RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices

Yi He and Zhenhua Zou, *Tsinghua University and BNRist;* Kun Sun, *George Mason University;* Zhuotao Liu and Ke Xu, *Tsinghua University and BNRist;* Qian Wang, *Wuhan University;* Chao Shen, *Xi'an Jiaotong University;* Zhi Wang, *Florida State University;* Qi Li, *Tsinghua University and BNRist*

https://www.usenix.org/conference/usenixsecurity22/presentation/he-yi

This paper is included in the Proceedings of the 31st USENIX Security Symposium.

August 10–12, 2022 • Boston, MA, USA

978-1-939133-31-1

Open access to the Proceedings of the 31st USENIX Security Symposium is sponsored by USENIX.

RapidPatch: Firmware Hotpatching for Real-Time Embedded Devices

Yi He, Zhenhua Zou
Tsinghua University and BNRist

Kun Sun
George Mason University

Zhuotao Liu, Ke Xu
Tsinghua University and BNRist

Qian Wang
Wuhan University

Chao Shen
Xi'an Jiaotong University

Zhi Wang
Florida State University

Qi Li
Tsinghua University and BNRist

Abstract

Nowadays real-time embedded devices are becoming one main target of cyber attacks. A huge number of embedded devices equipped with outdated firmware are subject to various vulnerabilities, but they cannot be timely patched due to two main reasons. First, it is difficult for vendors who have various types of fragmented devices to generate patches for each type of device. Second, it is challenging to deploy patches on many embedded devices without restarting or halting real-time tasks, hindering the patch installation on devices (e.g., industrial control devices) that have high availability requirements. In this paper, we present RapidPatch, a new hotpatching framework to facilitate patch propagation by installing generic patches without disrupting other tasks running on heterogeneous embedded devices. RapidPatch allows RTOS developers to directly release common patches for all downstream devices so that device maintainers can easily generate device-specific patches for different firmware. We utilize eBPF virtual machines to execute patches on resource-constrained embedded devices and develop three hotpatching strategies to support hotpatching for all major microcontroller (MCU) architectures. In particular, we propose two types of eBPF patches for different types of vulnerabilities and develop an eBPF patch verifier to ensure patch safety. We evaluate RapidPatch with major CVEs on four major RTOSes running on different embedded devices. We find that over 90% vulnerabilities can be hotpatched via RapidPatch. Our system can work on devices with 64 KB or more memory and 64 MHz MCU frequency. The average patch delay is less than 8 μs and the overall latency overhead is less than 0.6%.

1 Introduction

Real-time operating systems (RTOSes) used in microcontroller-based embedded devices do not put enough emphasis on security, considering that those devices are typically isolated and disconnected from the world and thus can hardly be accessed remotely. However, with the increasing market growth of Internet of Things (IoT), the connected embedded devices running RTOS (e.g., FreeRTOS [5], Zephyr [26]) have been widely used in many areas, such as medical devices, industry devices, critical infrastructure systems, and Smart Homes. Meanwhile, cyber-attacks have emerged and proliferated on these embedded devices. Quite a few recently discovered vulnerabilities, such as those reported in RIPPLE20 [21] and AMNESIA33 [6], have affected hundreds of millions of devices [10]. Thus, it is critical to timely fix the vulnerabilities on embedded devices to minimize the exploiting window.

Current IoT vendors need to support heterogeneous devices with different specifications and architectures that are manufactured on various product lines. Yet, it is challenging to safely merge the upstream security patches to fix the vulnerabilities on these devices, even though they may share the same RTOS/libraries. First, it is time consuming to apply source code patches since the devices maintainers need to recompile and test the post-merging firmware. Second, existing firmware deployment solutions often introduce a long service downtime, which is unacceptable to the embedded devices used in time-critical industry (e.g., power grid) or medical cares (e.g., cardiovascular implantable devices [31]). Most existing firmware updating works in IoT [19, 39, 51] use traditional OTA method to update entire or part of the firmware, inevitably disrupting the services running on the embedded devices for an unpredictable period of time.

Recently, hotpatching methods have been successfully developed on Linux and Android devices [33, 34, 57] to minimize the service downtime and reduce the system reboot times. However, none of these are comfortably applicable on embedded devices due to the hardware discrepancy. For instance, KARMA [34] and VULMET [57] require to modify the code in memory to add trampolines. However, since embedded devices execute code on flash-based ROM, the entire flash sector needs to be erased before updating, triggering high patching latency. Also, the trampolines-based hotpatching approaches may require halting or rebooting the system.

HERA [35] is the first hotpatching system designed for

real-time embedded devices. It can successfully hotpatch the firmware without modifying the ROM. It relies on the ARM Cortex-M3/M4 processors' unique flash patching hardware feature [4, 59] to trigger the trampolined patch code loaded in the RAM. Unfortunately, this hardware feature is no longer supported in new generation ARM successors such as Cortex-M7 [24]. To support a wide range of microcontroller units (MCUs) with various architectures such as Xtensa, AVR, and RISC-V, we need a general hotpatch triggering method. Also, HERA requires the device maintainers to merge the source code patch of vulnerabilities and generate binary patch via binary diff tool, which has been proved to be the main adoption obstacle of hotpatching technique in Android [33, 34]. Moreover, when the device maintainers have multiple types of device products, HERA's patch generating approach incurs repetitive efforts and cannot timely patch various devices. In general, current RTOSes lack a generic hotpatching mechanism (like the KSplice [30] in the Linux ecosystem) that can use a single patch to fix the same vulnerability in various embedded devices with different specifications and architectures.

In this paper, we present RapidPatch, the first hotpatching framework for the device maintainers to quickly develop patches for all their heterogeneous embedded devices and deploy these patches instantly without disrupting device operations. The core innovation of RapidPatch is to provide a common patch runtime executable on heterogeneous devices with different MCUs by leveraging the eBPF [2] virtual machine such that a single eBPF patch can fix the same vulnerability for *all* devices with the same RTOS/library version. Towards this end, RapidPatch is powered with multiple tightly coupled components: (i) a patch generation module that employs a *toolchain* to generate device-specific patches given the eBPF patches and configurations (obtained from the original C source code patches); (ii) a patch verification module that uses static analysis to check whether the patches include malicious behaviors (e.g., manipulating kernel data); and (iii) multiple patch deployment strategies tailored for various MCUs to install these patches by redirecting execution control flow to the patch dispatcher *without* modifying the ROMs/firmware.

Contributions. The major contribution of this paper is the design and implementation of RapidPatch, the first hotpatching system supporting heterogeneous real-time embedded devices. Compared with the existing firmware updating approaches [19, 35, 39, 51], RapidPatch does not require merging source code patches to generate binary patches. Instead, it employs an eBPF [2] virtual machine based common patch runtime to execute bytecode patches so that the patches can run externally without changing the firmware on various architectures. Thus, a common patch can fix the same vulnerability on different devices. Further, RapidPatch provides new hotpatching strategies for heterogeneous devices and thus supports MCUs that cannot use HERA [35], which mostly leverages a deprecated processor feature. Finally, to minimize the cost of manual patch validation, we develop a patch verifier and extend the eBPF patch runtime to support safe patches [50] that can be deployed without extra tests on individual devices. We prototype RapidPatch [1] with 6468 lines of C and 1733 lines of Python and use reported vulnerabilities of popular RTOS/Library to evaluate the prototype on different devices. The experiment results demonstrate the applicability, generality, and performance of RapidPatch. Particularly, the average patching delay introduced by RapidPatch (JIT mode) is less than 8 μs, which has negligible impacts on real-time tasks, and the average incurred overhead is less than 0.6%.

2 Background

2.1 Embedded Devices

Typical embedded devices [20] are monolithic systems based on low-power MCUs that are provided by various IC manufacturers and use different CPU architectures [11] such as ARM Cortex-M series, MIPS, Atmel AVR, and Xtensa. Embedded devices may contain a single MCU that integrates all the functions, e.g., nRF52 series [18]. Alternatively, they may use multiple MCUs (e.g., STM32 MCU and ESP32) interacting with each other via GPIO interfaces.

Compared with hardware, vendors typically have fewer choices on system software and have to use either mature open source embedded OSes or commercial close-sourced embedded OS, e.g., Samsung's TizenRT that is customized from NuttX [7]. Amazon's FreeRTOS and Linux Foundation's Zephyr are two popular embedded OSes that have supported hundreds of MCUs [11] since 2020. As a result, firmware with the same code base has been used in a large number of embedded devices. Since embedded devices are often deployed in sectors such as industry control systems and medical treatment, it is critical for these real-time devices to respond and finish tasks in strict time [37]. To facilitate the applications, RTOSes contain a number of system libraries to support extensive modules such as networking and TLS.

2.2 Embedded Device Hotpatching

Existing hotpatching solutions on Linux/Android cannot work on embedded devices due to hardware discrepancy (see Section 3.2). Fortunately, many MCUs provide build-in hardware features that can be used for hotpatching.

Flash Patching. ARM Cortex-m3/m4 processors provide the Flash Patch and Breakpoint Unit (FPB) [4] for patching the code in ROM at runtime. When the FPB mode is enabled, the hardware breakpoints can be used to mark instructions and redirect the instructions at the hardware breakpoints to the mapped instructions, which are defined in a remap table. Then, a branch instruction such as *bl trampoline_func* in the remap table can intercept the buggy function and jump to run the new function [35].

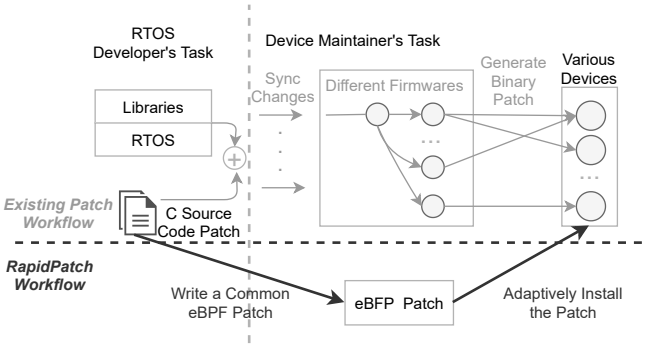


Figure 1: Existing patch workflow vs. RapidPatch workflow.

KProbe. The ARM and RISC-V MCUs support customer handlers defined at arbitrary addresses for debugging events, which can be used to implement the KProbe mechanism [12] in Linux. On ARM Cortex-M3 and beyond, we can define customer exception handlers for debugging events in the Debug Monitor mode [58] and then port the KProbe features to different embedded devices. Afterwards, KProbe can be leveraged to implement hotpatching solutions by redirecting the control flow of the buggy code to a trampoline function.

2.3 Extended Berkeley Packet Filter (eBPF)

The extended Berkeley Packet Filter (eBPF) is used as a Linux in-kernel virtual machine that can execute untrusted eBPF code received from user space. The original Linux eBPF supports various types of eBPF programs that are attached to predefined places to provide capabilities such as TCP/IP packet filter, tracing, and sandbox policy configuration. The eBPF code is powerful since it can be written in C and then compiled into eBPF bytecode with the bpf LLVM backend. To prevent malicious code, before running the eBPF code, the kernel uses a static verifier to vet the program and ensure that the program does not have memory accessing errors or unbounded loops. To enable cross-platform patches, we choose to implement a bytecode-based patching approach. Since the eBPF VM is more lightweight than other embedded VM-based languages including Lua, Javascript, and Python, we choose to port it to the resource constrained devices. Thus, we can benefit from eBPF's powerful features, such as supporting C grammar and memory manipulation via pointers, having a simple yet highly efficient instruction set, and being easily verifiable in terms of execution security. Furthermore, we customize the eBPF verifier and runtime to support safe patches.

