HERA: Hotpatching of Embedded Real-time Applications

Christian Niesler, Sebastian Surminski, Lucas Davi
University of Duisburg-Essen, Germany
{christian.niesler, sebastian.surminski, lucas.davi}@uni-due.de

***Abstract*—Memory corruption attacks are a pre-dominant attack vector against IoT devices. Simply updating vulnerable IoT software is not always possible due to unacceptable downtime and a required reboot. These side-effects must be avoided for highly-available embedded systems such as medical devices and, generally speaking, for any embedded system with real-time constraints. To avoid downtime and reboot of a system, previous research has introduced the concept of hotpatching. However, the existing approaches cannot be applied to resource-constrained IoT devices. Furthermore, possible hardware-related issues have not been addressed, i.e., the inability to directly modify the firmware image due to read-only memory.**

In this paper, we present the design and implementation of HERA (Hotpatching of Embedded Real-time Applications) which utilizes hardware-based built-in features of commodity Cortex-M microcontrollers to perform hotpatching of embedded systems. HERA preserves hard real-time constraints while keeping the additional resource usage to a minimum. In a case study, we apply HERA to two vulnerable medical devices. Furthermore, we leverage HERA to patch an existing vulnerability in the FreeRTOS operating system. These applications demonstrate the high practicality and efficiency of our approach.

I. Introduction

The Internet of Things (IoT) enables new use-cases and connectivity in many different domains: consumer electronics, household appliances, and even medical devices are increasingly connected to the Internet. While these devices get widely adopted, their security is often critical [1]. In particular, IoT devices in delicate areas, like medical applications, or safety-critical environments, like robotics, often suffer from programming errors and exploitable vulnerabilities. Recent studies show that pacemakers, i.e., critical medical devices implanted into patients bodies when their heart rhythm is too slow or irregular, suffer from security vulnerabilities [2]. In 2017, the US Food and Drug Administration (FDA) approved a firmware update for a pacemaker that was in use by 450k patients [3]. Although applying the firmware update was non-invasive, it introduced negative side-effects: the fall back to ventricular demand pacing, during reprogramming, may cause temporary symptoms for some patients. Furthermore, the manufacturer estimates a small rate of unpredictable device resets and failures. Such a device reset or failure bears the risk of serious health consequences, including death [4].

Network and Distributed Systems Security (NDSS) Symposium 2021
21-25 February 2021, Virtual
ISBN 1-891562-66-5
https://dx.doi.org/10.14722/ndss.2021.24159
www.ndss-symposium.org

Another challenge in this area concerns recent developments in the so-called Industrial Internet of Things (IIoT) area: systems initially designed without security in mind are now interconnected with each other and the Internet introducing a large attack surface [5]. Many of these systems require interaction with the physical world and, more importantly, have strict real-time constraints. These devices run jobs that must be completed within deadlines in order to work properly. Failing to meet these deadlines results in device malfunction and might even result in physical damage, e.g., regarding a vehicle control device, engine control unit, or robotics system. Security vulnerabilities on this type of devices can have disastrous consequences. For example, an industrial robot may fail with respect to safety and harm factory workers or cause a production halt with great financial loss [5]. Sophisticated attacks on industrial control systems are a real threat, as past incidents have shown, e.g., an attack against a German steel mill, which could not properly shut down its blast furnace [6]. Failures in individual control components resulted in an undefined state of the blast furnace and prevented its proper shutdown, leading to severe physical damage [7].

While there exist several methods to tackle the security problems of IoT devices (such as device hardening or software and hardware penetration tests, to name a few), a key mechanism to remove the vulnerability is *patching* or *updating* the device. Patching of computer systems is highly important and very common these days. Today, patch management and distribution of patches belong to the main and common tasks of IT departments [8]. Usually, updates are installed using an application or a special service. The update method has a high influence on the rate and speed of patch deployment [9]. In order to apply the patches, often a restart of the application or the system is required. In case of IoT systems, patches are often relatively large, i.e., monolithic blocks that have to be applied atomically because of the device architecture. As this leads to a loss of state and introduces a downtime of the system, these updates have to be scheduled using planned downtime and patching windows. Unfortunately, this opens a window of opportunity for an adversary to exploit a not-yet-updated device although patches are already available. At first glance, this does not seem to be a major issue as the time window is limited. However, recent studies showed that working exploits for a given vulnerability become public just on the same day or the subsequent day after the corresponding patch has been published [10].

Apart from the described time window, several devices do not accept any downtime, namely devices with real-time constraints such as plant control systems with real-time com-

munication between components [11] or pacemakers [4]. Real-time constraints often emerge from the application context and the importance of continuous operation to maintain a specific state. In case of the steel mill, its control systems cannot be halted arbitrarily, because they control physical processes that will continue. But industrial control systems can also be part of much larger networks: power plants are part of a international power supply network, connected via the power grid. The power grid needs to respond to varying energy demands and needs to ensure continuous power supply of factories, offices, and households. This requires coordination and planning in order to prevent power failures.

Hence, these devices require special treatment, i.e., the only reasonable but challenging option is to *hotpatch* these devices. Hotpatching means patching a program while it executes, without the need to stop or restart the program. This hotpatching process guarantees high availability and keeps the current state of the program at all times. To our surprise, hotpatching has been studied extensively in the context of traditional software [12] and server applications [13]–[15], but has received very little attention in the context of embedded systems.

Furthermore, hotpatching results in changes of code execution and may violate real-time constraints. Hotpatching is adding a patch, i.e., modifying the code, likely extending the execution time of a software fragment, with a certain amount of overhead. However, the overhead of the hotpatching process, causing a delay in the execution, is a risk since it is often not part of the scheduled program execution. This is not an issue in the context of traditional software [12]. However, within real-time systems, this can likely result in missing hard deadlines. Traditional software without real-time constraints is more generous regarding a single uninterruptible delay as it has no consequences due to a failed deadline. Hotpatching of real-time embedded systems strictly requires a small and predictable delay. A large uninterruptible execution delay is unacceptable for systems with hard real-time constraints, resulting in many vulnerable devices being unpatched. Especially in the most critical devices, the patching is stalled because interruptions are not possible. This opens the door to attacks on these systems.

Contributions. We propose HERA, the first vulnerability hotpatching system for embedded systems with real-time constraints. Our framework allows to perform security updates on real-time embedded systems during operation without any interruption or violation of real-time constraints. In contrast to existing approaches like Katana [16] and Kitsune [17], we do neither require dynamic linking nor previous changes to the target program. Instead, we exploit common built-in hardware debugging features readily available on the ARM Cortex-M mainstream processor to instrument all affected program locations with minimal overhead. By employing the on-board debugging unit for patch application, we also avoid any direct firmware image manipulation thereby allowing hotpatching to take place on devices with block-writable read-only memory; a common characteristic of embedded devices. As such, this allows us to induce minimal changes to the execution time as code redirection (trampoline insertion) as well as patch activation is encapsulated in atomic instructions on the hardware level.

Especially considering security patches, this has several advantages. It often eliminates the need for complex state transfer operations as most security patches do not modify the program state or its data structures. A good example are memory corruption vulnerabilities which can often be mitigated through additional bounds checking operations. This also reduces the amount of time for prior patch verification as no substantial modifications to the running system are added. Furthermore, it minimizes the additional resource usage as only minimal security-related enhancements are applied to the system.

While we allow the code to be updated in background, we achieve switching to updated code via an atomic instruction thereby minimizing time overhead and giving absolute predictability. We evaluate our solution theoretically through the examination of uninterruptible code sections, as well as through external switch time measurements. As our evaluation shows, the time overhead is minimal and constant. Furthermore, a case study with two medical devices and a hotpatch for an existing vulnerability in the FreeRTOS operating system shows general applicability. This makes HERA suitable for hotpatching of even hard real-time systems, i.e., the most critical real-time systems.

Hotpatches for the HERA framework can be developed on both: higher programming languages and then derived from the compiled binary using binary diffing, and directly on assembly level. While the first is the preferred way if source code is available, the latter allows to develop patches for closed-source devices.

To the best of our knowledge, our proposed system is the first hotpatching system that targets and solves the above described challenges like block-writable memory, minimizing the hotpatching overhead and spare resource usage in a single system. To summarize, this paper presents the following contributions:

- We present HERA, the first framework to allow hotpatching of real-time embedded systems without requiring any hardware changes.
- We give detailed guidelines how a hotpatch can be derived directly from normal binary files to allow the development of a hotpatch within existing workflows in Section V.
- In a case study in Section VII-A, we use this framework to patch security vulnerabilities of two real-time critical medical devices, a syringe pump and a heart rate sensor, as well as the FreeRTOS operating system.
- In our evaluation in Section VII-B, we measure the exact overhead of the framework. We show that the overhead of our framework is in the order of processor cycles and does not vary. This allows deployment of our system in real-time environments with hard deadlines.

II. Background

Since this paper addresses hotpatching of embedded systems with real-time constraints, we describe the main components and core concepts of such devices in this section.

ARM Architecture. With more than 160 billion units shipped, ARM is currently the most popular processor architecture [18], especially for IoT devices [19]. Furthermore, ARM architectures are frequently present in the real-time operation context.

For instance, a large share of PLCs (control units for industrial processes) are based on ARM [20]. Its microprocessor architecture is based on the reduced instruction set computer (RISC) design model [21]. By definition, it builds upon an instruction set with small and optimized instructions, instead of more complex and specialized instructions. ARM is an energy-friendly micro-architecture featuring asynchronous logic to reduce power consumption [22].

Flash Patch and Breakpoint Unit. Several ARM processors feature a hardware unit for debugging purposes which is called FPB (Flash Patch and Breakpoint) [23]. For example, the popular Cortex-M3 and M4 processors feature such a unit. The FPB unit allows to set hardware breakpoints. A comparator implemented in hardware is used to identify the correct breakpoint and then halt the processor or modify code memory. A full FPB unit comes with up to six comparators for hardware breakpoints [24].

This instruction replacement functionality is a typical feature of hardware debugging units. For example, the Tensilica Xtensa processor architecture features debug options that offer the same instruction replacement functionality as the FPB unit of the ARM Cortex-M3/M4 [25]. The popular microcontrollers ESP32 and its predecessor ESP8266 use such Tensilica Xtensa processors [26], [27]. ESP8266 and ESP32 chips are a popular platform for IoT devices: more than 100 million of such IoT chips have been shipped by the manufacturer [28].

Real-time Systems and Requirements. Computer systems can have different requirements due to expected or mandatory response times. In case a system has strict deadlines towards its response times, it is called a real-time system. There are different types of real-time expectations and system requirements, which are typically differentiated by their strictness. A system can have hard, soft and firm real-time requirements. A hard real-time system has deadlines, that must be met at all times. Otherwise, this has severe consequences. Typical examples for hard real-time systems are safety-critical applications like control units for car airbags. In contrast, the firm deadline allows for infrequent misses as the consequences are not catastrophic, but too many would degrade the system of its usefulness. A missed deadline has no further value in a firm system. The playback of music is such a firm system. A few missed bits of music will not degrade the entire system, but frequent misses will. A system neither hard nor firm is referred to as soft. The value of the information starts to decrease. A home heating system can be said to be a soft system. The system will tolerate frequent deadline misses as long as it will have a few timely temperature measurements. The value of past measurements decreases as the system operates on the most current measurements [29].

