MPUsh: Applying Security Hotpatches Instead of MPU Barriers

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

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

Abstract. Due to hardware limitations and stringent timing demands, runtime hotpatching of security vulnerabilities on flash-constrained, hard real-time embedded systems remains a significant challenge. We present MPUsh, a novel Memory Protection Unit (MPU)-based hotpatching approach. MPUsh leverages the MPU to render vulnerable flash regions non-executable. Fault handlers then intercept these violations and redirect execution to RAM-resident patches. Our proof-of-concept prototype, implemented on an ARM Cortex-M4 (NUCLEO-F446RE) processor, activates patches in 15 cycles and redirects execution in 46 cycles. MPUsh outperforms interpreter-based alternatives while supporting arbitrary patch locations without pre-inserted hooks. Furthermore, MPUsh provides more patch slots than approaches that use hardware breakpoints. When evaluated on a safety-critical syringe pump, MPUsh successfully demonstrated real-time capability.

Keywords: Hotpatching · Memory Protection Unit · Real-Time Embedded Systems · Security Patches

1 Introduction

Real-time embedded systems, such as automotive electrical control units (ECUs) and medical devices, require both functional correctness and continuous availability. However, growing connectivity exposes them to zero-day exploits [20, 10, 26]. Traditional firmware updates require reprogramming and rebooting, resulting in unacceptable downtime in hard real-time environments.

Hotpatching on embedded devices involves updating firmware or application code at runtime. The device keeps running, which is crucial for real-time or safety-critical systems, such as medical or industrial controllers, that cannot afford reboots or downtime. On microcontrollers such as ARM Cortex-M, hotpatch frameworks can redirect execution from vulnerable functions to patched versions using hardware features (e.g., flash patch units), while maintaining strict timing guarantees. Research such as HERA [19] shows that such patches can be deployed with microsecond-level latency and low memory overhead.

When hardware hotpatching features such as flash patch units are unavailable, embedded systems can trade a small amount of extra latency for portability by inserting software patch points that redirect execution to a generic patch runtime. This adds an additional branch and context handoff on each patched path, while keeping timing overhead low enough for many real-time workloads and avoiding reliance on vendor-specific debug hardware. RapidPatch [14] follows this model by using an embedded eBPF virtual machine and pre-compiled patch points in the firmware: when execution hits a patch point, it jumps into the eBPF VM, runs the patch bytecode, and then decides whether to continue, redirect, or skip the vulnerable code, enabling generic, cross-device hotpatches with modest latency overhead.

HERA [19] and RapidPatch [14] both assume that firmware executes directly from flash, and their hotpatching mechanisms are built around redirecting control flow for code stored in non-writable program memory. This leaves a gap for devices that use code shadowing, where a bootloader copies code from flash into RAM and the system runs primarily from RAM, because these systems face different security and consistency challenges when hotpatches share the same writable memory space as executable code. Kintsugi [18] addresses this challenge by providing a secure hotpatching framework specifically for code shadowing in real-time embedded systems, applying patches atomically during RTOS context switches, and enforcing strict memory policies to protect both the framework and the patch code, thereby complementing HERA and RapidPatch with a solution tailored to RAM-executing devices.

HERA [19] and RapidPatch [14] demonstrate two important but imperfect ends of the firmware hotpatching spectrum: HERA relies on a minimal pool of hardware breakpoints, which constrains how many vulnerabilities can be patched at once, while RapidPatch exposes many flexible patch points at the cost of significant interpreter overhead on the hot path. Kintsugi, in turn, targets a different class of systems by designing its hotpatching model specifically for code-shadowing devices that execute from RAM, rather than from flash, and thus faces different consistency and isolation challenges.

Building on these insights, we develop *MPUsh*, which follows a complementary direction: it revisits the idea of leveraging hardware breakpoints [19], but instead uses the Memory Protection Unit (MPU) to mark vulnerable code regions as inaccessible, and then reroutes control to patched code running within the exception-handling path. We believe that **MPU**-based **S**ecurity **H**otpatches (MPUsh) can be leveraged as an alternative hotpatching method.

Our contributions are as follows:

- We explore whether the MPU can be leveraged for hard real-time patches.
- We implement a proof-of-concept (POC) MPU-based hotpatching approach called *MPUsh*.
- We evaluate *MPUsh* based on a syringe pump setup.
- We measure patch activation and patch redirection times to compare against state-of-the-art solutions.

2 Technical Background

Since our approach relies on hardware-enforced memory protection and exception handling, we begin with an overview of flash memory limitations on embedded devices, the architecture and semantics of memory protection units, real-time operating systems, and the basics of runtime code detouring.

Flash Memory. Most microcontrollers execute code directly from on-chip flash memory, which typically allows erase and write operations for entire pages (e.g., 4KB). Erasing and programming flash memory involve high latency and significant energy costs, making in-place binary overwriting impractical for fine-grained runtime updates [16].