3 Problem Statement

3.1 Delayed RTOS Update

For embedded device vendors who customize their firmware based on official RTOS versions (e.g., Samsung TizenRT and Xiaomi Vela are based on Apache NuttX RTOS [7]), they need to integrate a number of third-party libraries (e.g., MbedTLS [15], the de facto standard of embedded TLS library) to extend device functionalities. We illustrate the patch propagation process on embedded systems in Figure 1 to explain why upstream security patches may not be propagated to the end users in time. First, it takes time for the downstream vendors (e.g., Samsung) to update their firmware by merging the new commits from the upstream repositories (e.g., Apache NuttX) and verify the correctness of the new firmware. Second, the end users usually lack the motivation and expertise to upgrade their own firmware [56]. Therefore, similar to the Android systems [62], this open model often introduces a significant patching delay on embedded devices. Quantitatively, we report the security patch delay using multiple CVEs on the MBedTLS Library [15]. Figure 2 shows that it usually takes two active RTOSes, namely, Zephyr and FreeRTOS, 3 to 6 months to update their systems, and the less active TizenRT even takes more than one year on average to patch vulnerabilities.

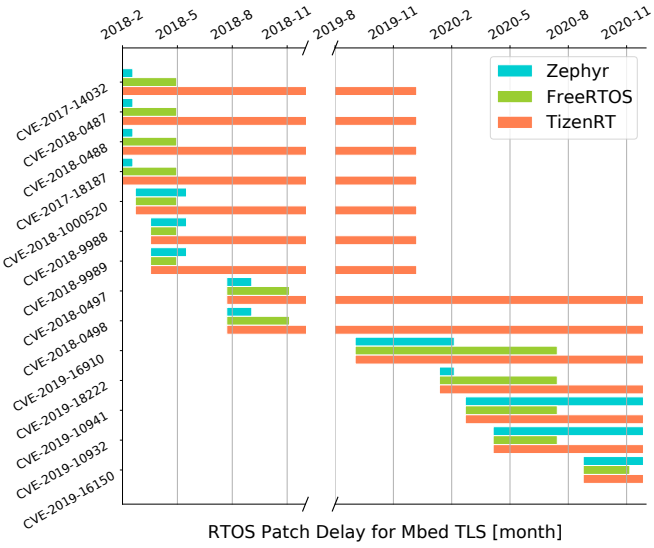


Figure 2: The MbedTLS's patch delay in different RTOSes.

3.2 Patching Challenges

We identify two major challenges to patch embedded devices in time. The first one is to generate patches correctly and timely for various embedded devices. For specific hardware devices, the vendors need to first merge the changes of source code and then build new firmware using each device's compilation configuration. After that, it is critical to verify the effectiveness of the patch, as well as the normal functionality of the post-patching device. Thus, patch generation is a meticulous process that is time consuming and requires specialized expertise (e.g., familiar with multiple firmware codebase).

The second challenge is to deploy patches without disrupting device operations. Existing hotpatching solutions on Android and Windows/Linux systems (e.g., relocating linked libraries [47], runtime function instrumentation [53, 60], and seamless update with A/B scheme [3, 43]) cannot be directly applied in RTOS for the following reasons. To support relocating linked libraries in RTOS, Simon et al. [47] separate user tasks of FreeRTOS to multiple Executable and Linkable Format (ELF) file. However, it can only update user level tasks but not the RTOS kernel. The runtime function instrumentation approach needs to modify instructions on-the-fly; however, since embedded devices use ROM or Nor-Flash memory to store the firmware code, it is time consuming to rewrite the flash memory and thus is impossible to update the firmware without halting the system. The A/B scheme, widely adopted by Linux [43], Android, and Espressif ESP32 IoT systems [9], maintain two instances and migrates from the old system A to the new system B. Thus, it naturally requires two storage slots, whereas most embedded devices do not have enough memory storage for two systems. Also, it still needs to reboot the system after updating the kernel. HERA [35] uses the Flash Patch and Breakpoint Unit (FPB) [59] on Cortex-M3/M4 MCUs to dynamically patch embedded devices; however, it cannot work on other types of MCUs that do not have the Flash Patch feature. Moreover, it requires fixing bugs in the source code to compile a new firmware, and the device maintainers need to manually generate and verify binary patches for different types of embedded devices before deploying these patches, which is labor-intensive.

3.3 A Motivating Example

We propose a generic patching framework to address above challenges. The core idea is to provide a common hotpatching runtime for heterogeneous embedded devices to execute patch code written in eBPF bytecode (referred to as *eBPF patch* hereafter) running in eBPF virtual machine. Since the eBPF patches can be written and deployed separately, they do not require any modification in the firmware.

We use the CVE-2020-10063 from Zephyr as an example to showcase our high-level design. The vulnerable functions and the corresponding eBPF patch code are shown in Figure 3. An adversary can send a CoAP packet with illegal options to trigger the integer overflow vulnerability (the line in red background) in function `parse_option`, leading to endless loop. The original patch adds extra checks to prevent the overflow (the green lines). In our eBPF patch solution, before executing the vulnerable function, it redirects the control flow to the eBPF VM, passes the function's arguments to the eBPF VM via stack, and then runs the eBPF code to check these arguments. When detecting malicious inputs, the eBPF VM directly returns the `EINVAL` error code to the function caller. Otherwise, it restores the origin control flow.

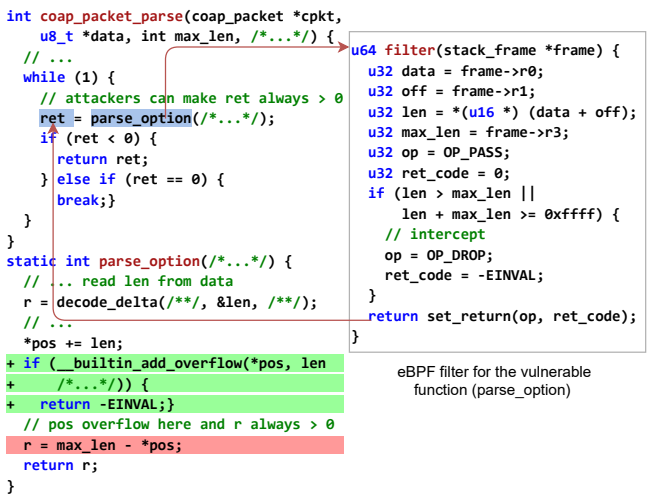


Figure 3: The eBPF filter patch for CVE-2020-10063.

3.4 Threat Model and Assumptions

We assume that the attackers can only attack the embedded devices remotely and cannot access the devices physically. Since the firmware of embedded devices is typically customized by the device vendors and does not support third-party user apps, attackers cannot inject malicious tasks (e.g., via user apps) to execute third-party code. We assume that the device maintainers can obtain patches from trusted sources. Similar to HERA [35], we assume the patches are signed and encrypted during transfer so that the maintainers can safely verify their integrity and authenticity. Since only authorized users (such as the device maintainers) are able to apply RapidPatch to perform hotpatching, attackers can only exploit the patch system via installing malicious patches after compromising the authorized users' credentials.

Since the patches from the RTOS developers may be buggy due to design flaws and implementation bugs, we propose to use filter patches that aim to prevent dangerous instructions from harming the systems, e.g., system crashes or memory corruption. However, similar to the existing approaches [33, 34, 50], RapidPatch itself cannot ensure the patches should work properly, which, in general, is decided by the patch developers.

4 System Design

RapidPatch contains a toolchain for the patch developing phase to generate and verify the patches and a hotpatch runtime for the patch execution phase that can be embedded into existing RTOS or bare-metal firmware to run patches on heterogeneous embedded devices. We introduce the design of these modules by following the workflow of RapidPatch.

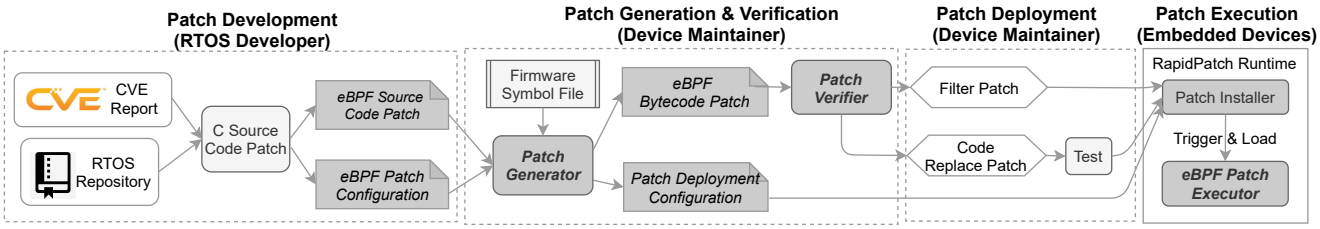


Figure 4: Overview of RapidPatch.

4.1 Overview of RapidPatch

The overall workflow of RapidPatch is shown in Figure 4. When a vulnerability is discovered, the RTOS developers first develop an original patch written in C to fix the vulnerability in the source code. Next, based on the C source code patch, the RTOS developers write an eBPF source code patch and a configuration file describing how to apply the eBPF patch.

Upon being notified with newly available eBPF source code patches, the device maintainers first check if the vulnerable RTOS is used by their devices and download the corresponding patches if necessary. Since the device maintainers may have different models of embedded devices, our toolchain provides a tool named *patch generator* to help them automatically generate corresponding patches for different devices. The patch generator compiles the eBPF source code into eBPF bytecode and analyzes the firmware symbol files to decide the correct patch installation address and obtain the global addresses (e.g., global variables, functions, or basic block addresses) that are required by the eBPF code. The patch generator has two outputs: the eBPF bytecode patch and the patch deployment configuration file for installing the patch on a specific firmware.

Based on the bug fixing strategy, we support two types of eBPF patch: the *filter patch* that resolves the vulnerability by filtering out malicious inputs and the *code replace patch* that replaces the vulnerable code with secure patched code. Given the eBPF bytecode patches, the device maintainers can use a tool named *patch verifier* to verify the safety of the patch by identifying harmful behaviors such as illegal access of kernel memory or excessive loop iterations. Since the filter patch only performs certain security checks without introducing new execution paths, it is considered as a form of safe patch [50]. The code replace patch, however, may require further test to verify its effectiveness and security.

Finally, the device maintainers install the patches to the target device via network or other external interfaces such as USB. The patch data transfer is executed in an independent thread and the patch is installed to the patch dispatcher using read-copy-update (RCU) operations. When the vulnerable code is executed, the patch will be triggered and executed by RapidPatch's hotpatching runtime. All the installation steps do not disrupt the interrupt requests (IRQ) and can be executed with negligible overhead.

In the following, we describe the design details of RapidPatch, including patch development, patch generation & verification, patch deployment, and patch execution.

4.2 Patch Development

Based on the original C source code patch obtained from a trusted source, it is straightforward for RTOS developers to write the eBPF source code patches. The eBPF filter patch can fix a large portion of vulnerabilities by validating the inputs. For the vulnerabilities that cannot be fixed by checking the inputs, code replace patch can be developed to replace the vulnerable code.

Filter Patch. It works by filtering out malicious or illegal inputs at the entrance of the vulnerable code segment. We use CVE-2020-17443 in AMNESIA33 [6] as an example to illustrate its development. The original patch in C source code is shown in Figure 5(a), where the green lines are the newly added boundary checking code. The root cause of the vulnerability is that PicoTCP library [23] does not check the payload length when processing ICMPv6 echo requests, so function `memcpy` may suffer from underflow attacks when it is provided an invalid packet with a small packet length (i.e., less than 8).