Real-time Operating System. Real-time capability is often ensured by the use of a real-time operating system (RTOS). The RTOS schedules tasks according to a given priority to meet hard and soft deadlines. It provides the capability to handle multiple threads and ensures given deadlines are met through a real-time scheduling algorithm, e.g., Rate Monotonic [30]. The RTOS is crucial to manage complex embedded applications and multiple tasks bounded by real-time requirements. Lightweight RTOS are a popular method to abstract from the bare-metal hardware while maintaining hard real-time capability. Although embedded devices are resource-constrained,

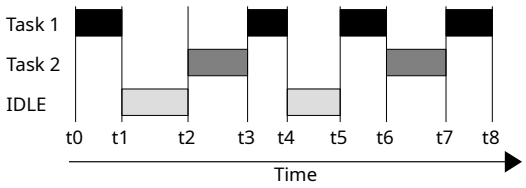


Figure 1: Example of a RTOS schedule.

there exists a wide variety of suitable operating systems [31]. RTOS exist in a variety of architectures, implementations and categories depending on the target system and its real-time properties, i.e., hard, soft or firm system [32]. The main component of any RTOS is a scheduler in control of the underlying hardware responsible to execute tasks with respect to their deadlines. Figure 1 shows a typical schedule of a RTOS. Tasks have different deadlines, thus different priorities. A lower priority task can be interrupted in favor of tasks with higher priority. This ensures that urgent tasks are not blocked by less important ones. High priority tasks are usually short and frequent, low priority tasks are usually long and infrequent. A system with enough available resources will complete all tasks before their respective deadlines [30].

One of the most popular embedded systems is FreeRTOS [33], and is suited for real-time purposes. We use FreeRTOS to implement and evaluate HERA in the case study described in Section VII-A.

Hotpatching. Hotpatching means patching a program while it executes, thus without restarting or rebooting the patched system or service. This requires to consider the current state of the program to circumvent negative side-effects. Hotpatching can be categorized by its changes, e.g., single code snippets, protocol changes, or larger changes in data structures [34].

Update Services. Updating and patching are commonly used to fix program errors and security vulnerabilities. In order to apply these patches, special applications are developed, which allow to manually or automatically install patches. These programs have the task to fetch the update, check the integrity, install, and verify the installation. IoT devices typically come with such update methods [1]. The FreeRTOS operating system features a module for over-the-air (OTA) updates [35]. With OTA updates, the device is able to download and apply the update by itself via the Internet without external interaction.

III. Problem statement & Challenges

In this paper, we present and evaluate a hotpatching approach that allows to patch resource-constrained embedded devices without violating real-time constraints. Such resource-constrained devices, which are interacting with the physical world, are a worthwhile target for adversaries with malicious intent. Those devices can cause potential harm to humans (safety) and even society (critical infrastructure), thus requiring regular updates to accurately respond to newly discovered vulnerabilities and novel attack techniques. Consider a modern automobile which includes a network of integrated electronic components and systems with different attack surfaces that can be targeted by an adversary. In an experimental analysis of a modern automotive, Koscher, Czeskis, Roesner, *et al.* demonstrate potential attacks and the large attack surface of

current cars, which will likely increase in upcoming years [36]. The medical sector moves into a similar direction. The health industry slowly shifts towards real-time disease monitoring for personalized healthcare, which can be seen by the evolution of wearable devices and their increasing application [37]. These wearable devices continuously measure physical signals like blood pressure or skin temperature and communicate with each other. Often, they are integrated into a so-called Body Area Network (BAN) [38], with vast communication mechanisms like Bluetooth, infrared, radio-frequency identification (RFID), and near-field communication (NFC) [37]. As any embedded device is powered by software, which may suffer from errors and vulnerabilities, security and privacy concerns arise. The current state of knowledge regarding such systems shows three major threats to medical devices: telemetry (wireless interface), software threats (alternating logic through vulnerabilities) and hardware/sensor interface threats (alter sensor signals) [38]. As a result, these devices require security updates in order to mitigate vulnerabilities. However, applying security updates is not always possible, if these devices may not be interrupted, or applying patches comes with side-effects [4]. In these cases, hotpatching is a viable solution to fix security vulnerabilities.

Challenges arise from the fact that these embedded devices must operate continuously and may not experience any interruption. The patching may not have any side-effects and the whole patching process has to be completely deterministic. This is especially important for highly-critical devices, e.g., manufacturing plants, but also for unattended patching processes, e.g., in the case of pacemakers. The patch itself must be validated to ensure correctness. As patches influence the execution of the system, this also changes the time required for the execution. While this is no problem for commodity systems, in case of systems with real-time capabilities changes in the timing can have severe consequences. Therefore, a predictable timing of the execution is mandatory for such systems. Furthermore, the patching process must work with minimal hardware requirements. Built-in embedded devices are manufactured and used in huge quantities. Hence, they are heavily optimized to minimize the needed hardware, in order to lower energy consumption as well as costs. As a result, these devices come with little resources, especially regarding spare hardware and memory. This makes standard A/B patching schemes or methods using a complete hardware switchover no practical solution. The hotpatching approach has to work using standard off-the-shelf hardware, so that it can be implemented into existing devices without hardware changes, as changes in the hardware usually require larger redesign and extensive development process.

Current research focus on correctness of patch application or minimizing resource consumption and downtime. Mugarza, Parra, and Jacob propose a "Quarantine-mode" to setup and monitor a live patch [39]. Zhang, Ahn, Zhang, *et al.* describe a software-based trampoline approach for hotpatching that minimizes the resource and energy use of embedded devices [40]. Real-Time Patch capabilities have been considered in [41], [42] and in [43], though the focus are the requirements and solutions for a full real-time patching process including state transfer and atomic switching. Frameworks for hotpatching such as Katana [16] or Kitsune [17] successfully showed the applicability of hotpatching and simplified development efforts for generation of hotpatches and applications. However, both require dynamic linking of application code. Embedded devices on the other hand often do not provide any means of run-time linking and use a static linking approach. No focus has been laid yet at hotpatching vulnerabilities in resource-constrained devices subjected to real-time constraints. Summing up, we identified several challenges that arise when hotpatching embedded real-time systems: (1) preserving hard real-time requirements, (2) compatibility with existing hardware, (3) memory architecture of embedded devices, (4) concurrency issues when applying patches.

The HERA framework we introduce in this paper addresses all these challenges. It works on standard Cortex-M3/M4 processors and relies on built-in debugging features. The memory overhead is minimal, as only the patch itself and a list of several memory addresses have to be stored. The actual switching process to enable the hotpatch is an atomic instruction on processor level. Using the built-in FPB unit, a processor instruction is replaced on-the-fly. This instruction is executed atomically, i.e., in one step and cannot be interrupted. As a result, no unexpected side-effects can occur. Since this is all done within a single instruction, the execution time is completely deterministic, which is mandatory for real-time systems with hard deadlines.

IV. Assumptions and Attacker Model

Assumptions. We assume a real-time system that features all types of deadlines, i.e., hard, firm, and soft deadlines. Furthermore, we assume a real-time operating system (RTOS) architecture. The RTOS itself ensures proper real-time operation. That is, it runs all tasks correctly with respect to deadlines. An update is available, either as source code or binary. The system is equipped with sufficient resources allowing to run the update within its real-time requirements as well as a low priority or idle-time task to download the patch to the system writable memory, i.e., RAM. Thus, no deadlines are missed. An updater service, that is able to download the patch as well as a secure update mechanism are assumed to be in place. The updater then triggers an atomic (hardware) switchover to enable the patch. The concrete implementation of the updater service is out of scope as this is heavily dependent on the target use-case. For example, the updater service should regularly check for updates and download these via an external interface. Before applying the patch, the integrity and authenticity has to be checked, as well as the transmission and the source of the patch has to be verified. Thus, the update mechanism itself is secure. As elaborated in Section II, software updates and corresponding updaters are commonly used in practice. Note that the real-time operating system FreeRTOS already includes an updater service that can be adapted to specific needs [35].

Attacker Model. We assume a remote attacker who is able to exploit an arbitrary vulnerability due to a programming mistake. A typical error is a buffer overflow error due to a missing bounds check. An attacker is able to exploit the vulnerability to overwrite parts of the memory and launch a run-time attack. We assume the root cause can be mitigated by introducing or replacing a code block, e.g., by introducing further checks or validating inputs. The attacker does not have physical access and can only attack the device remotely. Furthermore, we assume there is a secure updater available, that the attacker cannot manipulate or sabotage.

V. Overview of HERA

In this section, we introduce the design of HERA, the first hotpatching framework for embedded systems with real-time constraints. First, we analyze existing hotpatching strategies and present the high-level approach of our hotpatching strategy (cf. Section V-A). Thereafter, we present in detail the different components of HERA (cf. Section V-B).

A. On Hotpatching Strategy

There exist three main hotpatching strategies, namely (1) relocation of linked binaries, (2) instrumentation of the binary, and (3) A/B update schemes.

Hotpatching Relocatable Executables. This approach requires modification of the binary linking data structures. Dynamically linked binaries reference code that is not part of the compiled library and is loaded during run-time. This is often the case for libraries which are shared among different applications and common practice in modern operating systems like Windows or Linux without strict real-time requirements. The operating system holds a data structure with symbolic links to this shared code piece and resolves those links during run-time. The implementation of hotpatching with relocatable executables is straightforward in case dynamic linking is supported by the underlying operating system. With dynamic linking only a symbolic link needs to be adjusted during run-time to apply a hotpatch. Thus, whole components can be exchanged and the number of total active patches is unlimited. However, in embedded systems, due to real-time constraints, binaries are often linked statically and the operating system does not support dynamic linking [44]. This is because run-time linking creates overhead and may deteriorate predictability, i.e., a key property of a real-time system. In fact, the popular FreeRTOS real-time operating system is linked statically [45]. As such, popular patching frameworks such as Katana [16] and Kitsune [17] are not applicable as they both require dynamic linking.

Instrumentation-Based Hotpatching. This approach requires modification of the program execution. The most common technique is a direct rewrite of the firmware image (i.e., the running executable) by redirecting the execution through a so-called trampoline. A trampoline is a single instruction, e.g., a branch or jump instruction, inserted at an arbitrary point redirecting the original control-flow of an application. Another technique is dynamic binary translation. This approach is based on executing the application inside of a translator component enabling exchange and instrumentation of instructions on-the-fly [46]. However, this approach induces a high overhead and requires high computing resources which resource-constrained embedded devices, especially in a real-time context, do not provide.

A/B schemes. Another hotpatching method is the concept of a so-called A/B update scheme [39], [47]. This scheme consists of two instances: one instance is actively running; the other instance can be updated. At a dedicated point in time, a switchover from the active to the newly updated instance is performed. The main advantage of this strategy is that the update can take place completely independent from the active instance. A/B updated schemes are widely used in practice, e.g., over-the-air updates for Android devices [48] and Espressif ESP32 microcontrollers [49]. However, on the downside, this approach requires doubling the memory to hold both instances, and a dedicated management unit that allows the update of the inactive instance and performs the switchover. The real-time capability of a A/B scheme depends on the management unit. Common A/B solutions like in Android [48] require a full device reboot. As embedded systems are often equipped with minimal hardware and memory and in addition bound to real-time constraints, this is not a viable hotpatching strategy for these devices. An even more sophisticated method is to use a so-called complete switchover: multiple instances are executed in parallel and a single instance can be taken offline and patched, while the other instances continue to run. Again, this results in a large overhead as two instances have to be executed and managed by a dedicated unit at the same time [50].