Memory Protection Units (MPU). An MPU provides region-based access control on many ARM Cortex-M and other microcontrollers. Each MPU region defines a base address, a size (power-of-two aligned), and access attributes (read, write, and execute). An attempt to violate these access restrictions, e.g., retrieving an instruction from an area marked as non-executable, generates a memory management fault exception that can be caught and handled by the developers.

Real-Time Operating System (RTOS). In systems with hard real-time requirements, an RTOS (e.g., FreeRTOS) provides dynamic, prioritized scheduling and interrupt management to ensure task deadlines are met. The latency for exceptions to enter and exit must be carefully limited to avoid extending worst-case response times.

Code Detouring. Hotpatching techniques typically involve redirecting execution from the original code to the patch code via trampolines or patch points. On resource-intensive platforms, this is often achieved through dynamic linking or interpreter frameworks. On microcontrollers, hardware-supported breakpoints or pre-reserved instruction slots are used.

Interrupt Handling. When an interrupt occurs, the processor finishes the current instruction, saves essential context such as the program counter and status registers, and jumps to a dedicated interrupt service routine (ISR). The ISR quickly services the event (for example, reading data from a peripheral or clearing a flag) before restoring the saved context and resuming normal execution.

eBPF (extended Berkeley Packet Filter). is a lightweight, sandboxed bytecode format and virtual machine designed initially to run safe, verifiable programs inside the operating system, such as packet filters or tracing logic. It allows developers to write patch or instrumentation code in C, compile it to eBPF bytecode, and then execute it with strong guarantees about bounded runtime and safe memory access enforced by a verifier. In RapidPatch, eBPF is repurposed as a portable patch language for embedded devices: the framework embeds an eBPF VM in the firmware and uses it to run hotpatch code at predefined patch points, so that a single eBPF patch can fix the same vulnerability across heterogeneous microcontroller platforms that share the same RTOS or library version.

3 Related Work

Before addressing real-time embedded systems, researchers developed various hotpatching approaches for general-purpose platforms, including **Linux**, **Windows**, and **Android**. For a full-fledged Linux environment, extensive research and approaches, such as KSplice [4], LUCOS [6], Katana [22], and others [29, 11, 15], have been proposed.

Many existing hotpatching frameworks are designed for general-purpose operating systems and do not transfer well to small, real-time embedded platforms. In particular, they assume features such as dynamic linking, rich OS services, or hardware virtualization, which are absent on typical microcontrollers with tight memory and timing budgets. As a result, these solutions cannot be applied directly to firmware that is statically linked, runs without an MMU, and must meet strict worst-case execution time guarantees.

KSplice [4] operates at the object-code level and relies on the presence of relocatable kernel modules and dynamic linking to prepare and inject patches at runtime. In contrast, firmware for microcontrollers is usually built as a single, statically linked image, leaving no notion of loadable objects or symbol relocation at runtime, which effectively prevents the adoption of KSplice's update model. Katana [22] faces a similar limitation: it depends on dynamic linking and runtime symbol resolution to weave patches into the running system, assumptions that do not hold for bare-metal or minimalist RTOS-based deployments.

LUCOS [6] requires hardware and software support for virtualization to host a live-updated operating system alongside the running instance. This design is incompatible with most resource-constrained embedded devices, which lack virtualization extensions, have limited memory, and cannot afford the overhead of running multiple OS instances. InstaGuard [8] introduces GuardRules, a powerful mechanism to deploy runtime security policies, but the associated monitoring and enforcement framework introduces non-trivial CPU and memory overhead. Such continuous policy checks and instrumentation are difficult to reconcile with hard real-time constraints and tight resource envelopes typical of safety-critical embedded controllers. Likewise, KARMA [9] leverages filters inside the Android kernel to sanitize inputs and mitigate attacks. Its design presupposes a full-featured Linux/Android software stack. In contrast, many real-time embedded systems run directly on bare metal or on lightweight RTOSes such as FreeRTOS [12] or Zephyr [31], where there is no monolithic kernel to extend in this way and where additional filtering layers must be carefully bounded to preserve deterministic timing behavior.

A parallel line of research focuses not on the mechanics of hotpatching deployment, but on the upstream process of generating hotpatches efficiently and reliably, often through automated analysis of vulnerabilities and diffs between vulnerable and patched binaries. These techniques aim to reduce the manual effort required from developers while producing patches that are functionally correct and deployable with minimal disruption.