To fix this vulnerability, a filter patch can be triggered at the entrance of function `pico_icmp6_send_echoreply` to filter out illegal packet length values, as shown in Figure 5(b). The function arguments are pushed to the stack by the RapidPatch runtime, so the eBPF code can obtain the packet length argument via pointer operations and then validate the length. The eBPF VM returns different op codes based on the validation results, including `OP_PASS` (i.e., restore and continue to execute the function), `OP_REDIRECT` (i.e., return to a basic block of the vulnerable function), and `OP_DROP` (i.e., skip the vulnerable function). Finally, the RapidPatch runtime handles the results and redirects the control flow by modifying the return address (see Section 4.4).

Code Replace Patch. For the logical bugs that can only be fixed via other operations, e.g., calling new functions, changing variable values, changing data types, or redesigning the entire functions [57], we fix them by replacing the vulnerable functions with the eBPF patch code. A foreign function interface (FFI) is enabled to call and execute native C functions

```
1 int pico_icmp6_send_echoreply(struct pico_frame *echo) {
2  // ... omit
3 + if (echo->transport_len < PICO_ICMP6HDR_ECHO_REQUEST_SIZE) {
4 +     return -1; // invalid packet
5 + }
6  /*bug: When the echo->transport_len is less than the
7    PICO_ICMP6HDR_ECHO_REQUEST_SIZE, the memcpy len will
8    arithmetic underflow here */
9  memcpy(reply->payload, echo->payload, (uint32_t)
     (echo->transport_len - PICO_ICMP6HDR_ECHO_REQUEST_SIZE));
10  // ... omit
11 }
```

(a). The C Souce Code Patch

```
#include "ebpf_helper.h"
const int PICO_ICMP6HDR_ECHO_REQUEST_SIZE = 8;

uint64_t filter(stack_frame *frame) {
    uint8_t *echo = (uint8_t *)(frame->r0);
    uint16_t *transport_len_ptr = (uint16_t *)(echo + 38);
    uint16_t transport_len = (uint16_t)(*transport_len_ptr);

    if (transport_len >= PICO_ICMP6HDR_ECHO_REQUEST_SIZE) {
        return set_return(OP_PASS, 0);
    }
    return set_return(OP_DROP, -1);
}
```

(b). The eBPF Filter Patch

Figure 5: Filter patch for CVE-2020-17443

in the eBPF code via eBPF helper functions. The memory and current stack frame can also be accessed inside the eBPF VM. The RapidPatch runtime loads and executes the code replace patch at the location of the vulnerable code and redirects the control flow to the exit point (e.g., the function's return address) of the vulnerable code.

As an example, Figure 6(a) shows the C source code patch for CVE-2020-10023, where the buffer overflow vulnerability can be raised by wrongly passing `shift` (instead of `j`) in the third argument of function `memmove` in the shell subsystem.

To fix this bug, Figure 6(b) shows the code replace patch that revises the original function `shell_spaces_trim`. The calling of C function `memmove` is handled by the `C_CALL` eBPF FFI API. Similar to the filter patch, code replace patch uses `OP_DROP` or `OP_REDIRECT` as the return code to indicate if the patch loader should directly return to caller of the vulnerable function or return to the error handle procedure. We can further extend the sanity test on the vulnerable function by acquiring more status of function calls. For instance, when patching the Key Negotiation of Bluetooth (KNOB) vulnerability (CVE-2019-9506) [28] by banning the Bluetooth connections with insufficient encryption key lengths, we can obtain the key length via the Bluetooth's native Host-Controller Interface (HCI) API. Since both memory writing operations (out of the eBPF stack) and the native C function calls are allowed in code replace patch, it may introduce new bugs that should be well identified by the patch verifier (see Section 4.3).

Patch Configuration. Both filter patch code and code replace patch code are generated together with a configuration file that specifies critical information about patch compilation and patch deployment. We use YAML to write patch configuration files. An example configuration is shown in Figure 14 in Appendix. The first part provides the interception point of

```
1 void shell_spaces_trim(char *str){
2  // ...
3  for (u16_t j = i + 1; j < len; j++) {
     // ...
- 4   memmove(&str[i + 1],
- 5         &str[j], len - shift + 1);
+ 6   memmove(&str[i + 1],
+ 7         &str[j], len - j + 1);
    // ...
  }
  // ...
}
```

(a). The C Source Code Patch

```
void ebpf_spaces_trim(stack_frame
*frame ) {
  // ...
  for (u16 j = i + 1; j < len;
    j++) {
    // ...
    C_CALL(FUNC_memmove,
          &str[i + 1], &str[j],
          len - j + 1);
    // ...
  }
// ...
}
```

(b). The eBPF Code Replace Patch

Figure 6: Code replace patch for CVE-2020-10023

vulnerable RTOS, either the entrance of a vulnerable function or the entrance of a basic block, to trigger the patch code. The second and third parts define the trigger type and the run type of patch code in RapidPatch runtime, respectively. The last part gives a map of the native C functions and required global variables.

4.3 Patch Generation & Verification

Now we describe the patch compilation and patch verification process, which produces device-specific eBPF bytecode patches deployable on real devices.

Patch Generator. It takes three inputs to generate patches: the eBPF patch code, the path configuration file, and the symbol table of the target vulnerable firmware. At the beginning of compiling process, all macros in the eBPF patch code referring to function names and global variables are replaced with concrete firmware addresses. This is done by traversing symbols declared in `variable_map` of patch configuration file and searching the symbol table of the target firmware. Then, we use the Clang bpf tool to generate device-specific eBPF bytecode patches for the target firmware. Finally, we obtain the patch installation address by parsing the `install_point` tag defined in patch configuration file and searching the symbol table to find out the corresponding symbol's address. These steps allow us to generate device-specific eBPF patches.

Patch Verifier. The compiled eBPF bytecode patches are checked by the patch verifier to verify whether the patch is a filter patch that can be safely deployed. Certain patch code behaviors, such as unbounded loops, modifying the memory out of eBPF stack, and calling C functions, are considered as unsafe. patch verifier can automatically identify these behaviors by analyzing the eBPF instructions of eBPF bytecode patch. It can also report the code replace patches that may change the code logic and therefore require additional manual testing while the filter patches can waive such tests.

Note that Linux's original eBPF verifier cannot be directly applied here, since it performs very strict checks on memory accessing scope and loops. Even the filter patch may have to break these rules. For instance, reading arbitrary memory space to access global variables or local variables in vulnera-

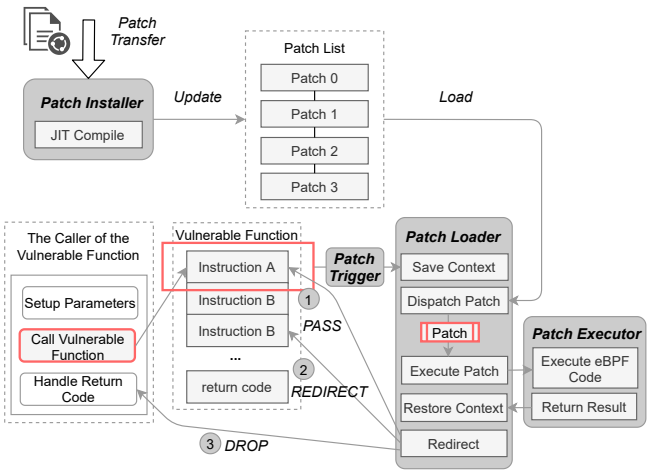


Figure 7: RapidPatch runtime system.

ble function stack is necessary in eBPF patch code in order to make proper decisions (such as dropping an incoming request). Further, applying unbounded loops has been a useful technique in developing eBPF patches. For example, when writing an eBPF patch for CVE-2020-10062 of Zephyr, we use an unbounded loop to obtain the length of input MQTT packet. Different from the Linux eBPF verifier that performs verification in runtime, we choose to verify the eBPF patch code offline, considering that the computation power of embedded devices are limited. Additionally, the results of patch verifier checks can alert the patch developer. If the patch is identified as safe, it can be deployed on the device directly without further tests (which is guarded by our software-based fault isolation (SFI) approaches, see Section 4.4). However, when the patch is reported as unsafe, the device maintainers have to perform further manual tests.

4.4 Patch Deployment and Execution

The device maintainers can deploy the patches on the embedded devices using the same encrypted transfer tunnels as the traditional OTA updates. Then the patches are installed on the vulnerable devices and executed by the RapidPatch runtime as shown in Figure 7.

Patch Installer. It first performs JIT compilation and then saves the eBPF patch bytecode, JIT native code, and the patch deployment configuration to a patch list. The installer adopts a lock-free updating approach, which performs an atomic switch to a new patch list that contains new patches, similar to Linux RCU (Read-Copy Update) that ensures patch list is continuously accessible by other threads such as patch loader.

Patch Trigger. We propose three methods to trigger patches: (i) the fixed patch trigger inserted manually by programmers or automatically by the compile-time instrumentation, (ii) the MCU's build-in patch features (e.g., FPB Flash Patch), and (iii) KProbe implemented via hardware debug primitives. As shown in Table 1, these three patch triggering methods have different usage scopes. The KProbe and FPB can be inserted or removed at arbitrary addresses during system runtime, but the fixed patch triggers are added when programming or compiling firmware and can only be placed at fixed positions of a function, e.g., the function entrance. However, the FPB and KProbe are triggered by the hardware breakpoints and therefore the number of active patches is limited by the hardware breakpoint number (e.g., 6 to 8 in ARM), while the number of fixed patch trigger is limited by storage space of devices.

	Patch Point	Support Device	#Patch
Fixed Patch Points	Function Begin	All	32+
FPB	Basic Block	Only Cortex-M3/M4	6
KProbe	Basic Block	Cortex-M3∼M55(all), RISC-V	8

Table 1: Comparison of different patch triggering methods.

Similar to the eBPF XDP [8] hooks, fixed patch triggers are placed in critical paths of ingress packets and can be applied to validate inputs. We observe that most of the vulnerabilities in FreeRTOS and Ripple 20 [21] are located in the payload processing functions of different layers in the network stack. To mitigate such vulnerabilities in the network stack, we can intercept original flawed packet processing flow by adding fixed patch triggers at the entrance of packet ingress functions and run eBPF patch code to validate the payload. Note that KProbe can only work on devices with hardware support such as ARM, RISC-V; FPB is only supported by the Cortex-M3/M4 [24]; and the fixed patch points are supported by all devices. We further discuss it in Section 6.2.

Patch Loader. When a patch is triggered, the patch loader first saves the context of current function and dispatches the patch based on the patch index (e.g., lr value) to locate the appropriate patch from the patch list. For instance, the context switch on ARMv7-M is shown in Figure 8 where the registers are saved in stack and passed to the eBPF VM. Thus the eBPF patch code can access the function arguments from the stack during execution. Finally, after the patch finishes and exits, patch loader handles the results and redirects the control flow based on the return code of the eBPF VM by modifying the *lr* (return address) register.

