43] and later adapted to Industrial Control Systems (ICSs) [5].

In Figure 6, *Co-Guard* is positioned within the ICS Kill Chain at a critical stage, immediately before physical damage may occur. At this point, *Co-Guard* serves as the final safeguard before a successful attack can lead to physical consequences. While tools such as *SCADMAN* [1] detect tampering and report compromises, *Co-Guard* performs immediate mitigation by triggering an emergency reboot or shutdown to prevent harm. This proactive response enables *Co-Guard* to effectively prevent physical damage, a capability that is largely absent from many existing approaches, including *OP-DFI* [66] and *KAGE* [20], which primarily emphasize preventing successful exploitation rather than stopping physical effects.

Although *Co-Guard* does not defend against side-channel attacks, basic *Reconnaissance* alone does not directly cause physical damage, as it primarily affects confidentiality. *Co-Guard* is explicitly designed to intervene once an attacker actively tampers with the system, enforcing safety constraints before critical operations are affected. The coprocessor's isolation from the main CPU ensures that, after a compromise, it cannot be influenced by the attacker, and this isolation is essential for preventing further escalation. However, Denial-of-Service (DoS) attacks targeting *Co-Guard* itself may still result in physical harm. The following subsections address these aspects by examining the *separation* between the main

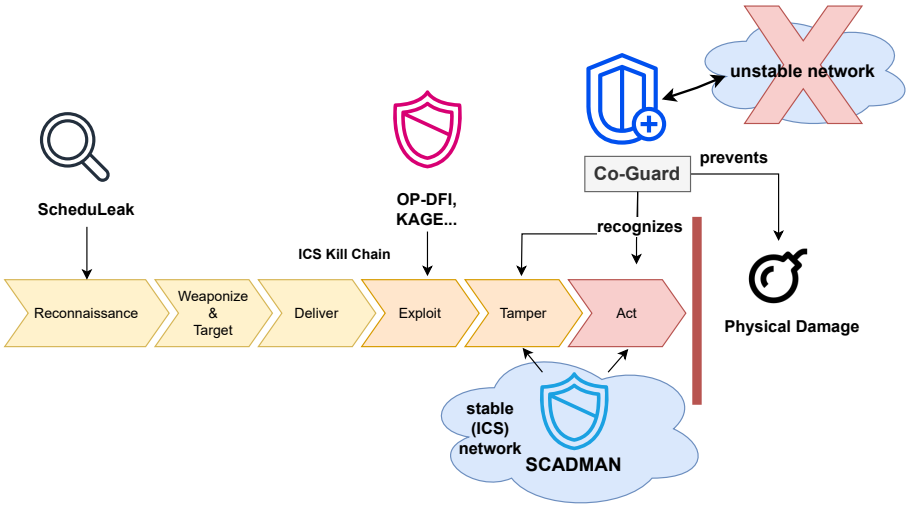

Fig. 6. Defense Objective of *Co-Guard* within the ICS Kill Chain [5]

CPU and the coprocessor and by discussing how *Co-Guard* mitigates sustained DoS attacks.

Separation between Main CPU and Coprocessor. The coprocessor and the main CPU constitute independent hardware components, even if they are integrated on the same chip and share certain memory regions for data exchange. For example, the ESP32 family provides such a shared memory segment [23]. While shared memory simplifies normal operation, it could undermine safety monitoring if the main CPU can access or modify coprocessor data.

Memory access can be constrained by hardware mechanisms such as the MPU on the RP2040 [52] and the MMU on the ESP32 [23]. The MPU offers static access control. In contrast, the MMU provides fine-grained virtual address mapping, enabling the system designer to prevent the main CPU from accessing memory regions used by the coprocessor.

Although the MPU and MMU can prevent the main CPU from reading or writing to coprocessor memory, their configuration must be locked after initialization to prevent an attacker from reconfiguring them. ARM architectures typically enforce privilege separation between different system modes [4]. On the RP2040, the MPU is configured in privileged mode during system startup, after which the system transitions to unprivileged mode for normal operation. The ESP32 family employs a *process ID controller (PID)* mechanism [23], where the MMU enforces access restrictions based on the current PID. Application code usually executes under PIDs that are not permitted to reconfigure the MMU. This arrangement ensures that application code running on the main CPU cannot interfere with the coprocessor's memory.