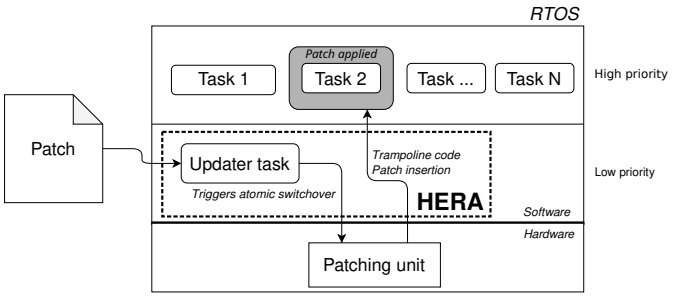
HERA Hotpatching. Our hotpatching strategy is specifically tailored towards embedded systems with minimal computing power and memory. We introduce a hardware-assisted hotpatching strategy which allows arbitrary code insertions through so-called trampolines dynamically inserted into the program binary. Our system is based on a standard real-time operating system (RTOS) which ensures that all deadlines are met through proper scheduling. The activation of a patch usually requires a so-called atomic patch activation, i.e., the activation process must not be interrupted to preserve consistency of the system. Leveraging hardware allows us to perform this atomic patch activation very efficiently thereby making it suitable for hard real-time constraints.

Our approach enhances the instrumentation-based hotpatching approach without requiring binary instrumentation and additional software layers. The key idea is to replace instructions during run-time using the on-board debugging unit, which is readily available on commercial off-the-shelf processors. Using the hardware breakpoint capability of the on-board debugging unit, trampolines can be inserted on-the-fly. The amount of available trampolines is dependent on the board's number of hardware breakpoints.

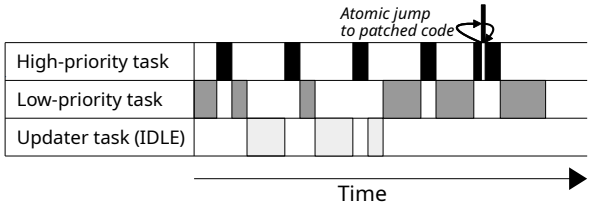
Modern microcontrollers feature a debugging unit allowing developers to insert these so-called breakpoints. A breakpoint allows developers to stop and restart the execution of the CPU at arbitrary points. This is used to inspect the CPU and peripherals, investigating software issues. However, a debugging unit can also implement additional functionality such as the capability to exchange a single instruction. For instance, this feature is implemented in the FPB debugging unit of ARM Cortex-M3/M4 processors [23]. This unit can temporarily stop the CPU, drop the fetched instruction, and fetch a new instruction. Hence, we can leverage this behavior to redirect the control flow to newly deployed hotpatches. Since the redirection to patched code is performed hardware-assisted, the overhead is negligible and constant. A detailed analysis of the performance of the FPB unit is performed in Section VII-B.

B. Design

As alluded to earlier, we have designed a hotpatching solution based on existing built-in hardware features used for debugging. A high-level overview of our HERA architecture



(a) Component View: The low-priority updater task modifies hardware to enable a higher priority task patch in real-time through atomic jump.



(b) Sequence View: The patch is added with a hardware supported trampoline and properly scheduled through the RTOS.

Figure 2: High-level architecture and functionality of the HERA framework.

and its functionality is depicted in Figure 2. Figure 2a shows the combination of hardware patching unit, real-time operating system (RTOS), and the HERA framework. An exemplary update and patching process is outlined in the sequence diagram in Figure 2b.

As shown in Figure 2, there are two main steps to patch the system: (1) patch preparation and (2) switchover. The patch preparation covers the updater mechanism and patch processing steps such that the patch can be reached without any issue by a trampoline. The switchover covers the notification of the hardware patching unit (atomic switchover) and the hardware-assisted trampoline insertion during run-time.

Our system is based on a RTOS which ensures that all deadlines are met through priority-based scheduling. The RTOS is running different tasks with different priority levels. Independent of the priority, a task may suffer from a vulnerability that needs to be quickly mitigated through a hotpatch. This hotpatch needs to be received, verified, and properly instantiated onto the embedded system. The updater task is executed in background and interrupted by higher-priority tasks to avoid any disturbance on time-critical tasks. Hence, we do schedule the updater task with either low priority or within the idle-time. The updater task is responsible for the patch preparation (1) as well as the switchover (2) step. The updater distinguishes between patch code, i.e., mitigation code precompiled for the target architecture and meta-information like patch size or patch insertion point. Meta-information includes all required information to apply the patch code. Both patch code and meta-information are packaged within a patch format. In particular, the updater copies the patch code to a patch slot in RAM and prepares the switchover by adding meta-information like patch insertion point and trampoline to respective data structures (FPB configuration). Once patch processing is finished, the updater task notifies the hardware-unit with an atomic instruction to activate the patch. After this hardware switchover, the FPB starts to insert trampolines through its hardware breakpoint functionality. Figure 2b shows how the trampoline insertion and patch execution is handled from the scheduler point of view. The RTOS is properly scheduling all its tasks to meet the deadlines. The patch is inserted on-the-fly by exchanging a single instruction of the task with a trampoline. Hence, the patch code itself is part of the normal scheduling process, just handled like any other code fragment of the same task. From a high level perspective, this approach resembles an in-place patching based on a trampoline [40]. The crucial difference is that the trampoline insertion is efficiently performed through commodity hardware and without interrupting the ordinary execution of the currently running tasks.

(1) Patch Preparation. This step prepares for an *atomic switchover* or *atomic patch enable*. All actions taken in the preparation step are uncritical and do not influence the application as they take place in a low priority or IDLE task. Recall, as previously described in Section II and shown in Figure 1 an RTOS system commonly uses priority-based scheduling. Given enough resources, this strategy will meet all deadlines of the scheduled tasks. The IDLE state is only entered when there are no concurrent tasks scheduled. For some architectures a direct control-flow redirection to RAM is not possible. Therefore, we use a preconfigured trampoline in ROM and a so-called dispatcher, which chooses the appropriate patch by checking the control-flow information stored on the stack. The preparation step includes the following steps: (1) Receive the patch from an external source, (2) verify the patch by checking its integrity and origin, (3) parse the meta-information from the patch format, (4) store the patch in selected memory area[1] (RAM), (5) add a new entry to the dispatcher[2], (6) configure the patching unit with insertion point and corresponding branching instruction (trampoline configuration), (7) trigger the atomic switchover.

(2) Switchover. After the patch preparation, the atomic switchover and the trampoline insertion process have to be performed. The patch is enabled by setting a hardware breakpoint [23]. This breakpoint, typically used for debugging, can be activated by a single assembly instruction, writing a designated hardware register. A single assembly instruction on a CPU is atomic by definition. The breakpoint uses the preconfigured data (insertion point and branching instruction) to perform the insertion of the trampoline during run-time. As shown in Figure 3, the debugging unit continuously monitors all instructions the CPU is executing. Each configured breakpoint has a comparator that triggers an event in case the CPU reaches a preconfigured memory location, identified by a memory address. This memory location is the insertion point for the trampoline. A breakpoint event usually shifts control to an external debugger. For the sake of hotpatching, we replace the breakpoint with a trampoline, i.e., branching instruction, thus inserting the patch. This process runs continuously without

[1]A special memory area is reserved as designated patch data storage.

[2]Only required on architectures with no direct control-flow redirection from ROM to RAM.

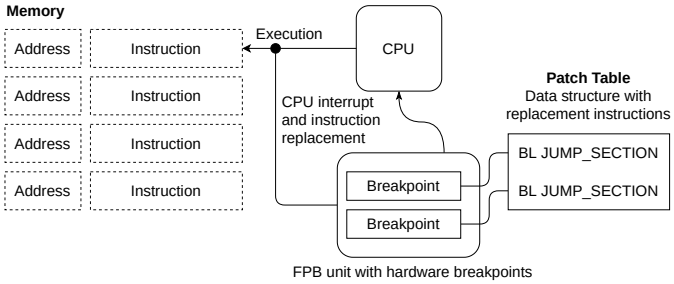


Figure 3: Hardware Architecture Overview: Hardware supported run-time trampoline insertion through debugging unit (FPB).

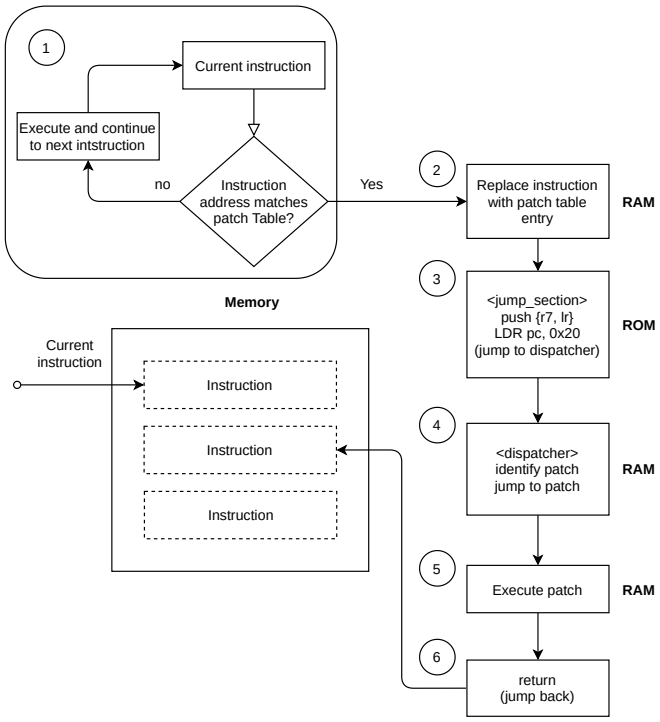


Figure 4: Patch Process Flow: Run-time trampoline insertion through debugging unit (FPB).

impacting the behavior of the processor. As breakpoints can be configured independently from each other, new trampolines can be added and deleted without disabling previous patches. This is important in order to support hotpatching different vulnerabilities that might affect a system over time.

Since different processors vary in programming models for their built-in debugging unit, we will focus on the ARM FPB unit as our proof-of-concept implementation is using it. Figure 3 shows the hardware perspective of our patching solution. The FPB unit has a fixed number of breakpoints (comparators) that continuously monitor the currently executed instruction. The debugging unit is attached to the CPU, which enables it to interrupt the processor's execution and replace the fetched instruction with another instruction, i.e., the trampoline instruction. Each breakpoint references an instruction, it will load instead of the aborted instruction on a breakpoint hit. These instructions are stored in a data structure called patch table allocated in the microcontroller's RAM region. Figure 4 covers the process flow of the trampoline (patch) insertion mechanism, which consists of: (1) continuous monitoring of the currently executed instruction, (2) instruction replacement on breakpoint hit, (3) control-flow redirection to a dispatcher, (4) dispatcher redirection to the appropriate patch, (5) patch execution, and (6) return and continued execution.