Patch Execution and Runtime Protection. The eBPF patch code is executed in a new eBPF VM instance using either interpreter mode or JIT mode. Similar to KARMA [34], we support sharing state among different eBPF VMs to enable stateful patches that can share values among different patches via an eBPF map. We use software-based fault isolation (SFI) to ensure the security of patch execution. We implement the SFI in both the eBPF interpreter and the eBPF JIT compiler by adding extra checks to risk-sensitive instructions, including the storage instructions and the jump instructions. We also bound the number of the loop iterations in filter patch by

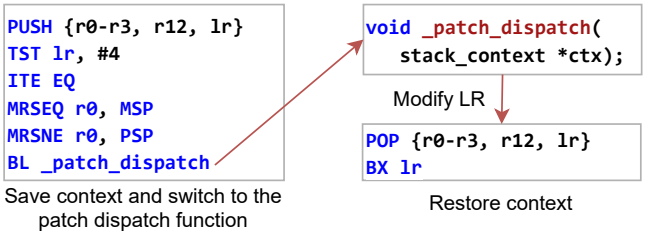


Figure 8: Context switch for patch dispatching

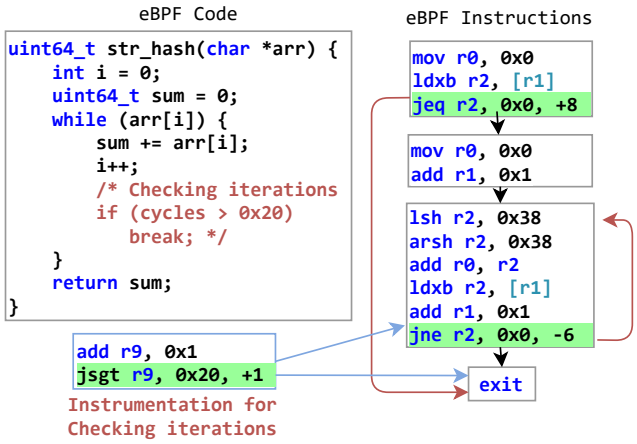


Figure 9: An example of limiting the loop iterations.

adding extra checks to the jump instructions. The jumps with negative offset are backward jumps and indicate that there are loops. By using a single variable to count all the backward jumps including both inner loops and outer loops, we can obtain the iterations of all loops. As shown in Figure 9, in JIT mode, we add extra instructions via instrumenting JIT compilation to record the iteration cycles and check if it is larger than the threshold (i.e., 0x20, 32) before jumping to negative offsets. In this way, we can limit total number of loop iterations and bound the total execution time of filter patches.

Note that, all the steps, from patch installation to patch triggering and execution, do not disrupt the interrupt requests (IRQ) and can run without halting other tasks. Thus, RapidPatch realizes hotpatching with limited latency, satisfying the real-time requirements on embedded devices.

5 System Implementation

We prototype the RapidPatch toolchain to generate and verify patches, and develop RapidPatch runtime as a module in popular RTOSes including Zephyr [26], Amazon's FreeRTOS [5], LiteOS [14], and NuttX [7] to install and execute patches. To demonstrate the generality of RapidPatch, we port RapidPatch runtime to devices with different specifications and architectures, as listed in Table 4. To reproduce CVEs in real devices, we also port specific modules associated with the vulnerable code, such as USB and Ethernet modules. The total developmental effort includes 6468 lines of C code to implement the RapidPatch runtime (including the inline assembly code), 1733 lines of Python to implement the RapidPatch toolchain, as well as roughly 1900 lines of C and python to implement several test applications and evaluation tools.

RapidPatch toolchain. The RapidPatch toolchain includes patch generator that processes and compiles the eBPF patch with the Clang bpf tool and patch verifier that utilizes static analysis to verify the safety of the patches. Specifically, before compilation, the patch generator scans the eBPF source code, which may use macros to represent variables or functions, and the patch configuration file, which defines the macros and describes the detailed variable/function names or statement lines in the corresponding C code. After that, the patch generator locates the variable register or addresses, function addresses, or basic block addresses by analyzing the symbol files (generated by `objdump`) and replaces the macros in the eBPF source code with their actual addresses or registers. Finally, the patch generator finds the patch installation address by locating the corresponding function name or the basic block entrance based on the symbol files.

Before installing the patch, the patch verifier needs to check if the patch is safe to deploy by analyzing the instructions used by that patch. Specifically, we only allow filter patches to write limited addresses inside the eBPF stack. Further, to constrain loops in filter patches, we can set a threshold for the number of iterations based on the MCUs' frequency and the latency requirements of devices (for instance, setting a larger threshold for the low priority tasks that are delay-tolerant). If the patch contains risk-sensitive behaviors such as memory writing and function call, patch verifier alerts the device maintainers that it is not a safe patch ready for instant deployment.

RapidPatch runtime. The runtime contains the hotpatching module and the eBPF module. We port the Linux eBPF VM and make it executable independently on various platforms using only a small part of libc that is also supported by the Newlib [17] on embedded devices. We also implement several types of eBPF maps for different scenarios. Note that, different from the original Linux eBPF map, we do not need to copy map data from user space to kernel space as most embedded systems do not have MMU and the data can be directly shared. In addition, we implement assemble, disassemble and compile tools for eBPF code using Python and Clang, which allow users to test the eBPF programs to ensure the correctness of the code.

We implement dynamic patch triggers leveraging FPB and KProbe, which use the hardware breakpoint features in ARM Cortex-m series. Similar to InstaGuard [33], we use hardware breakpoints to trigger the debug monitor exception to load patches. The debug event will be triggered when the instruction of the program counter (PC) hits the hardware breakpoint and enters the debug monitor exception handler.

	All CVE			High Risk CVE		
	#CVE	#Fix	#Filter	#CVE	#Fix	#Filter
Zephyr	29	24	17	18	16	13
FreeRTOS	13	13	11	6	6	4
Libraries	20	19	17	18	17	15
Total	62	56	35	42	37	32
Precent	100%	90.3%	56.5%	100%	88.1%	76.2%

Table 2: Patching various CVEs using RapidPatch.

Since PC will not move to the next instruction in the debug mode, we need to either modify the instruction address in the PC register or disable the hardware breakpoint temporarily to continue the execution of the program. In addition, we should enable the breakpoints again after exiting the debug monitor handler to make sure our patch system can still be triggered at this patch point in the following function calls. To solve this issue, we return and skip the breakpoint by modifying the PC for `OP_DROP` and `OP_REDIRECT`. For `OP_PASS`, the program needs to be continued in the buggy function from the breakpoint instruction. Thus, we temporarily reset the hardware breakpoints to exit the FPB debug mode.

Our eBPF module supports both interpreter and JIT mode. The original Linux eBPF JIT compiler is designed for instruction sets such as ARM-32, and does not support Cortex-m's ARM Thumb instruction set. We implement the JIT compiler for Cortex-m3+ MCUs that use the ARM Thumb-2 instruction set. As the Thumb-2 instruction set supports only 16-bit and 32-bit instructions, we map two 32-bit registers to one 64-bit eBPF register and translate most eBPF instructions into at least two Thumb-2 instructions. Meanwhile, using more 16-bit Thumb-2 instructions reduce storage consumption. We implement SFI in both eBPF interpreter and JIT compilers. In the interpreter mode, we can count and limit the total number of instructions and loop iterations. In the JIT mode, we stop compilation if the filter patch contains memory-writing instructions and add extra instructions to check and limit the loop iterations.

6 Performance Evaluation

We first demonstrate the generality and flexibility of RapidPatch and the adaptability of hotpatching using five devices. After that, we perform system evaluations to measure the delay of hotpatching and the overhead it incurs.

6.1 Applicability of RapidPatch

We collect CVEs of major embedded systems and develop RapidPatch patches for them to measure how many vulnerabilities can be fixed by RapidPatch.

CVE Dataset. We build a CVE dataset (shown in Table 7) by collecting all public CVEs with detailed vulnerability descriptions and corresponding vulnerable code from the open-access CVE database [16]. We cannot find any public CVEs for NuttX [7] and LiteOS [14], and some CVEs are excluded due to the lack of source code, such as the Treck TCP/IP stack vulnerabilities in RIPPLE20 [21] and several CVEs in FreeRTOS and MbedTLS. As listed in Table 7, we analyze the vulnerable code of 62 CVEs of RTOSes including FreeRTOS [5], Zephyr OS [26], and Libraries (such as MbedTLS [15], WolfSSL [25], and the AMNESIA33 [6] vulnerabilities).

We confirm if the vulnerabilities in our CVE dataset can be fixed by RapidPatch. The statistics indicate that more than 90% of CVEs can be fixed by either filter patch or code replace patch. Specifically, 42 CVEs are marked as High (risk score 7.0-9.0) or Critical (risk score 9.0-10.0) based on the severity ranking by NVD [16]. Among all these 42 high or critical risk CVEs, 37 (around 88%) can be fixed, and 32 (76%) can be fixed by filter patches. The results show that filter patches are effective in preventing the majority of vulnerabilities, such as out-of-range access or lack of sanity checks, which could lead to serious problems such as remote code execution (RCE).

There are several special cases that are not suitable for hotpatching. 6 out of 62 CVEs cannot be handled by RapidPatch due to one of the following two reasons. First, some patches modify the macros or struct definitions, impacting the entire firmware. For instance, fixing CVE-2017-14202 in ZephyrOS needs to increase the memory size for the shell history items by modifying the corresponding macro. Second, the original C patch is trying to fix too many vulnerable functions at the same time. A typical case is CVE-2020-10064 in Zephyr OS, which modifies a dozen of IEEE 802.15.4 frame processing functions. Hotpatching this many vulnerabilities simultaneously could exhaust the hardware resources.

6.2 Adaptability of RapidPatch

RapidPatch runtime is a portable C library that only relies on a few POSIX APIs (such as malloc/free) and can be easily ported to different RTOSes. We have successfully ported RapidPatch runtime to Amazon FreeRTOS, Zephyr OS, NuttX, and LiteOS (five most commonly used RTOSes [11]) with an average code modification of 47 lines (including the modification in Makefile and CMake). RapidPatch runtime is used as a new component for these RTOSes and developers can use the existing menuconfig tool to enable or disable this feature.

We choose three types of most used connective devices to evaluate the portability of RapidPatch runtime: (i) the NRF52840 development board with built-in wireless support in the MCU, (ii) STM32L475, STM32F429 and GD32VF103 which only support wired connection via SPI interfaces, and (iii) ESP-WROOM32, a dedicated WIFI/bluetooth MCU. The detailed specification of these devices is shown in Table 4. Currently we only implement JIT mode in ARMv7-m used by the Cortex-M3/M4 MCU. The RapidPatch runtime needs

	CVE-ID	OS/Lib	Vulnerability type	Bug description	Patch Type	Patch lines
c1	2020-10063	ZephyrOS	Integer Overflow	The uint16_t overflow in CoAP function _parse_option	Filter	12
c2	2020-10021	ZephyrOS	Out-of-Bounds Write	Usb massive storage over range write	Filter	14
c3	2020-10023	ZephyrOS	Logical Bug	Incorrect logic for string strip in shell	Replace	52
c4	2020-10024	ZephyrOS	Instruction Misuse	Misuse the signed comparison for the unsigned	Filter	11
c5	2020-10028	ZephyrOS	Lack Sanity Checking	The GPIO handlers do not verify the arguments	Filter	15
c6	2020-10062	ZephyrOS	Logical Bug	Packet length decoding error in MQTT	Replace	38
c7	2018-16524	FreeRTOS	Division by Zero	The RxWindowLength may divide zero (MSS = 0)	Filter	59
c8	2018-16528	FreeRTOS	State Confusion	Recv before the mbedTLS success to initialize	Filter + Map	6 + 12
c9	2018-16603	FreeRTOS	Out-of-Bounds Read	xProcessReceivedTCPPacket read over range	Filter	14
c10	2017-2784	mbedTLS	Invalid Free	Free the pointer in mbedtls_mpi before initializing	Filter	5
c11	2020-17443	AMNESIA33	Lack Packet Checks	ICMPv6 echo request header length has no limit	Filter	13
c12	2020-17445	AMNESIA33	Lack Option Checks	IPv6 header's dest options lengths not checked	Filter	25

Table 3: The eBPF patch for different CVE types.