False Data Injection through the Main CPU. One potential attack against *Co-Guard* consists of injecting falsified sensor readings via the main CPU to

mislead the coprocessor. However, such an attack can generally be detected by the coprocessor. The genuine sensor values continue to appear on the bus, and any discrepancy between the bus data and the values observed through the main CPU can be identified. An anomalously high number of sensor readings or inconsistent patterns would further increase suspicion, enabling the coprocessor to recognize and flag the attack.

Denial-of-Service Attacks through Forced Reboot or Shutdown. A DoS adversary may attempt to force the device into a persistent reboot loop, thereby preventing the deployment of effective countermeasures. This scenario would require repeated compromises to keep the system in a nearly continuous restart state. Although network instability (see Section 3) may inherently limit the feasibility of such an attack, it nonetheless represents a relevant threat.

Because the device is generally assumed to operate under unstable networking conditions, temporary offline periods do not inherently impede its functionality. Consequently, this work proposes taking the device offline for a predefined time window after a reboot as a countermeasure, allowing it to operate without being subject to continuous compromise. The duration of this offline period can be tuned to the specific application and deployment context to avoid safety violations while preserving normal operation as much as possible.

9 Related Work

Safety-critical systems pose distinctive challenges for integrating security mechanisms, as they have traditionally operated in isolation from external networks. Historically, safety considerations have primarily addressed *program faults, human errors, or process flaws*, while external adversarial threats received comparatively limited attention. With the increasing deployment of networked systems, however, safety-critical devices have become exposed to novel classes of vulnerabilities. A recent SoK paper by Hasan et al. [30] emphasizes that most existing security mechanisms concentrate on scheduler-level techniques, whereas *Co-Guard* adopts a different strategy that remains non-intrusive with respect to real-time constraints.

Safety engineering is concerned with modeling and guaranteeing the safety requirements of specific systems [44]. Such systems must cope with challenges such as maintaining real-time performance under unreliable communication conditions, as demonstrated by Masrur et al. [45]. Another persistent problem is the avoidance of defective software specifications [34]. Nevertheless, safety mechanisms do not inherently provide security guarantees. The introduction of network connectivity into safety-critical environments has rendered them susceptible to remote compromises, as illustrated by systems such as the robotic operating system ROS2 [17].

IoTMon [19] investigates the physical interactions among IoT devices and how adversaries can exploit these interactions to cause harm. For instance, a compromised heater could trigger automatic window opening, thereby enabling unauthorized physical access. While *IoTMon* primarily focuses on the interplay

between multiple devices, *Co-Guard* concentrates on safety violations within a single device, as illustrated by the water purification case study (see Section 7.1).

Opportunistic Data Flow Integrity (OP-DFI) [66] is a recent approach to securing embedded applications by enforcing data-flow integrity. This mechanism introduces execution delays due to periodic integrity checks during system operation. Furthermore, it protects only a subset of the system and depends on ARM hardware features such as memory tag extensions. In contrast, *Co-Guard* treats safety as a security property and continuously monitors for safety violations in real time by evaluating sensor measurements on a coprocessor. By detecting behavioral deviations indicative of safety violations, *Co-Guard* can take preventive action, assuming that a certified safety-critical device that fails to meet its safety criteria has likely been compromised.

Silhouette [68] is a compiler-assisted defense that deploys a hardened shadow stack to provide coarse-grained CFI for embedded systems. However, *Silhouette* has not been evaluated in the context of real-time systems. *KAGE* [20] applies CFI to real-time operating systems (RTOS) such as FreeRTOS, thereby enhancing security by reducing the attack surface available for return-oriented programming attacks.