Vulmet [30] targets Android devices and automates hotpatch creation by analyzing official security updates, extracting vulnerability fixes, and synthesiz-

ing lightweight patches that can be pushed over-the-air without full firmware replacement. While effective for mobile ecosystems with dynamic app lifecycles and abundant resources, Vulmet assumes a rich OS environment for patch verification and execution, which limits its applicability to bare-metal or RTOS-based embedded systems where such abstractions are unavailable. In contrast, AutoPatch [24] addresses embedded systems more directly by leveraging LLVM [17] to generate architecture-agnostic hot patches: it performs static analysis on official patches to derive equivalent replacements, then emits portable IR code that can be compiled for diverse processors (e.g., ARM Cortex-M or RISC-V) without manual porting. This approach excels at producing small, self-contained patches suitable for resource-limited devices. However, it still requires integration with a compatible runtime trigger mechanism, which our work complements by enabling hardware-assisted redirection.

Recent efforts have also explored hotpatching in specialized domains, such as event-driven real-time applications in industrial settings. ICSPatch [21] uses data dependence graphs to localize vulnerabilities in control-logic binaries and applies non-intrusive hot patches without source access, making it promising for legacy systems. Similarly, RLPatch [32] emphasizes reliability in real-time patching for event-triggered controllers, incorporating rollback mechanisms to handle partial failures. However, both target industrial control systems (ICS), which are often proprietary, closed-source ecosystems with vendor-specific protocols and hardware. These solutions prioritize compatibility with SCADA/PLC environments over general-purpose microcontrollers, placing them outside the scope of our firmware-focused framework, which emphasizes open, off-the-shelf embedded hardware.

Binary instrumentation—inserting trampolines or hooks to redirect control flow at runtime—remains a foundational technique for dynamic analysis and patching, but it poses challenges for our target domain. Traditional tools like PISTON [25] and other binary rewriting frameworks (e.g., based on Valgrind or DynamoRIO) assume mutable memory and fine-grained code modifications, often requiring disassembly, relocation, and rewriting of small code snippets. In flash-based embedded systems, however, code executes from read-only memory (ROM), where even minor changes necessitate erasing and reprogramming large sectors (typically 4–64 KiB blocks), incurring significant wear, time, and power costs. Like HERA [19], our approach avoids such invasive instrumentation by relying on hardware triggers (e.g., MPU faults or breakpoints) to intercept execution without altering the original binary, preserving the immutability of ROM while enabling seamless redirection to RAM-resident patches.

Among frameworks tailored to flash-constrained embedded devices, RapidPatch [14] stands out as the most recent and relevant: it enables cross-architecture hotpatching by compiling patches into eBPF bytecode, which is interpreted at runtime for portability across ISAs such as ARM and x86. This design allows deployment without architecture-specific recompilation, but it introduces substantial overhead from the eBPF verifier, just-in-time compilation, and a safe-execution sandbox—potentially hundreds of cycles per patch invocation—which

is prohibitive for hard real-time systems with microsecond deadlines. Moreover, RapidPatch requires the insertion of potential patch entry points (e.g., no-op hooks) at compile time into firmware, which increases binary size and complicates certification for safety-critical applications. Our *MPUsh* framework, akin to HERA [19], avoids such overhead by leveraging native hardware support (e.g., MPU regions) for low-latency redirection, ensuring that patches can be applied to arbitrary locations without pre-inserted hooks or interpretive layers. This hardware-centric design is essential for resource-starved, timing-deterministic devices, where even modest runtime or memory penalties can cascade into missed deadlines or system instability.

4 Problem Statement

Modern embedded systems, particularly those in real-time and safety-critical applications, often operate under strict timing constraints where downtime for patching is unacceptable. Applying critical security updates typically requires rebooting the device, disrupting functionality, and violating real-time guarantees. Hotpatching, i.e., applying patches at runtime without service interruption, has emerged as a solution. However, existing approaches face limitations in embedded environments due to hardware constraints and limitations on flash memory. Current hotpatching techniques for embedded devices can be broadly categorized into five groups.

Hardware-Assisted Hotpatching Techniques. These techniques use a dedicated hardware unit to perform trampoline insertion and redirect the program flow towards the patch. In HERA [19], Niesler et al. used the debugging unit typically implemented in ARM devices (called *FPBU*) to insert a trampoline and redirect control flow to patches. While efficient, this approach is limited to the availability of suitable hardware.

Dynamic Patching Through Pre-Compiled Patch Points. Current state-of-the-art solutions, such as RapidPatch [14], use patch points prepared during compile time and an interpreted language, namely the extended Berkeley Packet Filter (eBPF). Since the eBPF patch requires interpretation, this approach incurs a performance cost. Furthermore, the required pre-compiled patch points reduce flexibility in handling arbitrary patches.

Code-Shadowing & Binary Rewriting. Some approaches target code-shadowing devices (i.e., code is executed from RAM). With this approach, the entire code is copied from flash memory (ROM) into RAM during startup and executed from RAM. Approaches targeting such devices have different possibilities, e.g., binary rewriting. Flash memory is usually only page-writable (i.e., large chunks need to be erased and rewritten) and very energy- and time-consuming [16].