Figure 4 shows the process of on-the-fly breakpoint insertion as well as the interdependence between RAM and ROM memory. This interdependency is necessary as embedded system firmware is placed in read-only memory, which is only writable during programming. Some microcontrollers allow for page-rewrites during run-time. However, this would likely affect a significant portion, usually a page, of the stored firmware and thus the running RTOS. Furthermore, the process of page writing and erasing is very energy consuming [47]. As a consequence, direct firmware manipulation is not feasible. Therefore, we allocate our hotpatch to RAM which allows us to perform byte-wise writing in contrast to block-writing on page granularity to ROM. The *jump_section* (ROM) in Figure 4 is used to redirect the code execution to the stored patch in RAM via a small patch dispatcher. The *jump_section* and *dispatcher* are needed as our chosen proof-of-concept platform does not support direct branching from ROM to RAM. The RAM is out of range for a branch instruction.

Limitations. Our main scope are security patches for memory corruption vulnerabilities which typically require adding further checks replacing code parts. Furthermore, HERA can also be used to add further functionality, e.g., by adding new code parts. In case of complex updates, which require change in state or data structures, the HERA framework needs to be extended to allow state transfer, i.e., the migration of information from old to new data structures, thus ensuring a consistent program state. This is not necessary for typical security patches, as these are typically small and featureless [51]. Hotpatches should be as simple as possible, as they have to work during run-time. Different devices have varying memory content and state, hence changing values of variables and data structures is a delicate task.

HERA hotpatches are not reboot-safe as they are not stored in persistent memory. Hence, if a device is rebooted unexpectedly, the hotpatch is removed. This is due to the fact that writing persistent flash memory is often not feasible (see Section V-B). In order to cope with reboots, the updater has to be modified to automatically check for hotpatches and apply them upon every start. If indeed non-volatile writable memory is available, HERA can easily be extended to support persistent hotpatches.

The number of different patches that can be applied is limited by the number of breakpoints that the processor can keep. It is possible to fix multiple issues using a single breakpoint, as the breakpoint is just the entry point to the patch. So it is possible to replace a larger code segment, tackling several vulnerabilities using just a single breakpoint. Furthermore, the HERA framework focuses on hotpatching, i.e., fixing security issues until the next regular possibility for full system patching. After applying the full system patch, all breakpoints are free again for further hotpatches.

VI. Implementation

Based on the assumptions and design decisions made in Section V, we implemented a prototype of HERA as a library which uses the FPB unit of the ARM Cortex-M4 platform. The NUCLEO-F446RE development board from STMicroelectronics serves as a Cortex-M4 platform reference. With an ARM Cortex-M4 CPU and only $512\,\mathrm{kB}$ of flash memory and $128\,\mathrm{kB}$ of SRAM [52], it represents a typical Cortex-M based embedded system. Any embedded system such as the used NUCLEO-F446RE is constrained by different properties depending on the chip or the application context. A categorization of constraints is given by Bormann, Ersue, and Keranen [53]. In Section VI-A, we discuss the configuration of a debugging unit for run-time hotpatching and provide a proof-of-concept implementation. In Section VI-B, we provide guidelines how hotpatches can be created within the development cycle. Section VI-D discusses necessary preconditions for successful patching.

A. HERA Library

The core of our hotpatching approach is the use of debugging units integrated into modern microcontrollers, cf. Section V-B. These units often provide a code remapping mechanism in addition to their breakpoint functionality (Section V-A). The code remapping feature allows to drop the execution of an instruction at a previously selected memory location (memory address) and load another instruction from memory. Hence, control-flow can be manipulated by exchanging an instruction with a branch. With the aid of the code remapping mechanism, trampolines can be inserted during runtime. The configuration of code remapping generally requires the same information as a breakpoint: (1) insertion point, (2) trampoline, and (3) hardware breakpoint. The insertion point specifies the memory address, where the breakpoint should halt the CPU execution and perform the trampoline insertion. The trampoline is a branch instruction, that redirects control-flow from the insertion point to the patch. The hardware breakpoint of the debugging unit is used for runtime trampoline insertion. The debugging unit has a limited amount of hardware breakpoints, which represent a patch slot in HERA.

Since debugging units can differ in their programming model, we opted to choose the well-known ARM embedded platform [18], [19] for our prototype implementation; specifically, the ARM Cortex-M4 and the NUCLEO-F446RE board. The integrated debugging unit is called FPB (Flash Patch and Breakpoint) [23] and includes a code remapping feature. Note that HERA can be implemented on any architecture that provides a debugging unit with code remapping capability; only the hardware-related configuration needs to be adjusted to the target architecture. For industrial-grade usage the patch and dispatcher loading process need to be extended. To achieve code remapping, the FPB unit requires the same input as any other debugging unit: insertion point, trampoline, and hardware breakpoint. This input is stored together with the configuration of the FPB unit in two different data structures: the first one is a control structure located at an internal memory address outside of ROM and RAM. This data structure is used to configure and control the breakpoint and code remapping feature. It is also used to reference the second data structure, called patch-table, holding our trampoline instructions. To use the FPB debugging unit for hotpatching, we implemented a C-library to perform the necessary FPB unit configuration and prepare the load of the patch. The library provides the following functions:

fpb_init. This function initializes the FPB unit by preparing and properly referencing the required data structures. Furthermore, it checks the availability of a FPB unit on the given board. A successful initialization confirms the FPB capability of the used hardware unit.

fpb_enable. This sets the global enable bit of the FPB unit. If not set, the unit will ignore any configured and enabled breakpoint.

enable_single_patch. This function creates the trampoline through the code remapping feature and breakpoint usage. The trampoline consists of a branch instruction performing a control-flow redirect from the insertion point to the patch entry point. The branch instruction can be either supplied with the patch or a branch instruction calculation can be implemented based on the given offsets. This function sets the breakpoint to the trampoline insertion point and defines the breakpoint behavior as code remapping. Finally, it enables the hotpatch through a atomic switchover using a single register write.

load_patch_and_dispatcher. The function implements the patch loading and preparation process. A basic dispatcher (see Section V-B) is loaded and modified with the given knowledge of patch locations. Afterwards, the patch is copied to RAM.

The entire patching process runs in a low priority task, managed by the FreeRTOS [54] system. The atomic switchover is interruptible as it consists of a single assembly instruction, which cannot be interrupted by definition, see Section V and Section VII. As a consequence, the embedded system cannot miss any deadline as proper real-time scheduling stays in place. Furthermore, no further concurrency handling is required.

The implemented program flow can be summarized as follows: (1) initialize the FPB unit and check board compatibility, (2) load the dispatcher to RAM, (3) copy the patch to RAM (dedicated patch region), (4) modify the dispatcher with a patch location entry if required, (5) enable the FPB unit globally, (6) configure and enable the instruction matcher, (7) the patch flow is active.

B. Patch Development

Next, we describe how a patch is developed for hotpatching. This especially requires code changes in the embedded firmware to effectively mitigate the discovered vulnerability. In the software development cycle common within the industry [55], developers mitigate the vulnerabilities by constantly updating and modifying the software, creating patches. It is also a good practice to separate feature updates from security updates. Given our trampoline architecture, hotpatches can be obtained by means of static code comparison of the unpatched and patched firmware version. Creating a hotpatch is typically straight-forward and requires little additional work: the developer performs the static code comparison of the binary images and extracts the patch based on the instructions that have been changed or inserted. Development tools such as IDA Pro[3] or

[3] https://www.hex-rays.com/products/ida/

Listing 1: CVE2018-16601 source-level patch

```
1 if ( (uxHeaderLength >
2 (pxNetworkBuffer->xDataLength
3    - ipSIZE_OF_ETH_HEADER )) ||
4 (uxHeaderLength < ipSIZE_OF_IPv4_HEADER))
5 { return eReleaseBuffer; }
```

Radare2[4] already support the generation of an instruction difference on binary level. These development tools also directly indicate the required trampoline insertion points as they show the memory addresses of the instructions where the patched firmware differs from the unpatched version. The trampoline can be either directly derived from the insertion point and the future patch location within the devices RAM, or a dynamic instruction calculation can be implemented within the updater task (see Section V-B). A fully developed updater could also dynamically choose a free patch slot (hardware breakpoint). The developer may add slight modifications to the patch. For example, restoring some stack values and registers like the frame pointer. These modifications are currently only required because of the dispatcher solution, see Section V-B, which has been implemented to overcome the limited range for branching instructions on our proof-of-concept platform. However, these modifications can be reused along all hotpatches.

In general, the developer can follow these steps for hotpatch development: (1) create and compile a patched version of the application, (2) compare the patched and unpatched firmware on assembly level, (3) extract the differences and add modifications, (4) determine the insertion point and package the patch for application.

C. Example of Hotpatch Development for FreeRTOS

This section presents the development process of the FreeRTOS case study from Section VII-D, following the development guidelines described in Section V-B and Section VI-B.

FreeRTOS 9.0.0 is vulnerable to CVE2018-16601 [56]. The reason for this vulnerability is a missing bounds check on the length of the IP Header [57]. The vulnerable function is called *prvProcessIPPacket* and is located in the file *FreeRTOS_IP.c*. A comparison with the current FreeRTOS 10.0.3 shows the bounds check that was implemented to mitigate this vulnerability. Listing 1 lists the source code of the patch.

We backported the source-level patch into the vulnerable FreeRTOS version and compiled two binaries, with and without the patch. With the use of Radare2, we performed a binary diff that reveal the assembly instructions in Listing 2 that perform the bounds check. In a binary diff, newly inserted assembly instructions are marked with a '+' symbol. This indicates the trampoline insertion point. To cope with limitations of the FPB unit, the insertion point is shifted to the first precedent instruction at a 4-byte aligned memory address.

As explained in Section V-B, some instructions have to be adjusted. In case the patch should continue directly after the trampoline, the patch requires a return to the dispatcher and *jump_section*. In case the branch target is outside the patch,

[4]https://www.radare.org/n/

Listing 2: Binary diff between the vulnerable and the patched FreeRTOS binary

```
1   lsls r3, r3, 2
2   and r3, r3, 0x3c
3   str r3, [r7, 0x24]
4   + ldr r3, [r7]
5   + ldr r3, [r3, 0x1c]
6   + subs r3, 0xe
7   + ldr r2, [r7, 0x24]
8   + cmp r2, r3
9   + bhi 0x801619a
10  + ldr r3, [r7, 0x24]
11  + cmp r3, 0x13
12  + bhi 0x801619e
13  + movs r3, 0
14  + b 0x80162ca
15  ldr r3, [r7, 4]
```

the offset between RAM and ROM is typically too large to be coded directly into a branch instruction. This control-flow redirection is the reverted case of the *jump_section* control-flow redirect. The branch is performed by manipulating the program counter register, which contains the memory location of the next instruction to execute.