IoTGuard [14] is a dynamic, policy-based enforcement framework that monitors IoT device behavior in real time using models derived from firmware instrumentation. However, it targets platforms such as *If This Then That (IFTTT)*[1] and *Microsoft Flow (Power Automate)*[2]. For *IoTGuard*, a third device is employed to collect application execution traces and to construct the dynamic model. This architecture differs from that of *Co-Guard*, which operates directly on the protected device without requiring an auxiliary monitoring platform, rendering it more suitable for real-time systems.

SCADMAN [1], similar to *Co-Guard*, monitors the behavior of cyber-physical systems. To detect inconsistencies, it simulates the precise state of all Programmable logic controllers (PLCs) within an ICS. However, *SCADMAN* is intended for deployments with multiple interacting PLCs in stable network environments and depends on a central monitoring instance. In contrast, *Co-Guard* targets single, resource-constrained real-time devices operating in potentially unstable network conditions, for which centralized monitoring is not practicable.

9.1 Attestation in Safety-Critical Systems

Remote attestation constitutes another prominent technique for detecting compromised devices, particularly in resource-constrained safety-critical settings. However, traditional remote attestation schemes typically rely on a third-party verifier, which presupposes stable network connectivity and therefore conflicts with challenge *C1* (see Section 3). Carpent et al. [12] identify several fundamental challenges associated with remote attestation, including a trade-off between system availability and security.

[1] `https://ifttt.com/`
[2] `https://www.microsoft.com/de-de/power-platform/products/power-automate`

ERASMUS [13] proposes a potential mitigation by employing periodic self-measurements instead of purely on-demand attestation, which the verifier later collects. Preschern et al. [50] present a software-based remote attestation protocol for safety-critical systems. Their work primarily considers malicious PLC communication software; therefore, they employ a *black communication channel* to support the attestation process [50]. *SCAPI* [35] focuses on detecting physical attacks and identifying compromised hardware and software components. However, *SCAPI* relies on a network of devices to detect compromised hardware, which again conflicts with challenge **C1**, and it does not concentrate on safety-critical execution. *PAtt* [27] addresses remote attestation of control systems, which typically also execute safety-critical tasks. *PAtt* observes PLC control data to detect manipulated PLC logic and reports device manipulation events. In contrast, *Co-Guard* either initiates an emergency shutdown or, *at least temporarily*, returns the device to a benign operational state.

External Attestation. Carpent et al. [12] describe an availability conflict between real-time execution and software-based attestation. DMA'n'Play [61] offers a potential resolution to this conflict. Surminski et al. [61] propose a plug-in device that supports two operational modes. The central idea is to transmit data via DMA to an external device for attestation. In the *DMA'n'Play To-Go* mode, data collected from the attested device is forwarded to a remote verifier. Network outages directly affect this mode and therefore conflict with challenge **C1**. In the second mode, a constrained verifier is directly attached to the attested device and monitors it for firmware modifications.

Instead of employing an external device, *Co-Guard* integrates through a coprocessor into the platform. The primary use case of DMA'n'Play is remote attestation, with *DMA'n'Play To-Go* as its main operational mode, and it has not been designed with safety-critical deployments as a central concern. In contrast, *Co-Guard* is explicitly tailored to the local monitoring of safety-critical systems. As its primary objective, *Co-Guard* incorporates safety-focused capabilities, including active prevention of safety violations. To endow DMA'n'Play with capabilities comparable to *Co-Guard*, the device's sensors would also need to be connected to the external attestation hardware. This requirement introduces additional hardware complexity, for example, multiplexers to connect a single sensor to two independent devices. In *Co-Guard*, the coprocessor has internal, independent bus access to the **GPIO** pins, eliminating the need for external wiring.

Need for Multiple Diverse Security and Safety Approaches. Given the heterogeneity of embedded devices and their diverse requirements, ranging from stringent real-time constraints to highly interconnected environments, there is no *one-size-fits-all* solution. Safety-critical systems in particular exhibit unique challenges that vary across application domains. In some cases, even the identification of threats and the formulation of appropriate threat models remain topics of ongoing discussion [56]. While attestation frequently represents a viable option for monitoring safety-critical devices, it fails in the presence of partial network failures. Consequently, approaches such as *Co-Guard* are required to preserve

operational safety. *Co-Guard* can furthermore be deployed in conjunction with existing remote attestation mechanisms.