Compartmentalization-Based Patching. For large applications where resource constraints are not a significant concern, exchanging entire software components is a viable option. The software can be compartmentalized into smaller yet still substantial components during development, and those compo-

nents can be exchanged at runtime. Non-embedded server-side hot patches include functional updates that require state transfer functionality. POLUS [7] is a good example of an early hotpatch mechanism for large software applications. In contrast, hot patches for embedded devices typically include only security patches, which are usually small and featureless [1]. Thus, compartmentalization-based patching with complex state transfer is rarely seen in the embedded realm.

A/B hotpatching. uses dual firmware partitions—an active one running the current code and a backup for updates—with hotswap capability to switch seamlessly without complete reboots in some designs (switching without reboot may require expensive and complex hardware support and may not be feasible in some applications). The backup partition receives the patch while the active one continues execution, and a bootloader or runtime manager then atomically swaps roles, often marking the new active partition with a flag in persistent storage, such as flash. While memory-intensive due to storing two full images, this approach suits embedded devices with sufficient storage, providing rollback safety but incurring brief downtime during the swap, unlike true in-memory hotpatching.

Challenges of embedded real-time hotpatching. While hotpatching itself is challenging due to limited resources on embedded devices, the timing constraints of real-time systems make developing suitable patching mechanisms even more difficult, especially for hard real-time systems where timing must be met at all costs. In general, real-time constraints are categorized into three types: *(1) hard, (2) firm, and (3) soft*. Hard real-time systems must meet all deadlines without exception, as failure could lead to catastrophic consequences. Safety-critical applications, such as automotive airbag control systems, fall into this category. In contrast, firm real-time systems tolerate occasional deadline misses, though frequent misses would render the system ineffective. An example is audio playback, where skipping a few samples may go unnoticed, but consistent interruptions would degrade performance. Systems that do not fall into either the hard or firm categories are considered soft real-time, in which the value of information gradually diminishes rather than becoming immediately irrelevant. A home heating system exemplifies this, as it can operate effectively with intermittent temperature updates while still maintaining functionality [27].

Among hotpatching approaches specifically targeting embedded hard-real-time applications, HERA [19] is the most performant in terms of patch times, owing to the use of the *FPBU* debugging unit on ARM devices. However, the *FPBU* is limited to a fixed number of breakpoints, e.g., up to 6 on the Cortex M4 [2]. Furthermore, not all devices feature an integrated debugging unit capable of supporting real-time patches.

This raises a critical question: ***Can another commonly available hardware feature achieve comparable or at least sufficient switch time to support hard real-time patches?***

In this paper, we explore the use of the *Memory Protection Unit (MPU)*, which is a standard feature in many microcontrollers, as an alternative mechanism for runtime patching. By leveraging memory protection faults to intercept

and redirect execution, we investigate whether the *MPU* can provide a viable alternative to debugging units for patching flash-based embedded hard real-time systems.

5 Threat Model

We assume a real-time system executing mixed-criticality tasks that range from safety-critical to low-priority. Further, we assume sufficient computational resources and memory space to ensure deadline-compliant execution under nominal and peak loads. The system enforces privilege separation. That is, safety-critical components, including the RTOS kernel, fault handlers, MPU configuration logic, and patch management, execute in privileged mode. In contrast, application tasks run in unprivileged mode. A secure updater service, adapted from mechanisms like FreeRTOS AWS IoT OTA [13], handles authenticated patch downloads to a dedicated RAM region by means of low-priority background tasks that do not interfere with high-criticality deadlines, followed by hardware-assisted atomic activation (e.g., via context switches or MPU reconfiguration) to apply patches without disrupting ongoing execution.

Our attacker model considers a remote adversary capable of exploiting memory corruption vulnerabilities (buffer overflows, use-after-free errors, integer overflows) in unprivileged RTOS application tasks, libraries, or protocol handlers, triggered by network inputs or malformed peripheral data. Attackers lack physical access and cannot compromise the cryptographic update mechanism (e.g., signature verification via hardware keys). We assume that the attacker remains confined to unprivileged execution and cannot escalate to privileged mode. This assumption is enforced by RTOS privilege separation and MPU configuration, which restrict unprivileged tasks from accessing privileged memory regions, including MPU registers, fault handler code, and the patch memory area. Thus, the attacker may attempt to cause data leaks or denial-of-service within the exploited task's context, but cannot directly tamper with the hotpatching framework.

This model aligns with practical embedded security challenges in mixed-criticality systems, where runtime-patchable defects from programming errors or third-party libraries must be addressed rapidly without violating real-time deadlines, safety isolation between criticality levels, or security boundaries enforced by the RTOS and MPU.