D. Patch Application

After the patch has been developed, it is transmitted processed, and finally activated on the target device. This is performed by the updater service which executes as a low-priority task to handle the tasks described in Section V-B. Apart from the actual patch, a patch file also contains an insertion point, a trampoline, and a hardware breakpoint. All of this can be combined into a simple binary file format. For both, the updater and the patch format, already implemented solutions exist. These solutions can be adopted for industrial usage as such update methods are common among IoT devices [1]. The updater fetches and processes the patch file. Next, the updater triggers the atomic switchover once all FPB and patch preparation is finished, i.e., sending the patch activation signal. This is done by the library functions *load_patch_and_dispatcher* and *enable_single_patch* as described in Section VI-A. As the updater task is preemptable and patch activation is atomic, no update time needs to be defined or selected. The patch can be activated as soon as possible without negative impact on the application.

E. Example of Hotpatch Application for FreeRTOS

We implemented an updater service as described in Section VII-D and Section VI-D. The updater task handles the patch preparation and performs the atomic switchover. Afterwards, the FPB unit continuously monitors the program execution and halts the CPU at the predefined trampoline insertion point. Thereafter, the breakpoint drops this instruction and replaces it with the trampoline. Those steps are described in detail in Section V-B. The resulting instructions of the process after the run-time code-remapping are visualized in Listing 3. In line 4, the canceled instruction is replaced by a branch to the *jump_section*, i.e., the patch.

The control-flow is redirected to the *jump_section*, a special function serving as entry-point of the patch. This entry point

Listing 3: Instruction replacement on breakpoint hit

```
1   push {r7, lr}
2   [...]
3   ldrb r3, [r3]
4   lsls r3, r3, 2 bl jump_section
5   and r3, r3, 0x3c
6   [...]
```

Listing 4: Dispatcher in the FreeRTOS case study

```
1   push {r3}
2   ldr.w r3, [0x20000052]; dispatcher entry
3   cmp lr, r3
4   beq 0x20000074; patch location
5   [...]
6   pop {r3}
7   pop {r7, pc}
```

is required as no direct branch from ROM to RAM is possible on the used hardware, as described in Section V-B. The *jump_section* manipulates the program counter (PC) shifting execution to the dispatcher. The dispatcher checks the origin of each control-flow redirection, indicated by the LR register. This register contains the return address, which is stored automatically by each branch and link (BL) instruction. A comparison between the current origin (LR) and each possible origin (dispatcher entry) can determine the patch to execute, which is shown in the resulting dispatcher code presented in Listing 4.

VII. Evaluation

To demonstrate the practicability of our hotpatching framework HERA, we conducted diverse measurements and case studies in our evaluation. In particular, we evaluate HERA based on the open-source implementation of two real-world medical devices (a syringe pump and a heartbeat sensor), which both provide critical functionality and real-time requirements. We port the two programs to our target hardware platform and even integrated the electrical part of both devices, i.e., actuators, sensors, and displays to facilitate evaluation of HERA based on a real and representative physical setup. The representative test setup is shown in Figure 5. As no vulnerability is known for the open-source implementation of both devices, we implanted typical memory corruption vulnerabilities (out-of-bounds write) allowing a remote attacker to compromise the devices.

As the HERA framework patches systems on binary level, hotpatch development can be conducted on basis of binaries as well as using source code. However, developing patches for real-world embedded systems rises several challenges. Typical embedded devices do not come with the possibility to easily modify the software they run. Porting the software to development hardware leaves manifold issues due to the close interactions with the hardware. While it is straightforward to develop patches using source-code or a binary with minimal changes, this is more complicated if a binary contains a multitude of changes as more than a single bug are fixed. For the measurement study, exact knowledge of the internal software and debug features of the processor are

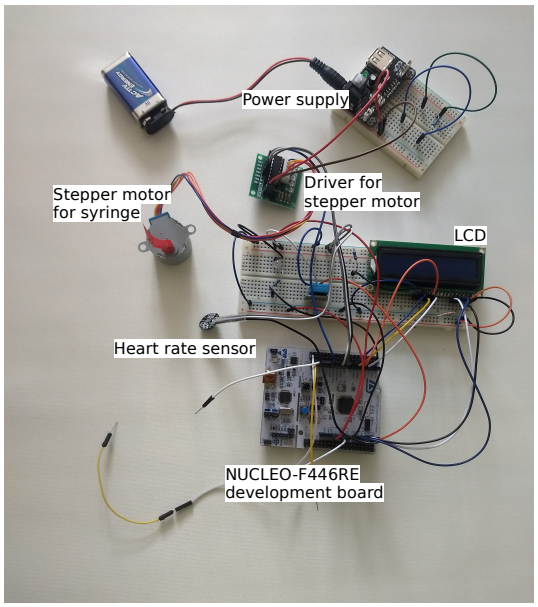


Figure 5: Photo of the implemented setup.

required. Therefore, our case study bases on two open-source medical devices, where both source code and hardware setup are publicly available and well-documented.

Embedded devices often execute software written in memory-unsafe languages, likewise our example projects. Due to the manual memory management, memory errors are prevalent and often lead to critical software vulnerabilities [58]. Programming errors belong to the most common causes for vulnerabilities with code execution capability for devices with local, remote or internet access [59]. A recent example of a critical real-world vulnerability is referred to as Ripple20 [60]. Ripple20 is a series of vulnerabilities discovered in a small TCP/IP stack affecting a wide range of embedded devices. The series include multiple vulnerabilities that are caused by improper input or parameter validation. Ripple20 proves that out-of-bound write and read vulnerabilities are a recent and major threat to embedded landscape.

Using our out-of-bounds write vulnerabilities (improper validation), we developed return-oriented programming (ROP) attacks against each application. Thereafter, we developed patches for both applications, of which each inserts a bounds-checking preventing the out-of-bounds write rendering the ROP-attack useless. Furthermore, we also performed a measurement study on these example applications, quantifying the overhead induced by hotpatching. These measurements clearly demonstrate that HERA is highly efficient inducing negligible overhead.

A. Case Study: Medical Devices

Next, we describe our two safety-critical medical devices, namely the syringe pump and the heartbeat sensor. Both devices build upon the popular Arduino platform. They are a good target for our evaluation as they both need to strictly meet hard deadlines to guarantee the patient's health.

Syringe Pump. A syringe pump is a medical device used by an external monitoring or sensor system to inject intravenous

medication to a patient in regular time intervals. It is critical to inject the correct amount, with the correct injection rate at precisely defined points in time. While commercial syringe pumps are typically expensive, there has been interest in developing syringe pumps as open source, allowing to reproduce such medical devices [61]. This enables manufacturing of such critical, but possibly life-saving medical devices, even though industrial-produced syringe pumps are not available, e.g., due to cost reasons or sudden incidents like catastrophes.

For the evaluation of HERA, we used an open-source implementation of a syringe pump [61]. This project was for example also previously used by Abera, Asokan, Davi, *et al.* to show feasibility of an implementation for remote attestation of embedded devices [62].

Heartbeat Sensor. The second example is a heartbeat sensor [63], which is a timing-critical sensing device. Accurate measurements depend on the real-time sampling rate of the sensor. If the timing varies, this massively influences the quality of the measurements. Measurements can easily become unreliable due to noise, peaks, or a variable signal, thus continuous sampling is required. Security is as important as high availability. An incorrect injected amount of medication or a false heart-rate caused by malicious manipulation can have lethal consequences for patients, who rely on the correct function of those devices.

In order to use our framework, we had to port both applications from the Arduino to the ARM Cortex-M4 platform and enable it to run on the FreeRTOS operating system. The Arduino platform is a popular platform for open-source projects as well as affordable medical system due low-cost and extensive software support [64]. Recall that the Cortex-M4 is a processor that readily features a FPB unit. For the implementation, as well as further measurements, we used the setup shown in Figure 5 and described in Section VI.

Exploitation Steps and Requirements. To create a proof-of-concept (POC), we implanted the following security vulnerability into both applications: a missing bounds check in the command receive function which allows an attacker to perform a buffer overwrite. Note that Cortex-M4 processors feature a Harvard bus architecture for simultaneous data and instruction fetches, but have a unified memory space. This means that program code, data, and peripherals share the same memory space [24, Section 3.1.1]. A Harvard architecture usually prevents direct code insertion [65], but since it is only applied at the micro-architecture level the shared memory space still allows for traditional code injection [66] as well as return-oriented programming (ROP) [67]. As mentioned in Section IV, we assume a remote attacker with no direct physical access, yet able to communicate with the device through one of its external interfaces. In this particular scenario, the attacker has the capability to use the serial interface of both applications hosting the command interface. In normal operations, the device can be controlled via the serial interface. In our scenario, allows this interface an attacker to exploit a buffer-overwrite/read error. We developed two exploits, one for each device. With these exploits, a ROP-attack is launched, manipulating the program execution in a way so that the syringe pump delivers a large amount of fluid independent of its configuration. The ROP-attack on the heartbeat sensor stops the continuous heartbeat measurement, altering the received sensor signal to a falsified value.

Patching the Vulnerabilities. We developed patches for both applications to add the missing bounds checks: if a message is too large for the buffer, it is dropped, as the applications both have fixed-size commands. Patch development has been carried out according to the description in Section VI. We created a hotfix for the application, developed in the C programming language. Afterwards, we created a binary diff between the compiled unpatched and patched version at assembly level. Then, we added minor modifications to reconstruct the stack frame pointer and adjusted the jump instructions. In our proof of concept, we use a pre-configured preemptable RTOS task to directly load the patch into RAM. The transfer to RAM and patch activation are triggered through a button press.

Patch Task. The implementation of the case study is based on an interruptible patch task originating from a preconfigured patch. In a real-world scenario the patch would be received through a secure update mechanism from an external interface (see Section II).

The patch task transfers the patch to a dedicated patch region located in RAM. The use of RAM for patch storage is often required since embedded devices provide only block-writable memory and Flash (ROM) modification is either difficult or not possible. Furthermore, the hotpatch is intended as a temporary solution to protect the device until the next, but likely distant, full firmware update opportunity. After copying the patch to the correct patch region, the patching task conducts all required preparations of the patch. Finally, using the FPB unit, the patch is activated via an atomic switch. This procedure is a single instruction on processor layer thereby it ensures minimal overhead and prevents the patching process from being interrupted.

FPB Atomic Switch. The FPB unit, which is the core of the HERA framework, can create a true atomic switch to the patched software version without the need to use a locking software pattern like critical code sections. As explained in Section V, a trampoline can be inserted on-the-fly by the FPB hardware. The trigger to activate this insertion mechanism is a single register write [23], which can be achieved in a single store assembly instruction. A disassembly of the case study binaries confirms that the register is accessed only once with a store instruction. The CPU executes a single instruction either fully or not at all. Thus, the FPB patch activation through a register is truly atomic. The execution of an instruction cannot be interrupted halfway to cause an inconsistent state. Therefore, the RTOS can operate without special consideration of the patch activation process.

The case study showed, that the unpatched device was vulnerable to the developed ROP-attack. Without the patch, it was possible to trigger both devices to malfunction. After the patch activation, which inserted the missing bounds checks, the exploit did not work anymore. During the patch activation process, both devices continued to work normally. In what follows, we measure the exact overhead and quantify the interruptions to argue practical applicability of our approach.

B. Measuring the Overhead

As the insertion of the trampoline is performed on-the-fly based completely on hardware, the overhead caused by the addition of further code is minimal. As the patching method is based on a small block of assembly instructions, there is no impact due to used compilers or intermediate software layers. The trampoline to the dispatcher code is required as the jump target cannot be addressed directly within a jump instruction.