Device MCU	Arch	Frequency	Flash	SRAM
NRF52840	Cortex-M4	64MHz	1MB	256KB
STM32L475	Cortex-M4	80MHz	512KB	128KB
STM32F429	Cortex-M4	180MHz	2MB	256KB
ESP-WROOM32	Xtensa	240MHz	448KB	520KB
GD32VF103	RISC-V32	108MHz	128KB	32KB

Table 4: The specifications of evaluated embedded devices.

about 13KB flash memory and 5KB RAM. Considering the RTOS and user applications typically need roughly 45KB flash memory, RapidPatch can work on devices with only 64KB or more flash memory.

Hotpatching Strategies on Different Devices. We use CVE-2018-16603 from FreeRTOS to test RapidPatch on the five devices. In ARM Cortex-M4 devices, both dynamic patch points (i.e., FPB and KProbe) and fixed patch points can be used to trigger the patch. Note that KProbe is not supported by Xtensa MCUs [13]. It can work on RISC-V MCUs [22] but we have not implemented it yet. Finally, we use fixed patch points to trigger the patch. The vulnerable function is patched successfully on these devices with a delay of 12.9 μs $\sim$ 28.1 μs in interpreter mode. We compare the delays of all the three hotpatching strategies on Cortex-M4 in Section 6.3.

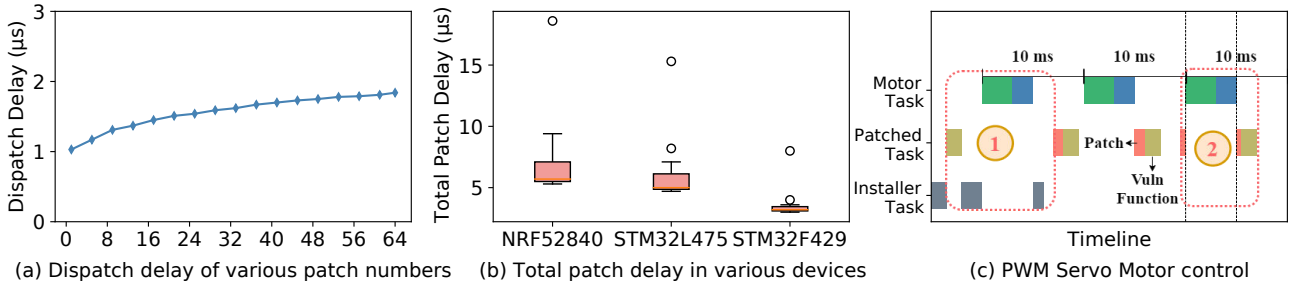
The Espressif devices with Xtensa architecture (e.g., ESP32) are widely used as addons for other MCUs without wireless functions. We use it to evaluate the practicality of fixed patch points and measure the delay and storage overhead with different numbers of fixed patch points. We develop a source code instrument tool to add a macro that saves the context and calls the patch dispatcher at the entrance of functions. Note that the tool can only hook the functions which are non-inline, or return a value type (e.g., int, pointers) or void, or use assembly code. In FreeRTOS on the ESP32 development board, we add fixed patch points to 1442 out of 4372 functions and the size of firmware increases by 31KB, resulting in a storage overhead of 3.8%. For Zephyr OS, we add fixed patch points to 1051 out of all the 1861 functions and need extra 14KB flash storage, resulting in a storage overhead of 7.0%. Nevertheless, if only the network stacks are instrumented, the incurred storage overhead is negligible. Since the functions with fixed patch points need to execute extra instructions to call the patch dispatching functions, we use a bitmap to quickly check if there are active patches at these functions. We evaluate the performance impact in Section 6.4 and the applications slow down 2.15 $\sim$ 9.14% when all the subsystem functions are instrumented. Note this hotpatching approach can also be used in MIPS, AVR MCUs.

6.3 Patching Delay

We first follow the RapidPatch runtime's design in Figure 7 to perform a micro-benchmark of each patch execution phase on both Zephyr OS and bare-metal firmware using the NRF52840 device. Then we evaluate the delay of the entire patching process on various devices.

Patch Loading Delay. Since the patch loading process only contains fixed instructions for triggering the patch and saving/restoring the context, which is consistent among different patches for the same patching strategies, we can measure the standalone loading delay by excluding the patch dispatching and executing phases from the whole patching process. We ran each patch triggering strategy for 10 times to count the average CPU cycles on the NRF52840 device. The `OP_PASS` operation needs to reset the hardware breakpoint status (see Section 5) to avoid repeatedly executing the breakpoint instruction. As a result, when using FPB and KProbe that are triggered by hardware breakpoints, the patch loading time changes in different redirection operations. The results are shown in Table 5. All these strategies can finish within 400 CPU cycles (about 7 μs on NRF52840). The fixed patch points have less instructions than the KProbe/FPB and only needs 66 cycles. As a comparison, the UART `printf` function needs about 180000 CPU cycles.

Patch Dispatching Delay. The patch dispatching approach



(a) Dispatch delay of various patch numbers (b) Total patch delay in various devices (c) PWM Servo Motor control

Figure 10: The patching delays on different devices.

	Fixed Patch Point		FPB		Debug Monitor	
OP	Cycles	Time	Cycles	Time	Cycles	Time
No Patch	66	1.03	0	0	0	0
Pass (Continue)	66	1.03	395	6.17	252	3.94
Drop / Redirect	66	1.03	120	1.87	135	2.1

Table 5: The CPU cycles and time required by various patch triggering methods for a single vulnerability in NRF52840.

selects the appropriate patch from all the active patches. We measure the time consumption of the patch dispatching function with different numbers of active patches. Due to memory constraints, most devices can only install a limited number of patches, and therefore we set 64 as the maximum active patch number. As is shown in Figure 10 (a), the dispatching time increases with the patch number and need about 2.0 μs for 64 patch points.

eBPF Code Execution Delay. The patch execution time can be evaluated independently by directly running the patch code. Since the delay of patch execution depends on the detailed eBPF code (e.g., instruction number, time complexity) and the inputs, we set different arguments to run the patches for different CVEs types, as shown in Table 3. The filter patch's delay is negligible as it only needs to check the inputs and the loop count is limited by SFI. Nevertheless, the replace patch can use all kinds of logic and its delay depends on the detailed implementation. We run the eBPF patches for the CVEs in Table 3 and measure the time consumption in both eBPF interpreter mode and JIT mode on the NRF52840 device. As shown in Table 6, when running in interpreter mode, the patch delay is 8 $\sim$ 30 μs and this duration can be smaller than 2 μs in JIT mode. However, calling C function in eBPF code is time consuming (about 300 extra cycles in JIT mode). As an example, the patch code for CVE-2020-10023 calls function `memmove` for tens of times and needs 14.7 μs while the original C function only needs 5.6 μs.

Comparison with HERA. We reimplement HERA's patching approach in our STM32L475 device using Cortex-M4's FPB flash patching feature and measure its patching delay. We use FreeRTOS's CVE-2018-16601 as an example. On

CVE	# of eBPF Instructions	eBPF Interpreter	eBPF-Jit	Memory (Bytes)
c1	8	27.3 μs	1.7 μs	56
c2	16	8.5 μs	1.6 μs	48
c3	100	133.3 μs	14.7 μs	260
c4	12	9.5 μs	2.0 μs	68
c5	14	23.5 μs	1.5 μs	48
c6	55	51.2 μs	4.4 μs	232
c7	46	26.8 μs	1.8 μs	188
c8	10+10	14.9 μs +16.2 μs	2.8 μs +2.7 μs	56+68
c9	10	28.1 μs	1.8 μs	52
c10	7	10.1 μs	1.4 μs	48
c11	7	9.5 μs	1.6 μs	48
c12	36	22.2 μs	3.9 μs	156

Table 6: The execution times of eBPF patches on NRF52840.

STM32L475, when dispatching 1 to 5 patches in the dispatcher, HERA needs 0.3 $\sim$ 0.85 μs, and RapidPatch needs 3.2 $\sim$ 4.3 μs since it needs to return to the original function for `OP_PASS`. For patch execution, HERA needs 0.25 μs to execute the extra patching instructions while RapidPatch needs 1.5 μs to execute the patch in JIT mode. Overall, RapidPatch has slightly higher, yet comparable, patching overhead compared with HERA.

Total Patching Delay. We use the CVEs in Table 3 to measure the total patching delays by applying KProbe patch triggers on various devices with Zephyr OS and bare-metal firmware. As shown in Figure 10(b), the average patching delay is less than 7.5 μs, and MCUs with higher frequencies have smaller delays. Since the patch loading delays depend on the trigger strategies and have a fixed delay of less than 6 μs, the entire patching delay is mainly decided by the detailed patching code. For the filter patches, we can limit the total loop iterations via SFI, and therefore can achieve bounded delays of 8 μs. We find that the patching delays are not affected by whether the devices are installed with RTOSes. Instead, the task schedules of the OS have more impacts.

Patching Real-Time Tasks. To further measure how other tasks may affect the patching process, we execute the hot-patching task under the Servo Motor controlling scene. As shown in Figure 10(c), the Servo Motor is controlled by the pulse-width modulator (PWM) timer exception in a fixed cy-

cle of 10ms. The motor task has a hard real-time constraint to control the motor every 10ms and thus has the highest priority. In our evaluation, the task to be patched (i.e., the patched task) runs with a medium priority. We set the patch installer task with the lowest patching priority, so it does not interrupt any other tasks. The patch installer task is shown in ① and it is interrupted by the motor task. As the patch installer task completes successfully, the patch is triggered immediately when the vulnerable function is invoked. The patch installation and execution are finished within $10 \sim 20$ ms since it is split into multiple CPU time slices. RapidPatch allows other task to intercept the patch execution at any time. The patch triggering processes can also be interrupted. In particular, the FPB and fixed patch points obviously do not block the tasks with high priority [35]. Since KProbe is implemented by the debug monitor [58], we can set a low exception priority via the nested vector interrupt control (NVIC). Thus, RapidPatch will not disrupt device operations with real-time constraints.

6.4 System Overhead of Patching

We evaluate the patching system's overhead under different patching strategies. Although the delay of a single eBPF patch execution is around 6 μs in the JIT mode and 20 μs in the interrupter mode, the total delay can be accumulated, which is decided by how frequently the patched function is executed. We patch the critical paths in the network stack triggered by the ingress packets and measure the end-to-end latency of the network applications.

Dynamic Patch Triggers. We first use dynamic patch triggers to evaluate the overheads in different scenarios. We choose three CVEs from different subsystems of Zephyr OS, including CoAP library (CVE-2020-10063), MQTT library (CVE-2020-10062) and the USB massive storage library (CVE-2020-10021). We use three corresponding demo apps running on the NRF52840 device, each containing the vulnerable function of the specific CVE in its critical execution path. For each app, we install a filter patch triggered by KProbe on the buggy function and send thousands of requests to measure the end-to-end interaction delay incurred by RapidPatch. The CoAP demo app works as a server, using a single thread to connect to a remote client, and we measure the request-response duration for every request. The MQTT demo app functions as a subscriber and connects to a broker to keep sending PUBACK messages after receiving PUBLISH packets. We record the delay between every PUBLISH-PUBACK pair. In the USB massive storage app, the board connects to the machine via a USB cable, and responds to a few read/write commands sent from the machine in every iteration. We measure the duration of every iteration. We use the original system as baselines and compare its performance with eBPF interpreter-patched and eBPF JIT-patched modes.