Furthermore, we assume that the applied hotpatches are small and featureless [1]. We assume that the attacker lacks the necessary credentials to directly upload maliciously crafted hotpatches to the device or alter its firmware. In accordance with prior research, such as Kintsugi [18], our threat model does not encompass verifying the functionality, safety, or security of the hotpatch code, as these aspects are beyond the scope of *MPUsh*.

Our threat model excludes physical attacks, side-channel attacks, supply-chain compromises, and privilege escalation beyond the unprivileged task boundary. In other words, we focus on post-deployment remediation of application-level code vulnerabilities.

6 Concept

Our approach leverages the Memory Protection Unit (MPU) to enable runtime patching by selectively blocking access to specific code regions in flash memory. When an instruction is fetched from a protected memory region, the MPU raises a hard fault or memory management fault. In the corresponding fault handler, *MPUsh* examines the faulting program counter (PC) to determine if it matches a registered patch location. If it does, execution is redirected to the patch code in a dedicated, executable RAM section. After the patch completes, execution returns to a safe address outside the protected region, and if necessary, the handler reconfigures the MPU to ensure correct subsequent operation. Figure 1 visualizes these steps.

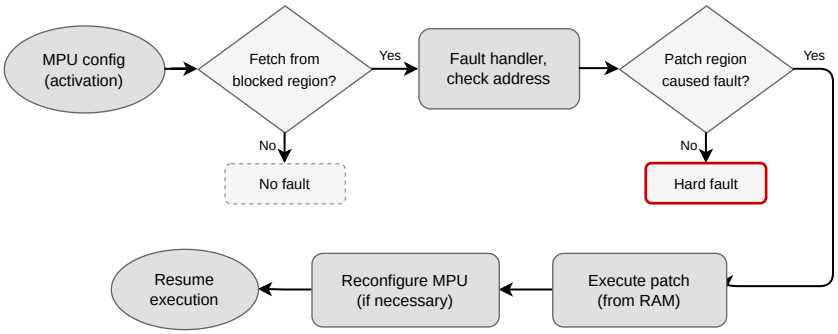

Fig. 1. Patch Application Workflow

The high-level system architecture of *MPUsh* follows a design similar to HERA [19], which applies patches dynamically without requiring system downtime. The system operates on a real-time operating system (RTOS), such as FreeRTOS, with multiple tasks at varying priorities. A dedicated low-priority (or idle) updater task manages patch preparation and deployment: it receives patch payloads, validates them, and stores them in a reserved region of executable RAM. Once a patch is ready, *MPUsh* configures the MPU to mark the vulnerable flash section as non-executable, thereby activating the patch. Figure 2 depicts the overall process.

The MPU's region-based access control allows patches to be enabled and disabled by dynamically remapping protection rules. Our method maintains low runtime overhead: patches are triggered only when the original protected code is executed, and the MPU's hardware checks add minimal performance overhead. The Implementation section details patch registration, dynamic MPU reconfiguration, and fault handling.

Comparison to Established Concepts Similar to HERA [19], this approach relies on dedicated hardware support to trigger a switchover to patched code, but it

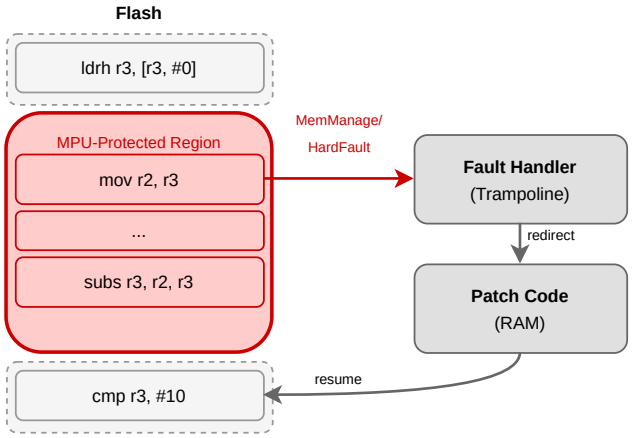

Fig. 2. High-Level Overview

uses the MPU for this. In HERA, debugging facilities—specifically, hardware breakpoints—serve as the trigger mechanism for patch activation. While breakpoints are attractive because this hardware is typically reserved for debugging and remains unused during normal deployment, an MPU-based design must account for any existing MPU configuration. Like the finite number of hardware breakpoints, the MPU offers only a limited set of regions that can be protected, and, unlike in HERA, these regions may already be occupied by other protection rules, requiring the hotpatching mechanism to integrate with the current memory protection setup. In this work, only unused MPU region slots are employed for patch-related protection. However, when MPU subregions are taken into account, the theoretical number of available hotpatch trigger slots increases significantly. Instead of being limited to the 8 coarse-grained MPU regions on the Cortex-M4, each region can be subdivided into 8 independently configurable subregions, yielding up to 64 distinct protection zones that can, in principle, each host a separate hotpatch entry point. HERA [19] is constrained by the underlying Flash Patch and Breakpoint Unit, which typically provides up to six hardware breakpoints on Cortex-M3/M4-class devices [19]. Hence, our approach provides a key advantage over related work: when MPU subregions are used, an MPU-based design can offer substantially more hotpatch slots (i.e., concurrent patches). At the same time, MPUs, like hardware breakpoints, are widely available across many microcontroller families and architectures, making this approach broadly portable beyond a single vendor or SoC line.