The inserted patch itself is handled the same way as any other code fragment. That is, it is likewise interruptible by the scheduling from the running RTOS system. The RTOS can interrupt the execution of the patch at any time, if this is required to meet real-time deadlines. The added overhead from the FPB unit is deterministic because it is only the instruction fetch or literal load for the replaced instruction [23]. This makes it possible to also patch real-time critical code sections. As the replaced instruction is well-known, the developer can exactly calculate the required time for a instruction fetch with knowledge of the microarchitecture and CPU frequency. In the following, we will further elaborate on the time required for the FPB unit to switch.

To quantify the overhead we measured the exact switching time using an oscilloscope[5] and the processors' clock as reference. We inserted triggers for external pins to determine which instruction is currently executed. The GPIO bus of the connected pins is directly connected to the CPU [52]. Furthermore, we configured the bus frequency driving the external pins to be the same as the CPU clock [52]. Hence, the pin trigger causes no additional delay other then the necessary instructions to drive the pins state. For the overhead measurements, we accept the additional delay caused by driving the pins states. However, we will eliminate this delay by a reference measurement, i.e., the times necessary for the pins to switch, see Figure 7. We repeated all measurements for five times. We did not measure any deviation. This comes as no surprise since our patch strategy completely controls the execution and builds upon hardware features that are executed atomically. Hence, we avoid giving any variance numbers in the following. Real-time hotpatching needs predictability and requires minimal overhead to meet the deadlines in place. With the help of an oscilloscope we examine two characteristics of the hotpatching system: (1) *Atomic Switch Time*: The time required to activate a single patch, that must pass through the execution cycle uninterrupted. (2) *Control Flow Redirection*: The time required to abort the execution of a single instruction and insert as well as execute the trampoline.

As we already discussed in theory, the *Atomic Switch Time* is the execution time of a single assembly instruction. The *Control Flow Redirection* just adds the cost of the transaction abort, i.e., a single instruction fetch, see Section V-A. By means of physical measurements, we check that the theoretical assumptions hold in practice. With the examination of variance of both characteristics, we can check for predictability, i.e., constant time overhead. Since the measurement itself adds overhead (e.g., pin overhead) and thus measuring a single assembly execution is not feasible, we focus on the properties. Even with the overhead in place, the measurements should have the expected properties, i.e., no variance. The overhead added through the pins is eliminated through a reference measurement. Thus, the actual overhead created by our hotpatching method can be estimated precisely.

[5]Siglent SDS1104X-E

Table I: Duration of the transaction abort for different examples. This represents the time needed to abort current instruction and switch to the *jump_section*.

Case	Duration	Pin Overhead	Difference
While-Loop	$1.624\,\mu s$	$1.384\,\mu s$	$240\,ns$
Syringe pump	$1.456\,\mu s$	$1.26\,\mu s$	$196\,ns$
Heartbeat	$1.476\,\mu s$	$1.288\,\mu s$	$188\,ns$

Atomic Switch Time. We measure the time to perform the atomic switch to enable the patch using the implemented use cases, see Section VII-A. In both cases we measure a switch time of $1.524\,\mu s$, including a pin overhead of $1.288\,\mu s$. By calculating the difference between total time and the pin overhead, we obtain the overhead time induced by the hotpatching, in this case $236\,ns$. Since the CPU was clocked at $42\,MHz$, we can calculate a total of $236\,ns \cdot 42\,MHz \approx 10$ clock cycles for the 5 switch instructions, whereby only one needs to be atomic. Thus, the atomic switch is performed in significantly less than 10 clock cycles. As mentioned before, these measurements were repeated five times, but no deviation was measured. The time required to enable the patch is small and constant, independent of the actual patch.

Control Flow Redirection. The second experiment evaluates the time to perform a trampoline insertion, i.e., the on-the-fly instruction exchange and trampoline jump. The results are presented in Table I. The expected overhead according to the data sheet is a single instruction abort [23]. As measurements on sub-instruction level are not feasible, we included the time to jump to the trampolines' target address into the measurement. We inserted a function called *jump_section* that serves as the target of the trampoline. In case of our example codes, the syringe pump and the heartbeat sensor, the measurement starts before the replaced instruction and terminates within the *jump_section*. The first table entry 'While-Loop' represents the time required to branch into the *jump_section* and directly return. Thus, the measurement is stopped a couple instructions later, after the trampoline was executed successfully. In case of the medical devices, only around eight CPU cycles are required to exchange instructions and branch into the *jump_section*: As the CPU clock is $42\,MHz$, the number of cycles that is required for the control flow redirection can be calculated. The syringe pump takes $196\,ns \cdot 42\,MHz \approx 8.2$ cycles, the heartbeat sensor takes $188\,ns \cdot 42\,MHz \approx 7.9$ cycles. The While-Loop example, which continuously performs jumps to and returns from the *jump_section*, takes $240\,ns \cdot 42\,MHz \approx 10$ CPU cycles. Because this case includes the return back to the trampoline insertion point, this is to be expected as additional instructions are executed. The minimal number of cycles to execute a single instruction is one cycle [23]. Thus, a difference of only two cycles is very small. The overall evaluation of the measured times to abort the transaction indicates that the overhead for the *Control Flow Redirection* is negligible.

In order to verify that the transaction abort, which costs a single instruction fetch, is constant and independent from the aborted instruction, we performed measurements replacing

Table II: Time required to abort different instructions.

Instruction	Cycles	Duration
NOP	1	1.644 µs
PUSH {lr}	2	1.644 µs
LDR	2	1.644 µs
B.n	2	1.644 µs
UDIV	2-12	1.644 µs

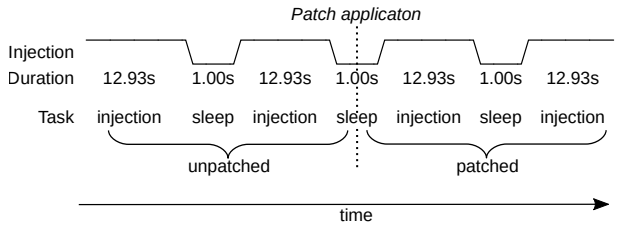


Figure 6: Measurements of the full end-to-end experiment on the syringe pump.

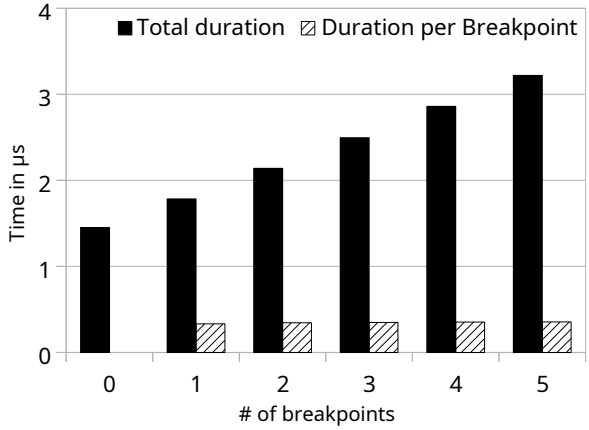


Figure 7: The duration of the patching depending on the number of breakpoints.

different instructions. The measured times, that include the pin overhead as well as the overhead to branch to and return from the *jump_section* with the respective instruction are presented in Table II. The chosen instructions are common and cover a wide range of execution costs. While the no-operation (NOP) instruction takes a single CPU cycle, the PUSH instruction takes two cycles, and a division takes up to 12 cycles to complete. The duration, i.e., the execution time, has not changed while exchanging the different instructions. Therefore, one can confirm that the transaction abort is constant and independent from the instruction, which will be replaced by the trampoline. The pin overhead is also constant and amounts to 1.384 µs for this measurement setup. The cost per instruction is broken down into details in the reference manual of the ARM Cortex-M4 processor [23].

C. Further Measurements

In general, patches can consist of multiple parts as several issues potentially have to be fixed in one hotpatch. This is easily possible with HERA as we support the use of multiple trampolines. We evaluated the time required to execute multiple patches in a row. The FPB unit on the Cortex-M4 processor supports up to six break points [23]. However, we used only five in our evaluation as we required one breakpoint for the measurement setup. The setup consists of a while-Loop containing NOP instructions. These NOP instructions have been replaced step by step with the trampolines by configuring the FPB breakpoints. Figure 7 shows the measured times for each number of breakpoints. A single measurement is a loop cycle with the previously configured amount of breakpoints. The reference baseline, i.e., the overhead due to the pin triggering, is indicated by zero breakpoints. Again, each measurement was repeated five times. Unsurprisingly, we did not measure any deviation as the switches by the FPB unit are atomic instructions with fixed execution time. It can be stated safely, that the duration per breakpoint is constant and multiple trampolines can be inserted with a known and constant overhead.

To validate that the patching process using HERA works correctly in practice, we performed a full end-to-end experiment on the syringe pump. We measured the timings of the full hotpatching process during the operation of the syringe pump using the oscilloscope. Figure 6 shows the experiment and the measured results. The syringe pump applies 1 mL with 1 s pauses in a continuous process. First, the normal operation of the unpatched program is measured. Then, during an idle phase, the patch is applied and activated. The operation continues seamlessly. Using the oscilloscope, we verified that the timings were not affected by neither the hotpatch nor the patch application. The patch application that was automatically scheduled during the idle-time and did not cause any delay.

D. Case Study: Patching FreeRTOS

In our second case study, we use the HERA framework to patch an existing vulnerability in FreeRTOS. CVE2018-16601 [56] describes a vulnerability in the TCP/IP stack of FreeRTOS, allowing to corrupt the memory and allowing a remote attacker to execute code remotely or perform a denial-of-service (DoS) attack, which disrupts the device. This vulnerability is caused by a missing bounds check because the size of the IP header is not validated at any point [57]. FreeRTOS up to version 10.0.1 is affected, the vulnerability has a CVSSv3 of 8.1 (high). This is a severe vulnerability, which should be patched immediately. However, updating a device interrupts its service, so that patches are stalled. Therefore, we developed a hotpatch which works without any impairments to the device and its service, following our previously described guidelines. As source code for FreeRTOS is available, we compiled two binaries. The first binary is a vulnerable release of FreeRTOS, the second one is a patched version. We integrated the HERA framework such that the patch is automatically applied to the first, vulnerable binary.

In detail, we developed a prototype basing on FreeRTOS Labs 160919, which uses FreeRTOS 9.0.0 together with the vulnerable TCP/IP stack [68]. First, we linked the HERA library to the vulnerable FreeRTOS and compiled it into binary. Then, we backported the fix from the recent FreeRTOS 10.0.3 with TCP/IP stack V2.2.1 to the vulnerable FreeRTOS 9.0.0. Using Radare2, we performed a binary diff from the two binaries and derived a hotpatch from it. We implemented a low-priority updater task that automatically applies the update

as soon as the system is in IDLE status. Details on the development process and implementation of this hotpatch have been already described in Section VI-C and Section VI-E. This case study shows that the HERA framework is suited to hotpatch practical security vulnerabilities.