As shown in Figure 11, the patching delay under all three application scenarios is negligible, no matter using an eBPF interpreter mode or an eBPF JIT mode. The latency distributions of both eBPF JIT patches and eBPF interpreter-patches are quite close to the original system, while the eBPF JIT mode imposes slightly lower latency than the interpreter mode. In particular, the delay incurred by the eBPF interpreter mode for all three demo apps ranges $0.06\% \sim 1.5\%$, and the delay incurred by the eBPF JIT mode ranges $0.01\% \sim 0.6\%$. Such extra delays are acceptable because even for the most delay-sensitive demo app, i.e., CoAP app, all the request-response latencies are milliseconds, which is much higher than the single patch execution delay (on the order of microseconds as discussed in Section 6.3). Overall, the above results demonstrate that a single patch with dynamic patch trigger introduces negligible impacts on embedded devices.

Fixed Patch Points. Since the fixed patch points can affect all the functions instrumented in RTOSes, an increasing number of functions equipped with fixed patch points may cause a higher delay to the system. As shown in Figure 12, we add different scales of fixed patch points in FreeRTOS and Zephyr OS, i.e., only the network stacks, all the subsystems, and the whole system. We install a patch to a vulnerable function of the network stack and send 10K requests to measure the average latency of CoAP Apps. On the NRF52840 device with Zephyr OS, we instrument 342 functions of the network stack (18.4% of all functions in the firmware) and the incurred latency overhead is about 1.84%. When instrumenting 401 functions of all subsystems (21.5%), the incurred latency overhead is about 2.15%. When instrumenting 1051 functions of the whole system (56.5%), the incurred latency overhead is about 18.8%. On the ESP32 device[1] running FreeRTOS, when 195 functions of the network stack (4.5%) are instrumented, the incurred latency overhead is about 0.42%. When 499 functions of all subsystems (11.4%) are instrumented, the incurred latency overhead is about 9.14%. When 1441 functions of the whole system (33%) are instrumented, the incurred latency overhead is about 12.1%. The latencies are increasing with the number of patch points even though there are no active patches installed at these places. The patch triggering and execution delays of each active patch are similar to the cases using dynamic patch points.

Discussion. Since the patch downloading, JIT compiling, and installing processes can run as background tasks, the major performance impacts of our patching system are caused by patch triggering and patch execution. When there are no patches installed, the dynamic triggers are vetted by the hardware and incur negligible overhead while the fixed patch points incur more significant delays as they are inserted into the functions and called repeatedly. As shown in Figure 12, our fixed patch points' instrumentation approach has a similar overhead as the stack canary [36] which only needs to load the global canary value twice from the SRAM to make a comparison. In our experiments, on different RTOSes, if we only

[1] We set the MCU frequence to 80 MHz.

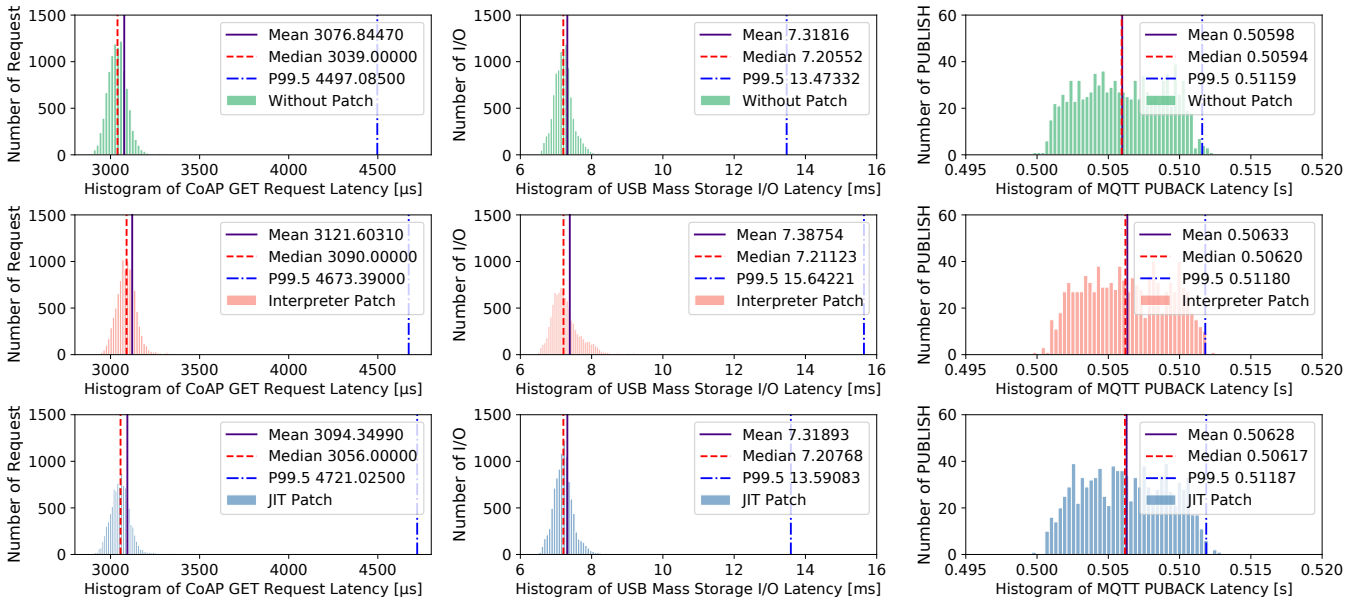


Figure 11: The incurred delays by RapidPatch on NRF52840DK with Zephyr OS.

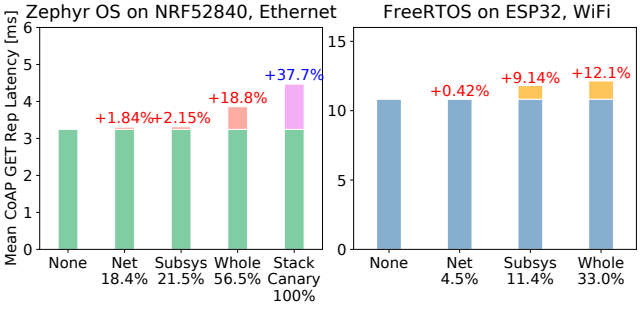


Figure 12: Overhead with various scales of fixed patch points.

place fixed patch points within the most risky code blocks (i.e., the network stacks), the overhead is less than 2%. When placing fixed patch points all around the systems, especially in the RTOS's core functions, the total overhead is up to 20%. The results on different devices and various RTOSes indicate that placing multiple active patches have little impact on the system's performance, because the execution of a single patch only incurs less than 20 μs delay (as shown in Section 6.3). In fact, the total installed patch number is mostly dominated by the memory size, rather than performance degradation.

7 Related Work

IoT Firmware Update. Normally, over-the-air (OTA) updating [19] is utilized to add new features in IoT systems or fix software bugs (and vulnerabilities) by fully or incrementally modifying the on-chip firmware. Existing works [29, 32, 39, 51] proposed reprogramming methods to reduce the transmitted data during updating. Several frameworks [27, 49] have been proposed to secure OTA updating, including the firmware generation, propagation, verification, and installation phases. However, OTA updating operations require reprogramming the NOR-Flash memory, which means erasing and rewriting partial code blocks of the system. As we have discussed, writing the execution code on flash definitely will halt the corresponding task threads. Although dynamic linking and relocating the RTOSes tasks/libraries can eliminate the system halt [47], it cannot be applied to existing RTOSes, most of which use static linked tasks. Instead of directly modifying the running code on flash, our approach leverages hardware debug features to dynamically load patches or only modify the quiescent code to avoid interfering with the system execution during system update.

Hardware Based Hotpatching. Different from the traditional hotpatching techniques, hardware based hotpatching triggered by hardware debug units can avoid injecting new instructions to vulnerable programs or the corresponding memory space [33, 35]. For instance, Instaguard [33] leverages hardware to enable a rule-driven hotpatching mechanism for Android mobile devices, which can significantly reduce the delay of system patching. Nevertheless, it cannot be applied on IoT devices that run RTOS applications because of the following reasons. First, most of RTOSes only run with one process with various threads in charge of separate task loops, and thus we cannot simply terminate the affected process by using their mechanism. Second, according to our studies on RTOS related CVEs, many CVEs are raised by logic errors in RTOSes, which cannot be fixed by applying rule-based filtering in Instaguard. RapidPatch well addresses this issue

by enabling dynamic code replacement with eBPF. Although HERA [35] can fix such bugs by leveraging the FPB to redirect the vulnerable code to the new binary code, it can only be applied in ARM Cortex-M3/M4 devices. RapidPatch enables a generic approach for different platforms by utilizing both fixed and dynamic patch triggers in various platforms. Particularly, RapidPatch allows dynamic update when the vulnerable function has states shared with other functions, e.g., to fix CVE-2018-16528 (i.e, the vulnerabilities in TLS connectivity modules), which has not been addressed via stateful patches.

Patch Analysis. Patch analysis for vulnerability discovery has been studied in the literature [61]. Fore example, Zhang et. al. propose FIBER [52, 61], a tool to analyze an Android kernel and check if the source code of the kernel have merged with a patch. Pewny et. al. [52] can verify if a bug exists in software by semantic comparison based binary analysis. Their method can also identify bugs across different architectures and find the heartbleed vulnerability in three types of instruction sets (i.e., x86, ARM, and MIPS). These works orthogonal to RapidPatch. We can leverage these tools to identify versions of vulnerable RTOS systems.

System Enhancement with eBPF. In Linux, eBFF are usually used to implement new network functionalities, e.g., performance analysis, network acceleration, and intrusion detection [44–46]. In particular, eBPF has been recently used to develop different defense systems with low processing overhead [38, 40–42, 48, 54, 55]. To best of our knowledge, RapidPatch is the first OS-independent RTOS library that leverage eBPF to enable safe hotpatching for various vulnerable RTOS applications with different instruction set architectures.

8 Discussion

Existing hotpatching solutions in Linux/Android are not applicable on embedded devices due to their resource limitations and requirements on real-time patching. Similar to HERA, RapidPatch can hotpatch the RTOS without modifying the flash (which could impose hundreds of milliseconds delays since the flash sectors need to be first erased before rewriting), and therefore avoid violating the real-time constraints. Moreover, to address the limitations of HERA, RapidPatch provides new patching strategies to support more devices and use bytecode patches to automatically adapt to heterogeneous devices. Yet, RapidPatch still has some common limitations of hotpatching. First, RapidPatch is applied to fix small bugs rather than perform large feature updates and it may fail in certain cases (discussed in Section 6.1). Second, even though the filter patches do not require extra tests, the eBPF verifier can only ensure that the fault patches do not damage the system, without providing guarantees that the patches are bug-free. Third, the device maintainers still need to manually obtain the patches from the providers (e.g., the RTOS/Library developers) and deploy the patches via RapidPatch. The RapidPatch is suitable for the vendors to quickly deploy temporary patches to prevent the exploitation of vulnerabilities. To better protect the devices, they need to regularly renew the versions of RTOSes/Libraries for the firmware.