Unlike RapidPatch [14], which requires pre-defined patch entry points in the firmware and allocates additional resources for patch activation, both HERA [19] and the *MPUsh* method integrate into existing binaries without requiring such

pre-compiled hooks. This is achieved by redirecting control flow using hardware features. Both hardware breakpoints and MPUs can be configured to intercept execution at any address and reroute it to patch routines, enabling flexible deployment of hot patches.

Kintsugi [18] also supports patching at arbitrary locations and is tightly integrated with a real-time operating system for coordination. However, Kintsugi is designed for code shadowing environments, where the full firmware image is run from RAM. As a result, Kintsugi does not face the ROM-specific constraints and complexity present in approaches like HERA or *MPUsh*, which must account for the immutability and limitations of ROM-based firmware.

Integration into Existing MPU Configurations A practical integration strategy is to treat *MPUsh* as an overlay on top of the existing MPU layout, consuming only spare region capacity while respecting current protection rules. That is, the hotpatching runtime would inspect the static MPU configuration. Next, it records which regions and subregions are already utilized for security or isolation policies. *MPUsh* should only allocate from the remaining region slots and, where supported, from free subregions within those regions. For each patch, the framework would compute an MPU region (or subregion) configuration that minimally intersects the vulnerable code interval, program this into an unused slot, and leave all pre-existing regions untouched, thereby maintaining the application's original protection. If no free regions are available, a more advanced strategy can be employed. In this strategy, existing protection rules are coalesced. Concretely, an existing region is expanded to cover both its original address range and the vulnerable code. Access permissions are then refined using subregion masks. This refinement allows no-execute subregions that trigger the hotpatch to coexist with the original read, write, and execute subregions. As a result, the hotpatching mechanism is integrated without violating previously established isolation guarantees.

7 Implementation

We implement a proof-of-concept for *MPUsh* on the NUCLEO-F446RE (also used by HERA [19]), clocked at 42 MHz.

Patches are staged in system RAM before activation, ensuring that all code and metadata are present and verified prior to redirecting execution. Similar to HERA [19], patch payloads are derived by computing the binary delta between the original vulnerable firmware image and the updated, fixed version, thereby keeping transfer sizes small and preserving compatibility with existing binaries.

In contrast to HERA's breakpoint-driven redirection, the MPU-based design requires additional metadata for each patch instance to support runtime fault-driven control-flow rerouting. This metadata explicitly includes the target region, specifying the MPU-protected flash interval that contains the vulnerable instructions, which must be marked non-executable to trigger the patch. *MPUsh* requires:

1. **Patch Address:** RAM location of the patched code (currently specified manually, but it is possible to automate this process).
2. **Target Region:** the MPU-protected flash region that contains the vulnerable code.
3. **Return Address:** the resume point after patch completion.

To host this information and the corresponding code, a dedicated 4 kB RAM section called *PATCH* is reserved at link time for patch payloads and their control structures. This statically carved-out *PATCH* region guarantees that sufficient contiguous memory is available for patch deployment, avoids interference with regular heap/stack usage, and reduces the risk of fragmentation-related allocation failures during long-running system operation. By fixing both the size and the location of *PATCH*, the runtime can use simple pointer arithmetic and constant-time checks when installing, activating, or rolling back patches.

The 4 kB PATCH section can host only a limited number of hotpatches, determined by the combined size of their metadata and code. Each patch entry consists of a small metadata header (patch entry address, MPU region configuration: region index, aligned base, size, attributes), which in a compact layout should be roughly 20 bytes, plus the patch code itself. The patch implements the security fix and replicates adjacent non-vulnerable instructions to satisfy MPU alignment (minimum 32 bytes regions). In practice, patch sizes vary substantially: in the hotpatch set used by RapidPatch [14], AutoPatch [24], and Kintsugi [18], individual patches range from about 48 bytes to roughly 944 bytes of code, with 20 bytes of metadata, yielding effective per-patch footprints from approximately 68 bytes to about 964 bytes. Under these assumptions, a 4 kB PATCH region could hold about 60 small patches (68 bytes each). Considering large patches (964 bytes each), the patch region would accommodate only around 4 patches. The NUCLEO-F446RE's Cortex-M4 provides 8 MPU regions [28], each of which can be subdivided into 8 subregions. This yields up to 64 hotpatch slots, i.e., 64 concurrent patches. Yet, application code will usually occupy some of the MPU's regions. Thus, fewer slots will remain available for hotpatching in realistic deployments.