E. Conclusion of Evaluation

With the case study on the medical devices, we showed that HERA can be used to hotpatch critical embedded systems. While the vulnerabilities in these devices have been inserted intentionally, the hotpatch for CVE-2018-16691, a serious vulnerability in FreeRTOS is an example for a existing vulnerability that can be hotpatched. Through our extensive measurements, we provide evidence that the performance overhead induced by HERA is negligible both in theory and practice. The overhead for a *Control Flow Redirection* is on a sub-instructional level, and the *Atomic Switch Time* is only a single assembly instruction, which is either executed up to completion or disregarded by the CPU. The design of the trampolines is the minimal number of instructions required to jump to the patch. It is possible to insert multiple trampolines with constant, predictable, and negligible overhead, which was verified in a full end-to-end experiment. This makes HERA suitable for patching systems with hard real-time properties, i.e., the most strict and critical requirements. In particular, our case-studies on real-world medical device software demonstrate that HERA provides an efficient and effective mechanism to hotpatch resource-constrained embedded devices that have real-time constraints.

VIII. Related Work

The need for Hotpatching comes from high availability constraints. The idea of hotpatching or dynamic software updating has been around for a while and focused on traditional software [12]. Research has spent great efforts in applying hotpatching to traditional software, especially server applications [13]–[15]. Gu, Cao, Xu, *et al.* optimized a Java VM using a lazy updating approach to avoid a disruptive halt and restart cycle [69]. Even the multi-threaded nature of server applications has been considered [15], [70]. While traditional updates, where programs or systems are restarted, erase the system state, in hotpatching this state has to be maintained. A main problem in hotpatching is to apply changes and to transfer this state. Makris and Bazzi discuss a stack reconstruction algorithm, which allows the update of active functions [71]. Other work focuses on a checkpoint model, i.e., putting program into a state where state transfer is easily achievable. The framework Kitsune [17] makes use of such update points, that need to be specified by the programmer. This model has been extended to support arbitrary complex software updates [13]. Giuffrida, Iorgulescu, Kuijsten, *et al.* present a fault-tolerant state transfer approach [72]. The framework POLUS targets server software and introduces a state synchronization model to eliminate the need for specific update points. It maintains the new and old data structures in parallel and slowly shifts towards the new data structure by means of synchronization [15]. However, not all updates require a complex state transfer mechanism. In contrast to generic program updates, security patches are said to be "small, isolated, and feature-less" [51]. The prevalent model for applying hotpatching is the relocation of executables [16], [17], [73]. In general, hotpatching, also called dynamic software updates, can be categorized by the used model and effect of the software update [34]. With the rise of Internet of Things (IoT) and the spread of cyber-physical systems, hotpatching has been considered as an approach to target security issues. Cyber-physical systems or the Internet of Things represent distributed and interact with real-world entities. Thus, real-time or high availability constraints can apply in different forms. Park, Kim, Kim, *et al.* describe an architecture on how to gather and distribute updates [74]. Felser, Kapitza, Kleinöder, *et al.* also present a special architecture for patching sensor nodes [75]. It includes the automatic calculation of differences and image creation to update the sensor node in order to incrementally link new code to the existing application. Current research focuses on the domain of smart devices and its application fields, e.g., energy management or smart cities. In sensor nodes, energy consumption is often a critical factor. This challenge is tackled by Zhang, Ahn, Zhang, *et al.* in the context of energy-harvesting devices [40]. Mugarza, Amurrio, Azketa, *et al.* describe the application and deployment of a hotpatch and live update framework Cetratus [39] in the smart city domain [76]. Salls, Shoshitaishvili, Stephens, *et al.* present Piston, a framework that allows to apply hotpatching to devices that are not designed for such hotpatching by using exploits [77].

Real-time requirements introduce further challenges to hotpatching [41]–[43]. Wahler, Richter, and Oriol describes a generic software-based hotpatching for real-time systems and its components [41]. They investigate how to identify points in time suitable for a update process and propose a state transfer model. The key assumptions are that the critical update process has a linear time amount. Thus, the critical update process fits in one cycle. Furthermore, the state transfer assumes a shared memory space that an updated component can take over. The main limitation is that the state transfer has to fit in one cycle. Therefore, Wahler, Richter, Kumar, *et al.* propose a state synchronization algorithm to synchronize two components, i.e., old and new component. An atomic switchover takes place at the full synchronization point. This allows updates with arbitrarily large states [42]. Wahler and Oriol present FASA (Future Automation System Architecture), implementing the component-based updating model [43].

An important additional aspect in high-availability systems is often fault-tolerance. Besides availability, many systems also need fault-resistance. A measure to achieve fault-tolerance is redundancy or replication. A variety of those replication schemes exist and they can be integrated into hard real-time applications [78]. Replicated systems can be patched one at a time as those systems usually substitute each other during downtime. There exist multiple solutions to achieve replication [79]. While this is a good solution if multiple, redundant instances are already available, this is not a general solution. Redundancy is costly and requires hardware and resources not present in limited embedded systems. Therefore, hotpatching embedded systems is the only solution.

IX. Conclusion and Summary

Current research has not yet addressed the challenges of applying hotpatches on embedded devices: the low resource

availability due to the need for low-energy consumption, guaranteed availability, and the simultaneous need for hard real-time capabilities. In this paper, we present HERA, the first framework to tackle all these challenges allowing the application of hotpatches to real-time constrained embedded systems using commercial off-the-shelf hardware. Patches in HERA are processed during idle-time and activated within just a single processor instruction. Thus, patching cannot interfere with the application or other processes. The patch itself is added by the on-board debugging unit during run-time with negligible and exactly predictable overhead verified by oscilloscope measurements. This makes HERA suitable for systems with the most strict real-time requirements. In a case study, we used HERA to hotpatch two medical devices with a critical vulnerability. Furthermore, we developed hotpatches for HERA to patch a real-world vulnerability in the TCP/IP stack of FreeRTOS. This shows the effective and efficient applicability of HERA to hotpatch real-time critical embedded systems.

Acknowledgment

This work has been partially funded by the DFG as part of project S2 within the CRC 1119 CROSSING. This work was supported by the DFG Priority Program SPP 2253 Nano Security (Project RAINCOAT). We thank our shepherd Jeyavijayan Rajendran and the anonymous reviewers for their valuable feedback.

References

[1] O. Alrawi, C. Lever, M. Antonakakis, and F. Monrose, "SoK: security evaluation of home-based iot deployments," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2019. DOI: 10.1109/SP.2019.00013.

[2] D. Halperin, T. S. Heydt-Benjamin, B. Ransford, S. S. Clark, B. Defend, W. Morgan, K. Fu, T. Kohno, and W. H. Maisel, "Pacemakers and implantable cardiac defibrillators: Software radio attacks and zero-power defenses," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2008. DOI: 10.1109/SP.2008.31.

[3] US Food and Drug Administration, *Firmware update to address cybersecurity vulnerabilities identified in abbott's (formerly st. jude medical's) implantable cardiac pacemakers: FDA safety communication*, Oct. 18, 2017. [Online]. Available: https://www.fda.gov/medical-devices/safety-communications/firmware-update-address-cybersecurity-vulnerabilities-identified-abbotts-formerly-st-jude-medicals (visited on 06/25/2020).

[4] D. B. Kramer and K. Fu, "Cybersecurity concerns and medical devices: Lessons from a pacemaker advisory," *Jama*, vol. 318, no. 21, 2017.

[5] D. Quarta, M. Pogliani, M. Polino, F. Maggi, A. M. Zanchettin, and S. Zanero, "An experimental security analysis of an industrial robot controller," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2017. DOI: 10.1109/SP.2017.20.

[6] E. Kovacs, "Cyberattack on german steel plant caused significant damage," *Security Week*, vol. 18, 2014.

[7] T. D. Maiziere, "Die Lage der IT-Sicherheit in Deutschland 2014," *Bundesamt für Sicherheit in der Informationstechnik*, 2014.

[8] T. Gerace and H. Cavusoglu, "The critical elements of the patch management process," *Communications of the ACM*, 2009. DOI: 10.1145/1536616.1536646.

[9] A. Nappa, R. Johnson, L. Bilge, J. Caballero, and T. Dumitras, "The attack of the clones: A study of the impact of shared code on vulnerability patching," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2015. DOI: 10.1109/SP.2015.48.

[10] M. Shahzad, M. Z. Shafiq, and A. X. Liu, "A large scale exploratory analysis of software vulnerability life cycles," in *International Conference on Software Engineering (ICSE)*, IEEE, 2012. DOI: 10.1109/ICSE.2012.6227141.

[11] G. N. Ericsson, "Cyber security and power system communication–essential parts of a smart grid infrastructure," *IEEE Transactions on Power Delivery*, 2010. DOI: 10.1109/TPWRD.2010.2046654.

[12] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for a dynamic updating," *IEEE Software*, 1993. DOI: 10.1109/52.199735.

[13] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum, "Automating live update for generic server programs," *IEEE Transactions on Software Engineering*, 2017. DOI: 10.1109/TSE.2016.2584066.

[14] M. Payer and T. R. Gross, "Hot-patching a web server: A case study of ASAP code repair," in *Annual Conference on Privacy, Security and Trust (PST)*, IEEE, 2013. DOI: 10.1109/PST.2013.6596048.

[15] H. Chen, J. Yu, R. Chen, B. Zang, and P.-C. Yew, "POLUS: A powerful live updating system," in *International Conference on Software Engineering (ICSE)*, IEEE, 2007. DOI: 10.1109/ICSE.2007.65.

[16] A. Ramaswamy, S. Bratus, S. W. Smith, and M. E. Locasto, "Katana: A hot patching framework for ELF executables," in *International Conference on Availability, Reliability and Security (ARES)*, IEEE, 2010. DOI: 10.1109/ARES.2010.112.

[17] C. M. Hayden, E. K. Smith, M. Denchev, M. Hicks, and J. S. Foster, "Kitsune: Efficient, general-purpose dynamic software updating for c," in *ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2012. DOI: 10.1145/2384616.2384635.

[18] ARM, *Arm Processors for the Widest Range of Devices–from Sensors to Servers*, 2020. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu (visited on 07/08/2020).

[19] Eclipse Foundation, *Iot developer survey 2019*, 2019. [Online]. Available: https://outreach.eclipse.foundation/download-the-eclipse-iot-developer-survey-results (visited on 07/06/2020).

[20] M. D. Schwartz, J. Mulder, J. Trent, and W. D. Atkins, "Control system devices: Architectures and supply channels overview," *Sandia Report SAND2010-5183, Sandia National Laboratories, Albuquerque, New Mexico*, 2010.

[21] J. Goodacre and A. N. Sloss, "Parallelism and the ARM instruction set architecture," *Computer*, vol. 38, no. 7, 2005. DOI: 10.1109/MC.2005.239.

[22] S. Furber, "Computing without clocks: Micropipelining the arm processor," in *Asynchronous Digital Circuit Design*, Springer, 1995.

[23] ARM, "Cortex-M4 processor technical reference manual," *Revision: r0p1, ARM 100166_0001_00_en*, 2015. [Online]. Available: https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf (visited on 12/16/2020).

[24] J. Yiu, *The Definitive Guide to ARMő Cortexő-M3 and Cortexő-M4 Processors*. Elsevier Science, 2013, ISBN: 9780124079182.

[25] Tensilica, Inc., "Xtensa instruction set architecture (isa) reference manual," *RC-2010.1 Release*, 2010.

[26] Espressif Systems, "Esp8266ex datasheet," *Version 6.6*, 2020.