9 Conclusion

In this paper, we propose RapidPatch, a novel hotpatching framework aiming at accelerating the patch development and deployment for real-time embedded devices. RapidPatch allows different vendors to share the same source code patch among all the heterogeneous devices for the same vulnerability and the generated patches can be instantly installed by all downstream devices. We prototype RapidPatch that can be deployed on different types of embedded devices. In particular, by porting the eBPF VM JIT mode to these devices, RapidPatch enables hotpatching the devices with a negligible delay. We evaluate RapidPatch on major CVEs on diferent RTOSes with various devices to demonstrate the applicability, genericity, and performance of RapidPatch.

Acknowledgments

We would like to thank our shepherd Ahmad Reza Sadeghi and the anonymous reviewers for their comments. This work is supported in part by the National Key R&D Program of China under Grant 2018YFB1800304, NSFC under Grant 62132011, 61825204, U20B2049, 61822207, 61822309, and 61773310, Beijing Outstanding Young Scientist Program under Grant BJJWZYJH012019100030011, Shaanxi Province Key Industry Innovation Program under Grant 2021ZDLGY01-02, and BNRist under Grant BNR2020 RC01013. Kun Sun's work is supported in part by the U.S. Department of the Army grant W56KGU-20-C-0008 and the National Science Foundation under Grant CNS-1822094. Qi Li is the corresponding author of this paper.

References

[1] The RapidPatch source code. https://github.com/IoTAccessControl/RapidPatch.

[2] A thorough introduction to eBPF. https://lwn.net/Articles/740157/, (Last Accessed, May 3, 2021).

[3] A/B (Seamless) System Updates. https://source.android.com/devices/tech/ota/ab, (Last Accessed, May 3, 2021).

[4] About the flash patch and breakpoint unit in arm cortex-m3/m4. https://developer.arm.com/documentation/ddi0337/h/debug/about-the-flash-patch-and-breakpoint-unit--fpb-, (Last Accessed, May 3, 2021).

[5] Amazon FreeRTOS. https://aws.amazon.com/freertos/, (Last Accessed, May 3, 2021).

[6] AMNESIA:33 – Foresout Research Labs Finds 33 New Vulnerabilities in Open Source TCP/IP Stacks. "https://www.forescout.com/company/blog/amnesia33-forescout-research-labs-finds-33-new-vulnerabilities-in-open-source-tcp-ip-stacks/", (Last Accessed, May 3, 2021).

[7] Apache NuttX. https://nuttx.apache.org/, (Last Accessed, May 3, 2021).

[8] eBPF XDP. https://blogs.igalia.com/dpino/2019/01/07/a-brief-introduction-to-xdp-and-ebpf/, (Last Accessed, May 3, 2021).

[9] Espressif's A/B Systems based over the air update (OTA). https://docs.espressif.com/projects/esp-idf/en/latest/esp32/api-reference/system/ota.html, (Last Accessed, May 3, 2021).

[10] Hacking the Printer. https://www.hackread.com/28000-exposed-printers-hacked-over-lack-printer-security/, (Last Accessed, May 3, 2021).

[11] IoT Developer Survey 2019 Results and 2020 IoT Developer Survey Key Findings. https://iot.eclipse.org/community/resources/iot-surveys/, (Last Accessed, May 3, 2021).

[12] Kernel Probes (Kprobes). https://www.kernel.org/doc/Documentation/kprobes.txt, (Last Accessed, May 3, 2021).

[13] KProbe is not supported by Xtensa yet. https://www.kernel.org/doc/html/latest/xtensa/features.html, (Last Accessed, May 3, 2021).

[14] LiteOS. https://www.huawei.com/minisite/liteos/en/, (Last Accessed, May 3, 2021).

[15] Mbed TLS. https://github.com/ARMmbed/mbedtls, (Last Accessed, May 3, 2021).

[16] National vulnerability database. https://nvd.nist.gov/, (Last Accessed, May 3, 2021).

[17] Newlib: Libc for embedded systems. https://sourceware.org/newlib/, (Last Accessed, May 3, 2021).

[18] nRF52 Series. https://www.nordicsemi.com/-/media/Software-and-other-downloads/Product-Briefs/nRF52-Series-SoC-PB-50.pdf, (Last Accessed, May 3, 2021).

[19] Over-the-Air (OTA) Updates in Embedded Microcontroller Applications. https://www.analog.com/en/analog-dialogue/articles/over-the-air-ota-updates-in-embedded-microcontroller-applications.html, (Last Accessed, May 3, 2021).

[20] RFC for IoT Device Classes: Terminology for Constrained-Node Networks. https://tools.ietf.org/html/rfc7228#page-8, (Last Accessed, May 3, 2021).

[21] Ripple20: 19 Zero-Day Vulnerabilities Amplified by the Supply Chain. https://www.jsof-tech.com/ripple20/, (Last Accessed, May 3, 2021).

[22] RISC-V KProbe Feature. https://www.kernel.org/doc/html/latest/riscv/features.html, (Last Accessed, May 3, 2021).

[23] The PicoTCP Library. http://picotcp.altran.be/, (Last Accessed, May 3, 2021).

[24] There is no flash patch feature in the cortex-m7 processor. https://www.keil.com/appnotes/files/apnt_270.pdf, (Last Accessed, May 3, 2021).

[25] wolfSSL Embedded SSL/TLS Library. https://github.com/wolfSSL/wolfssl, (Last Accessed, May 3, 2021).

[26] Zephyr OS. https://www.zephyrproject.org/, (Last Accessed, May 3, 2021).

[27] Mahmoud Ammar and Bruno Crispo. Verify&revive: Secure detection and recovery of compromised low-end embedded devices. In *ACM ACSAC*, 2020.

[28] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B. Rasmussen. The KNOB is broken: Exploiting low entropy in the encryption key negotiation of bluetooth br/edr. In *28th USENIX Security*, 2019.

[29] Konstantinos Arakadakis, P. Charalampidis, Antonis Makrogiannakis, and Alexandros Fragkiadakis. Firmware over-the-air programming techniques for IoT networks - a survey. *ArXiv*, abs/2009.02260, 2020.

[30] Jeff Arnold and M Frans Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proceedings of the 4th ACM European conference on Computer systems*, pages 187–198, 2009.

[31] Adrian Baranchuk, Marwan M. Refaat, Kristen K. Patton, Mina K. Chung, Kousik Krishnan, Valentina Kutyifa, Gaurav Upadhyay, John D. Fisher, and Dhanunjaya R. Lakkireddy. Cybersecurity for cardiac implantable electronic devices: What should you know? *Journal of the American College of Cardiology*, 71(11):1284 – 1288, 2018.

[32] J. Bauwens, P. Ruckebusch, S. Giannoulis, I. Moerman, and E. D. Poorter. Over-the-air software updates in the internet of things: An overview of key principles. *IEEE Communications Magazine*, 58(2):35–41, 2020.

[33] Yaohui Chen, Yuping Li, Long Lu, Yueh-Hsun Lin, Hayawardh Vijayakumar, Zhi Wang, and Xinming Ou. Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In *NDSS'18*, 2018.

[34] Yue Chen, Yulong Zhang, Zhi Wang, Liangzhao Xia, Chenfu Bao, and Tao Wei. Adaptive android kernel live patching. In *26th USENIX Security*, 2017.

[35] Lucas Davi Christian Niesler, Sebastian Surminski. Hera: Hotpatching of embedded real-time applications. In *NDSS*, 2021.

[36] Thurston H.Y. Dang, Petros Maniatis, and David Wagner. The performance cost of shadow stacks and stack canaries. In *10th ASIA CCS*, 2015.

[37] B. Dasarathy. Timing constraints of real-time systems: Constructs for expressing them, methods of validating them. *IEEE Transactions on Software Engineering*, 11(01):80–86, jan 1985.

[38] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. Sysfilter: Automated System Call Filtering for Commodity Software. In *RAID*, 2020.

[39] W. Dong, B. Mo, C. Huang, Y. Liu, and C. Chen. R3: Optimizing relocatable code for efficient reprogramming in networked embedded systems. In *2013 Proceedings IEEE INFOCOM*, pages 315–319, 2013.

[40] William Findlay, David Barrera, and Anil Somayaji. Bpfcontain: Fixing the soft underbelly of container security. *arXiv preprint arXiv:2102.06972*, 2021.

[41] William Findlay, Anil Somayaji, and David Barrera. bpfbox: Simple precise process confinement with ebpf. In *Proceedings of the 2020 ACM SIGSAC Conference on Cloud Computing Security Workshop*, pages 91–103, 2020.

[42] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal system call specialization for attack surface reduction. In *29th USENIX Security*, 2020.

[43] C. Giuffrida, C. Iorgulescu, G. Tamburrelli, and A. S. Tanenbaum. Automating live update for generic server programs. *IEEE Transactions on Software Engineering*, 43(3):207–225, 2017.

[44] Sasha Goldshtein. The next linux superpower: ebpf primer. Dublin, July 2016. USENIX Association.

[45] Brendan Gregg. Performance superpowers with enhanced BPF. Santa Clara, CA, July 2017. USENIX Association.

[46] Brendan Gregg. BPF performance tools. Portland, OR, October 2019. USENIX Association.

[47] Simon Holmbacka, Wictor Lund, Sébastien Lafond, and Johan Lilius. Lightweight framework for runtime updating of c-based software in embedded systems. In *5th Workshop on Hot Topics in Software Upgrades (HotSWUp 13)*, San Jose, CA, June 2013. USENIX Association.

[48] Taesoo Kim and Nickolai Zeldovich. Practical and effective sandboxing for non-root users. In *USENIX ATC 13)*, 2013.

[49] Antonio Langiu, Carlo Alberto Boano, Markus Schuß, and Kay Römer. Upkit: An open-source, portable, and lightweight update framework for constrained iot devices. In *39th ICDCS*, 2019.

[50] A. Machiry, N. Redini, E. Camellini, C. Kruegel, and G. Vigna. Spider: Enabling fast patch propagation in related software repositories. In *IEEE S&P*, 2020.

[51] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff. Zephyr: Efficient incremental reprogramming of sensor nodes using function call indirections and difference computation. In *USENIX ATC*, 2009.

[52] J. Pewny, B. Garmany, R. Gawlik, C. Rossow, and T. Holz. Cross-architecture bug search in binary executables. In *IEEE S&P*, pages 709–724, 2015.

[53] Florian Rommel, Christian Dietrich, Daniel Friesel, Marcel Köppen, Christoph Borchert, Michael Müller, Olaf Spinczyk, and Daniel Lohmann. From global to local quiescence: Wait-free code patching of multi-threaded processes. In *14th USENIX OSDI, 2020*.

[54] Chengyu Song, Chao Zhang, Tielei Wang, Wenke Lee, and David Melski. Exploiting and protecting dynamic code generation. In *NDSS*, 2015.

[55] D. J. Tian, G. Hernandez, J. I. Choi, V. Frost, P. C. Johnson, and K. R. B. Butler. Lbm: A security framework for peripherals within the linux kernel. In *IEEE S&P*, 2019.

[56] Qiushi Wu, Yang He, Stephen McCamant, and Kangjie Lu. Precisely characterizing security impact in a flood of patches via symbolic rule comparison. In *NDSS*, 2020.

[57] Zhengzi Xu, Yulong Zhang, Longri Zheng, Liangzhao Xia, Chenfu Bao, Zhi Wang, and Yang Liu. Automatic hot patch generation for android kernels. In *29th USENIX Security*, 2020.

[58] Joseph Yiu. Chapter 14.3: Debug modes. In *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*, pages 456–462. Newnes, USA, 3rd edition, 2013.