Handling the MPU and Patch Activation. The MPU is the central hardware element that enables runtime patch activation in this design. A lightweight helper library encapsulates the low-level register operations. It exposes a small API: `mpu_disable()` and `mpu_enable()` control the global MPU state, while `mpu_region_enable()` and `mpu_region_disable()` manage individual regions, and `mpu_patch_activate()` configures protection for vulnerable code sections. In addition, the library defines common control and attribute bit masks, reducing duplication and helping keep MPU configuration consistent across the firmware.

The `mpu_patch_activate()` primitive programs a dedicated MPU region for use by the hotpatching mechanism. It takes three parameters: the region index, the region size (minimum 32 bytes), and the aligned base address of the target memory range. Internally, the function first selects the region via the *RNR* register, then writes the aligned base address to *RBAR*, and finally configures

the region attributes in *RASR*, including size, access permissions, execute-never settings, and optional subregion masks. By centralizing this logic, the patching framework can reliably install, update, or remove protection for specific flash regions at runtime while respecting MPU alignment constraints and ensuring that region activation and deactivation follow a well-defined sequence.

Fault Handling and Patch Application. When a fault, such as a HardFault or Memory Management fault, occurs on ARM Cortex-M devices, the processor automatically pushes the current register state onto the stack. This fault context includes the values of critical registers such as `PC` and `LR`, as well as general-purpose registers (`R0`–`R3`, `R12`), enabling the handler to precisely reconstruct the interrupted execution state.

The fault handler workflow proceeds as follows:

1. **Cause Identification:** The handler determines the underlying fault reason, such as a memory access violation, by examining fault status registers and other metadata.
2. **Stack Frame Inspection:** For patch activation, the handler accesses the fault stack frame using a type such as `HardFaultStackFrame`, which provides direct access to the saved register values at the time of the fault.
3. **Register Restoration:** If required, specific registers are restored or modified to prepare execution for the patch code.
4. **Patch Redirection:** The stacked program counter is set to the patch code in RAM, thus rerouting execution to the patch handler.
5. **Post-Patch Handling:** Upon completion, the handler chooses the correct resume address—either the original execution point, a designated safe location, or another site, depending on the patch logic. The handler may also deactivate or reconfigure the relevant MPU region if access to the previously protected code is still needed.

If a patch does not require access to MPU-blocked code regions, the handler can skip region deactivation and optionally duplicate the necessary instructions within the patch stub for self-containment. This approach ensures safe, flexible patch application while maintaining precise control over the system state at the time of the fault.

7.1 Implementation Challenges

Register Restoration. Since patches often extend existing functions, they must preserve the original function's register state. Register loss—especially during HardFaults—necessitates careful restoration. For ARM Cortex-M devices, the stack frame of the fault handler contains the state of the ARM Cortex-M registers at the time that the fault occurred [5]. At a minimum, critical registers must be recovered; ideally, all registers should be restored to maintain consistency and integrity.

Interrupt Handling. Redirecting execution via interrupts can cause the system to erroneously remain flagged as being in interrupt mode, even after

exiting the HardFault handler. This occurs if the handler jumps to an arbitrary address (e.g., by directly manipulating the *pc* register) instead of using a branch instruction (Link Register *lr*). To avoid this, the *lr* register must be explicitly overwritten with the target address before the jump.

MPU Alignment Constraints. The base address alignment (relative to region size) can enforce protection of neighboring instructions if the aligned base does not match the entry point exactly. Therefore, patch authors must include these newly blocked instructions in the patch.

Patch Bytecode Generation. Direct compilation of high-level C code is unreliable without a compiler that is aware of the calling conventions and stack layout of the live firmware. A more robust approach is to create or verify patches at the assembler level. Tools such as Cutter [23] or a plain assembler can assist, but validation is essential to ensure correctness and comply with size restrictions.

8 Evaluation

For a direct comparison with HERA [19], our evaluation uses the same hardware setup and case study: a NUCLEO-F446RE development board is connected to a stepper motor, a driver circuit, an LCD screen, and a 3D-printed syringe assembly. The system is designed as an automated syringe pump.