[27] ——, "Esp32 series datasheet," *Version 3.4*, 2020.

[28] ——, *Espressif achieves the 100-million target for iot chip shipments*, 2018. [Online]. Available: https://www.espressif.com/en/news/Espressif_Achieves_the_Hundredmillion_Target_for_IoT_Chip_Shipments (visited on 11/02/2020).

[29] K. G. Shin and P. Ramanathan, "Real-time computing: A new discipline of computer science and engineering," in *Proceedings of IEEE, Special Issue on Real-Time Systems*, IEEE, 1994.

[30] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *Journal of the ACM (JACM)*, 1973. DOI: 10.1145/321738.321743.

[31] O. Hahm, E. Baccelli, H. Petersen, and N. Tsiftes, "Operating systems for low-end devices in the internet of things: A survey," *IEEE Internet of Things Journal*, 2016. DOI: 10.1109/JIOT.2015.2505901.

[32] P. Hambarde, R. Varma, and S. Jha, "The survey of real time operating system: Rtos," in *International Conference on Electronic Systems, Signal Processing and Computing Technologies*, IEEE, 2014. DOI: 10.1109/ICESC.2014.15.

[33] AspenCore, *2019 embedded markets study*, Mar. 2019. [Online]. Available: https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf (visited on 06/18/2020).

[34] C. Giuffrida and A. S. Tanenbaum, "A taxonomy of live updates," in *Annual Conference of the Advanced School for Computing and Imaging*, 2010.

[35] FreeRTOS, *Over the air (ota) updates*, 2020. [Online]. Available: https://www.freertos.org/ota/index.html (visited on 07/09/2020).

[36] K. Koscher, A. Czeskis, F. Roesner, S. N. Patel, T. Kohno, S. Checkoway, D. McCoy, B. Kantor, D. Anderson, H. Shacham, and S. Savage, "Experimental security analysis of a modern automobile," in *IEEE*

Symposium on Security and Privacy (SP), IEEE, 2010. DOI: 10.1109/SP.2010.34.

[37] K. Guk, G. Han, J. Lim, K. Jeong, T. Kang, E.-K. Lim, and J. Jung, "Evolution of wearable devices with real-time disease monitoring for personalized healthcare," *Nanomaterials*, vol. 9, no. 6, p. 813, 2019.

[38] M. Rushanan, A. D. Rubin, D. F. Kune, and C. M. Swanson, "Sok: Security and privacy in implantable medical devices and body area networks," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2014. DOI: 10.1109/SP.2014.40.

[39] I. Mugarza, J. Parra, and E. Jacob, "Cetratus: A framework for zero downtime secure software updates in safety-critical systems," in *International Symposium on Industrial Embedded Systems (SIES)*, IEEE, 2018. DOI: 10.1002/spe.2820.

[40] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers, "Live code update for iot devices in energy harvesting environments," in *Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, IEEE, 2016. DOI: 10.1109/NVMSA.2016.7547182.

[41] M. Wahler, S. Richter, and M. Oriol, "Dynamic software updates for real-time systems," in *Workshop on Hot Topics in Software Upgrades (HotSWUp)*, ACM, 2009. DOI: 10.1145/1656437.1656440.

[42] M. Wahler, S. Richter, S. Kumar, and M. Oriol, "Non-disruptive large-scale component updates for real-time controllers," in *Workshops of the IEEE International Conference on Data Engineering (ICDE)*, IEEE, 2011. DOI: 10.1109/ICDEW.2011.5767631.

[43] M. Wahler and M. Oriol, "Disruption-free software updates in automation systems," in *IEEE Emerging Technology and Factory Automation (ETFA)*, IEEE, 2014. DOI: 10.1109/ETFA.2014.7005075.

[44] R. Zurawski, *Embedded Systems Handbook*. CRC press, 2005. DOI: 10.1201/9781420038163.

[45] S. Holmbacka, W. Lund, S. Lafond, and J. Lilius, "Lightweight framework for runtime updating of c-based software in embedded systems," in *Workshop on Hot Topics in Software Upgrades (HotSWUp)*, USENIX Association, 2013.

[46] M. Payer, B. Bluntschli, and T. R. Gross, "Dynsec: On-the-fly code rewriting and repair," in *Workshop on Hot Topics in Software Upgrades (HotSWUp)*, USENIX Association, 2013.

[47] J. Kwon, J. Cho, and D. Park, "Function block-based robust firmware update technique for additional flash-area/energy-consumption overhead reduction," in *International Symposium on Intelligent Signal Processing and Communication Systems (ISPACS)*, IEEE, 2019. DOI: 10.1109/ISPACS48206.2019.8986373.

[48] Android Open Source Project, *A/B (Seamless) System Updates*, 2020. [Online]. Available: https://www.arm.com/products/silicon-ip-cpu (visited on 07/09/2020).

[49] Espressif Systems, *Over the air updates (ota)*, 2020. [Online]. Available: https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html (visited on 07/09/2020).

[50] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew, "Live updating operating systems using virtualization," in *International Conference on Virtual Execution Environments (VEE)*, ACM, 2006. DOI: 10.1145/1134760.1134767.

[51] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: online patches and updates for security," in *USENIX Security Symposium*, USENIX Association, 2005.

[52] STMicroelectronics, *Stm32f446xc/e technical reference manual*, 2019. [Online]. Available: https://www.st.com/resource/en/data_brief/nucleo-f446re.pdf (visited on 12/16/2020).

[53] C. Bormann, M. Ersue, and A. Keranen, "RFC 7228: Terminology for constrained-node networks," *Internet Engineering Task Force (IETF)*, vol. 7228, 2014. DOI: 10.17487/RFC7228.

[54] FreeRTOS, *GitHub - FreeRTOS*, 2020. [Online]. Available: https://github.com/FreeRTOS/FreeRTOS/tree/master (visited on 07/09/2020).

[55] Microsoft, *SDL Process Guidance Version 5.2*, 2012. [Online]. Available: https://www.microsoft.com/en-us/download/details.aspx?id=29884 (visited on 07/21/2020).

[56] *CVE-2018-16601*. Available from MITRE, CVE-ID CVE-2018-16601. 2018. [Online]. Available: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2018-16601 (visited on 07/23/2020).

[57] O. Karliner, *Freertos tcp/ip stack vulnerabilities the details*, 2018. [Online]. Available: https://blog.zimperium.com/freertos-tcpip-stack-vulnerabilities-details/ (visited on 07/23/2020).

[58] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *IEEE Symposium on Security and Privacy (SP)*, IEEE, 2013. DOI: 10.1109/SP.2013.13.

[59] D. Papp, Z. Ma, and L. Buttyán, "Embedded systems security: Threats, vulnerabilities, and attack taxonomy," in *Annual Conference on Privacy, Security and Trust (PST)*, IEEE, 2015. DOI: 10.1109/PST.2015.7232966.

[60] JSOF-Tech, *Ripple20 - 19 zero-day vulnerabilities amplified by the supply chain*, 2020. [Online]. Available: https://www.jsof-tech.com/ripple20/ (visited on 07/14/2020).

[61] B. Wijnen, E. J. Hunt, G. C. Anzalone, and J. M. Pearce, "Open-source syringe pump library," *PloS one*, vol. 9, no. 9, 2014.

[62] T. Abera, N. Asokan, L. Davi, J.-E. Ekberg, T. Nyman, A. Paverd, A.-R. Sadeghi, and G. Tsudik, "C-FLAT: control-flow attestation for embedded systems software," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2016. DOI: 10.1145/2976749.2978358.

[63] Joy-IT, *Heartbeat Sensor KY-039*, 2018. [Online]. Available: http://anleitung.joy-it.net/wp-content/uploads/2018/11/SEN-KY039-Manual.pdf (visited on 07/23/2020).

[64] D. Kushner, "The making of arduino," *IEEE Spectrum*, vol. 26, 2011.

[65] A. Francillon and C. Castelluccia, "Code injection attacks on harvard-architecture devices," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, ACM, 2008. DOI: 10.1145/1455770.1455775.

[66] O. Aleph, "Smashing the stack for fun and profit," *Phrack Magazine*, 1996. [Online]. Available: http://www.shmoo.com/phrack/Phrack49/p49-14.

[67] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, 2007. DOI: 10.1145/1315245.1315313.

[68] FreeRTOS, *Labs 160919*, 2016. [Online]. Available: https://www.freertos.org/FreeRTOS-Labs/downloads/160919_FreeRTOS_Labs.zip (visited on 07/23/2020).

[69] T. Gu, C. Cao, C. Xu, X. Ma, L. Zhang, and J. Lu, "Javelus: A low disruptive approach to dynamic software updates," in *Asia-Pacific Software Engineering Conference (APSEC)*, IEEE, 2012. DOI: 10.1109/APSEC.2012.55.

[70] F. Rommel, L. Glauer, C. Dietrich, and D. Lohmann, "Wait-free code patching of multi-threaded processes," in *Workshop on Programming Languages and Operating Systems, (SOSP)*, ACM, 2019. DOI: 10.1145/3365137.3365404.

[71] K. Makris and R. A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," in *USENIX Annual Technical Conference*, USENIX Association, 2009.

[72] C. Giuffrida, C. Iorgulescu, A. Kuijsten, and A. S. Tanenbaum, "Back to the future: Fault-tolerant live update with time-traveling state transfer," in *Large Installation System Administration Conference (LISA)*, USENIX Association, 2013.

[73] H. Jeong, J. Baik, and K. Kang, "Functional level hot-patching platform for executable and linkable format binaries," in *IEEE International Conference on Systems, Man, and Cybernetics (SMC)*, IEEE, 2017. DOI: 10.1109/SMC.2017.8122653.

[74] M. J. Park, D. K. Kim, W.-T. Kim, and S.-M. Park, "Dynamic software updates in cyber-physical systems," in *International Conference on Information and Communication Technology Convergence (ICTC)*, IEEE, 2010. DOI: 10.1109/ICTC.2010.5674807.

[75] M. Felser, R. Kapitza, J. Kleinöder, and W. Schröder-Preikschat, "Dynamic software update of resource-constrained distributed embedded systems," in *Embedded System Design: Topics, Techniques and Trends, IFIP TC10*, Springer, 2007. DOI: 10.1007/978-0-387-72258-0_33.

[76] I. Mugarza, A. Amurrio, E. Azketa, and E. Jacob, "Dynamic software updates to enhance security and privacy in high availability energy management applications in smart cities," *IEEE Access*, 2019. DOI: 10.1109/ACCESS.2019.2905925.

[77] C. Salls, Y. Shoshitaishvili, N. Stephens, C. Kruegel, and G. Vigna, "Piston: Uncooperative remote runtime patching," in *Annual Computer Security Applications Conference (ACSAC)*, ACM, 2017. DOI: 10.1145/3134600.3134611.

[78] P. Chevochot and I. Puaut, "Scheduling fault-tolerant distributed hard real-time tasks independently of the replication strategies," in *International Conference on Real-Time Computing Systems and Applications (RTCSA)*, IEEE, 1999. DOI: 10.1109/RTCSA.1999.811280.

[79] R. Guerraoui and A. Schiper, "Software-based replication for fault tolerance," *Computer*, 1997. DOI: 10.1109/2.585156.