10 Conclusion and Summary

In this paper, we presented *Co-Guard*, a coprocessor-based approach for monitoring safety-critical devices and preventing physical harm in emergency and post-compromise scenarios. By operating locally on the device rather than relying on network connectivity or external monitoring components, *Co-Guard* remains effective even in unstable or offline network environments. The coprocessor autonomously acquires sensor measurements and detects potential safety violations, enabling *Co-Guard* to reboot the device into a safe state when required.

The practicality of *Co-Guard* was demonstrated through two case studies: a water purification system and a smoke detector, both representative of real-world safety-critical applications where compromise could result in severe damage. In both scenarios, *Co-Guard* successfully identified unsafe conditions and prevented harm by initiating automatic reboots. Furthermore, the implementation across multiple microcontroller platforms, including the resource-constrained RP2040 and the more powerful ESP32 family, illustrates the approach's portability and adaptability across heterogeneous embedded environments.

Overall, this work contributes a novel perspective on post-compromise security for embedded systems. Whereas much prior research concentrates on preventing initial exploitation, *Co-Guard* targets the critical phase following compromise, maintaining device safety even under unreliable networking conditions. By supporting diverse microcontroller architectures and focusing on safety as a primary objective, *Co-Guard* provides a flexible and robust mechanism for protecting safety-critical devices against malicious attacks.

Acknowledgments. This work was funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of Priority Programme RAINCOAT II (440059533) and under Germany's Excellence Strategy - EXC 2092 CASA - 390781972.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Adepu, S., Brasser, F., Garcia, L., Rodler, M., Davi, L., Sadeghi, A.R., Zonouz, S.: Control behavior integrity for distributed cyber-physical systems. In: 2020 ACM/IEEE 11th International Conference on Cyber-Physical Systems (ICCPS). pp. 30–40 (2020). https://doi.org/10.1109/ICCPS48487.2020.00011
2. Anand, N., Joseph, G., Oommen, S.S., Dhanabal, R.: Design and implementation of a high speed serial peripheral interface. In: 2014 International Conference on Advances in Electrical Engineering (ICAEE). pp. 1–3. IEEE (2014)

3. Arducam: Pico4ml pro tinyml dev kit, https://docs.arducam.com/Arduino-SPI-camera/Legacy-SPI-camera/Pico/Arducam-Pico4ML-TinyML/Pico4ML-Pro-TinyML-Dev-Kit/
4. ARM Holdings Limited: Armv6-m architecture reference manual. online (2024), https://developer.arm.com/documentation/ddi0419/latest/
5. Assante, M.J., Lee, R.M.: The industrial control system cyber kill chain. SANS Institute InfoSec Reading Room **1**(1), 2 (2015)
6. Bace, R.G., Mell, P.: Intrusion detection systems. Tech. rep., National Institute of Standards and Technology (NIST) (2001)
7. Baden-Wuerttemberg, F.s.: § 15brandschutz, https://dejure.org/gesetze/LBO/15.html
8. Balachandran, S.: General purpose input output (gpio). Michigan State University College of Engineering. Published pp. 08–11 (2009)
9. Barbareschi, M., Barone, S., Casola, V., Montone, P., Moriconi, A.: A memory protection strategy for resource constrained devices in safety critical applications. In: 2022 6th International Conference on System Reliability and Safety (ICSRS). pp. 533–538. IEEE (2022)
10. Bellec, N., Hiet, G., Rokicki, S., Tronel, F., Puaut, I.: Rt-dfi: Optimizing data-flow integrity for real-time systems. In: ECRTS 2022-34th Euromicro Conference on Real-Time Systems. pp. 1–24 (2022)
11. Benham, B.L., Ling, E., Wright, B., Haering, K.: Virginia household water quality program: Total dissolved solids (tds) in household water. Tech. rep., Virginia Cooperative Extension (2011)
12. Carpent, X., Eldefrawy, K., Rattanavipanon, N., Sadeghi, A.R., Tsudik, G.: Reconciling remote attestation and safety-critical operation on simple iot devices. In: 2018 55th ACM/ESDA/IEEE Design Automation Conference (DAC). pp. 1–6. IEEE (2018)
13. Carpent, X., Tsudik, G., Rattanavipanon, N.: Erasmus: Efficient remote attestation via self-measurement for unattended settings. In: 2018 Design, Automation & Test in Europe Conference & Exhibition (DATE). pp. 1191–1194. IEEE (2018)
14. Celik, Z.B., Tan, G., McDaniel, P.D.: Iotguard: Dynamic enforcement of security and safety policy in commodity iot. In: NDSS (2019)
15. Chen, C.Y., Mohan, S., Pellizzoni, R., Bobba, R.B., Kiyavash, N.: A novel side-channel in real-time schedulers. In: 2019 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 90–102. IEEE (2019)
16. Cohen, G.: Throwback attack: How the modest bowman avenue dam became the target of iranian hackers, https://www.industrialcybersecuritypulse.com/facilities/throwback-attack-how-the-modest-bowman-avenue-dam-became-the-target-of-iranian-hackers/
17. Deng, G., Xu, G., Zhou, Y., Zhang, T., Liu, Y.: On the (in) security of secure ros2. In: Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security. pp. 739–753 (2022)
18. Devesa, R., Dietrich, A.: Guidance for optimizing drinking water taste by adjusting mineralization as measured by total dissolved solids (tds). Desalination **439**, 147–154 (2018). https://doi.org/https://doi.org/10.1016/j.desal.2018.04.017, https://www.sciencedirect.com/science/article/pii/S0011916417323056
19. Ding, W., Hu, H.: On the safety of iot device physical interaction control. In: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security. pp. 832–846 (2018)