[59] Joseph Yiu. Chapter 23.10: Flash patch feature. In *The Definitive Guide to ARM Cortex-M3 and Cortex-M4 Processors, Third Edition*, pages 761–765. Newnes, USA, 3rd edition, 2013.

[60] C. Zhang, W. Ahn, Y. Zhang, and B. R. Childers. Live code update for iot devices in energy harvesting environments. In *2016 5th Non-Volatile Memory Systems and Applications Symposium (NVMSA)*, pages 1–6, 2016.

[61] Hang Zhang and Zhiyun Qian. Precise and accurate patch presence test for binaries. In *27th USENIX Security*, 2018.

[62] Zheng Zhang, Hang Zhang, Zhiyun Qian, and Billy Lau. An investigation of the android kernel patch ecosystem. In *30th USENIX Security*, 2021.

A CVE Case Study

Our in-depth CVE case study reveals common patterns of vulnerabilities and proves the effectiveness and necessity of our patch design. The most common (34/62) vulnerabilities in embedded network stacks are the lack of boundary or sanity checks when processing ingress packets, which can be efficiently fixed via filter patches. Taking CVEs of FreeRTOS as examples, CVE-2018-16602 (DHCP), CVE-2018-16603 (TCP), CVE-2018-16600 (ARP) and CVE-2018-16527 (ICMP) have the same root cause that the IP packet handler function `prvProcessIPPacket` fails to check whether a packet uses a valid protocol or has a valid packet length. We use a single filter patch at the entrance of function `prvProcessIPPacket` to validate the ingress IP packets by adding extra checks to ensure that the packets have valid headers and lengths that are coherent with their protocols. As such, we fix a bundle of vulnerabilities simultaneously. Actually, considering that the packets using various upper-layer protocols usually share the same lower-layer protocol and dispatching entrance, we can fix vulnerabilities with filter patches more efficiently.

Logical bugs in RTOSes reveal the necessity of code replace patch. These bugs may cause system failures even under legitimate inputs. For instance, CVE-2020-10023 arises due to wrongly passing *shift* (instead of *j*) when invoking `memmove` in function `shell_spaces_trim`. To fix it, we need to rewrite the passed parameters. While for CVE-2017-14201, the misuse of stack variable in function `dns_result_cb` leads to a Use after Free vulnerability. As such, we must modify the parameter construction, passing and interpretation logic of function `dns_result_cb` via code replace patches.

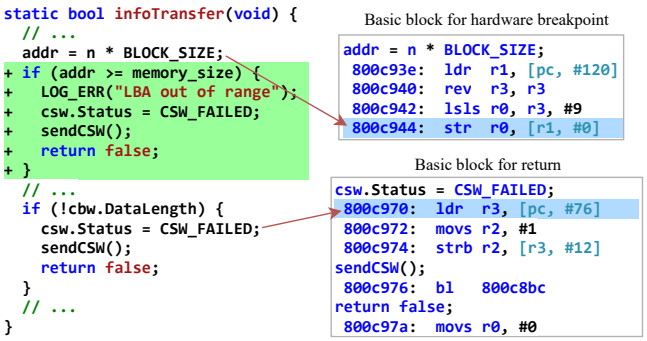


Figure 13: The patch entrance and exit basic blocks for CVE-2020-10021

```
patch_type: filter_patch

trigger_point:
  function_position: infoTransfer
  statement_position: addr = n * BLOCK_SIZE;

trigger_type: KPROBE

run_type: JIT

iteration_threshold: default

priortiy: default

variable_map:
  - MARCO_MEM_SIZE:
    address_type:  global_variable
    match_name: memory_size
```

Figure 14: Patch configuration for CVE-2020-10021

B Baisc Block-Level Patch

As is shown in Figure 13, the root cause of Zephyr's CVE-2020-10021 is that the address written by USB Mass Storage may exceed the memory size. Mitigating it requires adding extra checks to the write range. Right after the statement `addr = n * BLOCK_SIZE;` is executed, we can easily validate the address to be written stored in variable `addr` using register `r0`. The original source code patch is shown in green lines which handles the error by calling `sendCSW()` to notify the memory-write error. Instead of writing a code replace patch from the beginning, we redirect the control flow to reuse another existing error handling basic block. With the help of basic block level patch, RapidPatch has a very lightweight way of fixing the bugs by just blocking or replacing the vulnerable basic blocks. The patch configuration file for CVE-2020-10021 is shown in Figure 14.

C CVE Dataset

Table 7 shows the CVEs studied in this paper. Around 90.3% CVEs (i.e., 56/62) can be fixed by hotpatching, while only six CVEs cannot be fixed, which are marked as – in the Patch Type column.

CVE-ID	OS/Lib	Patch Description / Reason Why Cannot Fix	Patch Type
2018-16528	FreeRTOS	Record mbedtls_ssl_handshake status with eBPF map to prevent using unready mbedTLS context	Filter Patch + Map
2018-16522	FreeRTOS	Basic block replacement to initialize protocol with memset name buffer after malloc	Code Replace Patch
2018-16526	FreeRTOS	Rewrite function prvProcessIPPacket to update pxIPHeader->ucVersionHeaderLength	Code Replace Patch
2018-16525	FreeRTOS	Add UDP payload length field validation for prvProcessIPPacket function	Filter Patch
2018-16599	FreeRTOS	Add UDP payload length field validation for prvProcessIPPacket function	Filter Patch
2018-16601	FreeRTOS	Add IP header length field validation for prvProcessIPPacket function	Filter Patch
2018-16523	FreeRTOS	Check if uxNewMSS equals zero before use	Filter Patch
2018-16524	FreeRTOS	Add validation for pucLast against received packet buffer bound in function prvCheckOptions	Filter Patch
2018-16603	FreeRTOS	Check if the received frame is larger than the minimum packet size	Filter Patch
2018-16602	FreeRTOS	Add length checks for the DHCP option fields walking loop	Filter Patch
2018-16600	FreeRTOS	Check received frame size in function prvProcessEthernetPacket for ARP case	Filter Patch
2018-16527	FreeRTOS	Check received frame size in function prvProcessIPPacket for ICMP case	Filter Patch
2018-16598	FreeRTOS	Match outgoing DNS query with received answer using eBPF map	Filter Patch + Map
2017-14199	ZephyrOS	Add out-of-bound check for function dns_resolve_cb	Filter Patch
2017-14201	ZephyrOS	Modify the way to pass and interpret void *user_data for function dns_result_cb	Code Replace Patch
2017-14202	ZephyrOS	Macro and struct definitions modification are involved	–
2019-9506	ZephyrOS	Add filter point to ban low encryption key length	Filter Patch
2020-10019	ZephyrOS	Limit upload length to the size of the request buffer	Filter Patch
2020-10021	ZephyrOS	Check if LBA is within the range of memory_size	Filter Patch
2020-10022	ZephyrOS	Too many vulnerable functions are involved	–
2020-10023	ZephyrOS	Rewrite the logic for counting the length in function shell_spaces_trim	Code Replace Patch
2020-10024	ZephyrOS	Obtain C flag in APSR after the cmp instruction to help enforce unsigned comparison	Filter Patch
2020-10027	ZephyrOS	Obtain C flag in APSR after the cmp instruction to help enforce unsigned comparison	Filter Patch
2020-10028	ZephyrOS	Perform validation for arguments of the GPIO handlers	Filter Patch
2020-10058	ZephyrOS	Perform validation for arguments of the kscan syscalls	Filter Patch
2020-10059	ZephyrOS	Enable DTLS peer checking for the UpdateHub module	Code Replace Patch
2020-10060	ZephyrOS	Add array length check to avoid reference uninitialized stack memory	Filter Patch
2020-10061	ZephyrOS	Basic block-level prevent referencing uninitialized stack memory in function updatehub_probe	Filter Patch
2020-10062	ZephyrOS	Rewrite packet_length_decode function to correct the looping logic	Code Replace Patch
2020-10063	ZephyrOS	Check for integer overflow during CoAP option parsing	Filter Patch
2020-10064	ZephyrOS	Patch code is too complex and involves many changes	–
2020-10066	ZephyrOS	Add argument nullpointer checks for function hci_cmd_done in Bluetooth HCI core	Filter Patch
2020-10067	ZephyrOS	Add integer overflow checks for is_in_region function	Filter Patch
2020-10068	ZephyrOS	Drop response with no local initiated request and duplicate requests	Filter Patch
2020-10069	ZephyrOS	Add arguments validation for function ull_slave_setup	Filter Patch
2020-10070	ZephyrOS	Add arithmetic overflow and buffer bound checks for function mqtt_read_message_chunk	Filter Patch
2020-10071	ZephyrOS	Further check the length field on publish messages	Filter Patch
2020-10072	ZephyrOS	Patch code is too complex and involves many changes	–
2020-13598	ZephyrOS	Macro definition modification are involved	–
2020-13600	ZephyrOS	Rewrite function eswifi_reset and __parse_scan_res logic to add buffer overflow check	Code Replace Patch
2020-13601	ZephyrOS	Add out-of-bounds read check in the middle of function dns_read	Filter Patch
2020-13602	ZephyrOS	Basic block-level intercept redundant branch for malformed packet in function do_write_op_tlv	Filter Patch
2020-17441	PicoTCP	Validate IPv6 payload length field against actual size for function pico_ipv6_process_in	Filter Patch
2020-17442	PicoTCP	Validate hop-by-hop IPv6 extension header length field for function pico_ipv6_process_hopbyhop	Filter Patch
2020-17443	PicoTCP	Restrict that echo->transport_len is no less than 8 in pico_icmp6_send_echoreply	Filter Patch
2020-17444	PicoTCP	Check possible overflow of header extension length field for pico_ipv6_check_headers_sequence	Filter Patch
2020-17445	PicoTCP	Validate optlen using a loop prior to function pico_ipv6_process_destopt	Filter Patch
2020-24337	PicoTCP	Validate TCP packet option length field before invoking option handler	Filter Patch
2020-24338	PicoTCP	Add bound check on iterator for the while loop in function pico_dns_decompress_name	Filter Patch
2020-24339	PicoTCP	Add out-of-bounds check for iterator of pico_dns_packet in function pico_dns_decompress_name	Filter Patch
2020-17437	uIP	Add validation for Urgent data offset(uip_urglen) in TCP data	Filter Patch
2020-24334	uIP	Add checks for nameptr when processing DNS answers to prevent possible out-of-bound read	Filter Patch
2021-3336	wolfSSL	Exit function DoTls13CertificateVerify on signature without corresponding certificate	Filter Patch
2020-24585	wolfSSL	Add check to reject DTLS application data messages in epoch 0 as out of order	Filter Patch
2020-12457	wolfSSL	Record with eBPF map to prevent multiple ChangeCipherSpecs in a row	Filter Patch + Map
2019-18840	wolfSSL	Need to add too many sanity checks	–
2019-16748	wolfSSL	Add sanity check on length before read for function CheckCertSignature_ex	Filter Patch
2020-24335	Contiki-NG	Rewrite function decode_name to limit pointer bound during domain name parse	Code Replace Patch
2020-24336	Contiki-NG	Validate DNS answer‘s length field before using it for memcpy in function ip64_dns64_4to6	Filter Patch
2020-13987	Contiki	Add out-of-bounds check for upper_layer_len in function upper_layer_chksum	Filter Patch
2020-17439	Contiki	Match DNS reply with outgoing DNS query using eBPF map for function newdata	Filter Patch + Map
2020-25111	Nut/Net	Rewrite function ScanName to validate domain name length and label length	Code Replace Patch

Table 7: CVEs studied in this paper.