20. Du, Y., Shen, Z., Dharsee, K., Zhou, J., Walls, R.J., Criswell, J.: Holistic {Control-Flow} protection on {Real-Time} embedded systems with kage. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2281–2298 (2022)
21. Dutta, D., Arya, S., Kumar, S.: Industrial wastewater treatment: Current trends, bottlenecks, and best practices. Chemosphere **285**, 131245 (2021). https://doi.org/https://doi.org/10.1016/j.chemosphere.2021.131245, https://www.sciencedirect.com/science/article/pii/S0045653521017173
22. El Khaled, Z., Mcheick, H.: Case studies of communications systems during harsh environments: A review of approaches, weaknesses, and limitations to improve quality of service. International journal of distributed sensor networks **15**(2), 1550147719829960 (2019)
23. Espressif Systems: Esp32 technical reference manual. online (2024), https://www.espressif.com/sites/default/files/documentation/esp32_technical_reference_manual_en.pdf
24. Espressif Systems: Ulp coprocessor programming. online (2024), https://docs.espressif.com/projects/esp-idf/en/stable/esp32/api-reference/system/ulp.html
25. Espressif Systems (Shanghai) Co., L.: Esp-idf programming guide, https://docs.espressif.com/projects/esp-idf/en/stable/esp32/index.html
26. Foundation, R.P.: About the raspberry pi foundation, https://www.raspberrypi.org/about/
27. Ghaeini, H.R., Chan, M., Bahmani, R., Brasser, F., Garcia, L., Zhou, J., Sadeghi, A.R., Tippenhauer, N.O., Zonouz, S.: {PAtt}: Physics-based attestation of control systems. In: 22nd International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2019). pp. 165–180 (2019)
28. Goud, N.: Cyber attack on israel water system, https://www.cybersecurity-insiders.com/cyber-attack-on-israel-water-system/
29. Guan, F., Peng, L., Perneel, L., Timmerman, M.: Open source freertos as a case study in real-time operating system evolution. Journal of Systems and Software **118**, 19–35 (2016)
30. Hasan, M., Kashinath, A., Chen, C.Y., Mohan, S.: Sok: Security in real-time systems. ACM Computing Surveys (2024)
31. U.S. Department of Homeland Security, N.C., Center, C.I.: Mar-17-352-01 hatman - safety system targeted malware (update b) - cisa, https://www.cisa.gov/sites/default/files/documents/MAR-17-352-01%20HatMan%20-%20Safety%20System%20Targeted%20Malware%20%28Update%20B%29.pdf
32. Hounsell, B., Taylor, R.: Co-processor synthesis: a new methodology for embedded software acceleration. In: Proceedings Design, Automation and Test in Europe Conference and Exhibition. vol. 1, pp. 682–683. IEEE (2004)
33. Jagdale, S.: Why raspberry pi's rp2040 is the popular choice for development boards, https://embeddedcomputing.com/technology/open-source/development-kits/why-raspberry-pis-rp2040-is-the-popular-choice-for-development-boards
34. Knight, J.C.: Safety critical systems: challenges and directions. In: Proceedings of the 24th international conference on software engineering. pp. 547–550 (2002)
35. Kohnhäuser, F., Büscher, N., Gabmeyer, S., Katzenbeisser, S.: Scapi: a scalable attestation protocol to detect software and physical attacks. In: Proceedings of the 10th ACM Conference on Security and Privacy in Wireless and Mobile Networks. pp. 75–86 (2017)

36. Koscher, K., Czeskis, A., Roesner, F., Patel, S., Kohno, T., Checkoway, S., McCoy, D., Kantor, B., Anderson, D., Shacham, H., et al.: Experimental security analysis of a modern automobile. In: 2010 IEEE symposium on security and privacy. pp. 447–462. IEEE (2010)
37. Laddha, N.R., Thakare, A.: A review on serial communication by uart. International journal of advanced research in computer science and software engineering **3**(1) (2013)
38. Liu, S., Guan, N., Ji, D., Liu, W., Liu, X., Yi, W.: Leaking your engine speed by spectrum analysis of real-time scheduling sequences. Journal of Systems Architecture **97**, 455–466 (2019)
39. Lopriore, L.: Memory protection in embedded systems. Journal of Systems Architecture **63**, 61–69 (2016)
40. Lutz, R.R.: Analyzing software requirements errors in safety-critical, embedded systems. In: [1993] Proceedings of the IEEE International Symposium on Requirements Engineering. pp. 126–133. IEEE (1993)
41. Mankar, J., Darode, C., Trivedi, K., Kanoje, M., Shahare, P.: Review of i2c protocol. International Journal of Research in Advent Technology **2**(1) (2014)
42. Marquardt, A., Levenson, E., Tal, A.: Throwback attack: How the modest bowman avenue dam became the target of iranian hackers, `https://edition.cnn.com/2021/02/10/us/florida-water-poison-cyber/index.html`
43. Martin, L.: Cyber kill chain, `https://www.lockheedmartin.com/en-us/capabilities/cyber/cyber-kill-chain.html`
44. Martins, L.E.G., Gorschek, T.: Requirements engineering for safety-critical systems: overview and challenges. IEEE software **34**(4), 49–57 (2017)
45. Masrur, A., Kit, M., Matěna, V., Bureš, T., Hardt, W.: Component-based design of cyber-physical applications with safety-critical requirements. Microprocessors and Microsystems **42**, 70–86 (2016)
46. Meier, L., Honegger, D., Pollefeys, M.: Px4: A node-based multithreaded open source robotics framework for deeply embedded platforms. In: 2015 IEEE international conference on robotics and automation (ICRA). pp. 6235–6240. IEEE (2015)
47. Nair, S., De La Vara, J.L., Sabetzadeh, M., Briand, L.: An extended systematic literature review on provision of evidence for safety certification. Information and Software Technology **56**(7), 689–717 (2014)
48. Nyman, T., Ekberg, J.E., Davi, L., Asokan, N.: Cfi care: Hardware-supported call and return enforcement for commercial microcontrollers. In: Research in Attacks, Intrusions, and Defenses: 20th International Symposium, RAID 2017, Atlanta, GA, USA, September 18–20, 2017, Proceedings. pp. 259–284. Springer (2017)
49. Pinto, S., Santos, N.: Demystifying arm trustzone: A comprehensive survey. ACM computing surveys (CSUR) **51**(6), 1–36 (2019)
50. Preschern, C., Hörmer, A.J., Kajtazovic, N., Kreiner, C.: Software-based remote attestation for safety-critical systems. In: 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops. pp. 8–12. IEEE (2013)
51. Raghunathan, V., Chou, P.H.: Design and power management of energy harvesting embedded systems. In: Proceedings of the 2006 international symposium on Low power electronics and design. pp. 369–374 (2006)
52. Raspberry PI Ltd: Rp2040 datasheet. online (2024), `https://datasheets.raspberrypi.com/rp2040/rp2040-datasheet.pdf`
53. Robotics, A.: Shelly plus smoke, `https://www.shelly.com/de/products/shop/shelly-plus-smoke`

54. Sadeghi, A.R., Wachsmann, C., Waidner, M.: Security and privacy challenges in industrial internet of things. In: Proceedings of the 52nd annual design automation conference. pp. 1–6 (2015)
55. Shen, W., Norrie, D.H.: Agent-based systems for intelligent manufacturing: a state-of-the-art survey. Knowledge and information systems **1**, 129–156 (1999)
56. Singer, B., Pandey, A., Li, S., Bauer, L., Miller, C., Pileggi, L., Sekar, V.: Shedding light on inconsistencies in grid cybersecurity: Disconnects and recommendations. In: 2023 IEEE Symposium on Security and Privacy (SP). pp. 554–571. IEEE Computer Society (2022)
57. Song, C., Moon, H., Alam, M., Yun, I., Lee, B., Kim, T., Lee, W., Paek, Y.: Hdfi: Hardware-assisted data-flow isolation. In: 2016 IEEE Symposium on Security and Privacy (SP). pp. 1–17. IEEE (2016)
58. SparkFun: Sparkfun micromod rp2040 processor, `https://www.sparkfun.com/products/17720`
59. Stankovic, J.A.: Real-time and embedded systems. ACM Computing Surveys (CSUR) **28**(1), 205–208 (1996)
60. Studio, S.: Grove - gas sensor(mq2): Seeed studio wiki, `https://wiki.seeedstudio.com/Grove-Gas_Sensor-MQ2/`
61. Surminski, S., Niesler, C., Davi, L., Sadeghi, A.R.: Dma'n'play: Practical remote attestation based on direct memory access. In: Proc. of 21st International Conference on Applied Cryptography and Network Security (ACNS). Springer, Kyoto, Japan (jun 2023)
62. Systems, E.: Espressif systems - anual report 2023, `https://www.espressif.com.cn/sites/default/files/financial/Espressif%20Systems%202023%20Annual%20Report.pdf`
63. Team, P.: Px4/px4-autopilot: Px4 autopilot software, `https://github.com/PX4/PX4-Autopilot`
64. Techfun: Waveshare obstacle laser sensor. online (2024), `https://techfun.sk/de/produkt/laserovy-senzor-prekazok-waveshare/?lang=de&currency=EUR`
65. Walls, R.J., Brown, N.F., Le Baron, T., Shue, C.A., Okhravi, H., Ward, B.C.: Control-flow integrity for real-time embedded systems. In: 31st Euromicro Conference on Real-Time Systems (ECRTS 2019). Schloss-Dagstuhl-Leibniz Zentrum für Informatik (2019)
66. Wang, Y., Li, A., Wang, J., Baruah, S., Zhang, N.: Opportunistic data flow integrity for real-time cyber-physical systems using worst case execution time reservation. In: 33rd USENIX Security Symposium (USENIX Security 24) (2024)
67. Yagan, M.Y., Kayraklik, S., Kesir, S., Sumen, G., Hokelek, I., Basaran, M., Alexandropoulos, G.C., Basar, E., Cavdar, C., Arslan, H., Gorcin, A.: Fast network recovery from large-scale disasters: A resilient and self-organizing ran framework (2024), `https://arxiv.org/abs/2408.08609`
68. Zhou, J., Du, Y., Shen, Z., Ma, L., Criswell, J., Walls, R.J.: Silhouette: Efficient protected shadow stacks for embedded systems. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 1219–1236 (2020)