The evaluation firmware intentionally contains a safety-critical vulnerability to showcase hotpatching capabilities. Specifically, the system's command line interface (CLI) allows users to send commands that control the syringe motor's direction (push/pull). However, input validation is lacking, resulting in a buffer overflow vulnerability. This flaw enables attackers to corrupt the call stack and manipulate return addresses, thus facilitating code-reuse (ROP) attacks. An adversary could exploit this weakness to hijack the pump—redirecting execution to arbitrary code, such as logic that forces an overdose by injecting excessive medication. We implement and test a hotpatch using *MPUsh*. The following fix would add the missing bounds check:

Listing 1.1. C-Code of the patch

```
if ((message[1] - message[0]) > 10)
 { return; }
```

The syringe-pump case study demonstrates that the MPU-based hotpatch mechanism operates correctly in a realistic embedded control application and can meet stringent yet predictable real-time constraints. To quantify the timing overhead introduced by hotpatching, the firmware uses the on-chip Data Watchpoint and Trace (DWT) unit on the NUCLEO-F446RE, configured to run at 42 MHz, to collect precise cycle-level measurements.

We focus on two latency components that are critical for real-time behavior:

1. the *patch activation* latency, which captures the time required to configure the MPU, install the protection rules for the vulnerable code region, and deploy the corresponding patch metadata,

2. the *execution redirection* latency, which measures the end-to-end delay from the first instruction fetch in the protected flash region to the start of the patch code in RAM, including fault entry, stack-frame processing, and return path adjustment.

The reported cycle counts inherently include minor measurement artifacts, such as the cost of reading the DWT counter and effects from the Cortex-M4 three-stage pipeline[1], so the values should be interpreted as representative reference numbers rather than absolute lower bounds.

Activating the hotpatch requires writing to three registers: MPU_RNR, MPU_RBAR, and MPU_RASR. According to the Cortex-M4 User Guide [3], these registers can be modified independently. This allows the region number and base address to be configured in MPU_RNR and MPU_RBAR, respectively, without affecting MPU_RASR, since the region only becomes active after it is configured in this register. Consequently, the hotpatch activation process consists of three distinct steps that can be executed separately. In the final step—the activation of the region—the first two registers must already hold valid values. The patch activation, which involves setting the MPU_RASR register (a critical step), completes in 15 cycles.

Table 1. Performance Measurements on the NUCLEO-F446RE

	MPUsh	HERA
Patch activation	28 cycles	10 cycles
Patch activation (critical step)	15 cycles	10 cycles
Flow redirection	46 (136) cycles	8.2 cycles

Patch activation latency is measured by sampling the cycle counter immediately before and after the MPU configuration sequence that enables a hotpatch. In this setup, configuring the MPU region for a patch requires approximately 28 clock cycles in total.

In addition to the activation cost, an important timing metric is the delay introduced when diverting control flow from the normal execution path in flash to the patch code in RAM. This redirection sequence encompasses entry into the HardFault handler, inspection of the fault context, and the jump to the patch routine. It completes in about 46 CPU cycles. Depending on the chosen patching strategy, execution can then continue in a dedicated post-patch function that performs extra checks, such as validating the return address and confirming the MPU region configuration. This optional validation phase (post-patch) adds roughly 91 cycles but can run at a lower priority, allowing higher-priority RTOS tasks to preempt it. Overall, the full redirection path—from fault trigger to return from the validation logic—incurs a latency of approximately 136 cycles.

[1] Due to pipelining, multiple instructions can overlap in execution, so individual operations may not map one-to-one to clock cycles.

9 Security Considerations

The proposed hotpatching mechanism increases system flexibility but also expands the attack surface, so the patch framework itself must be treated as security-critical. Patch payloads, metadata, and control-flow redirection mechanisms can all be abused by an adversary if not properly authenticated and isolated. In particular, any component that can alter MPU configuration, register patch locations, or modify return addresses must be strictly protected and kept as small and verifiable as possible.

First, patch distribution and storage require end-to-end guarantees of integrity and authenticity. A trusted authority should sign patches, verify them on the device before installation, and store them only in memory regions protected against unauthorized modification. The updater task must validate versioning, apply strict bounds checks on patch metadata (addresses, sizes, target regions), and reject malformed or conflicting updates to prevent arbitrary code injection masquerading as a legitimate patch.

Second, the redirection path—from MPU fault to RAM-resident patch and back—must preserve control-flow integrity. The fault handler should only redirect execution to registered patch entry points, enforce that return addresses lie within valid, executable regions, and prevent arbitrary jumps into the middle of functions or data segments. Additional invariants, such as non-executable data (where supported) and MPU rules that isolate PATCH memory from untrusted code, help ensure that an attacker cannot repurpose the patch facility as a generic ROP or code-reuse gadget. Since this hotpatching approach, similar to HERA [19], relies on a trampoline mechanism, it is essential that the hotpatch be tailored to the currently deployed firmware and preserve the application's intended control flow.

Third, the framework must consider denial-of-service and priority inversion risks in real-time environments. An attacker who can repeatedly trigger MPU faults or long-running patches could starve high-priority tasks or violate timing constraints. To mitigate this, patch handlers should be time-bounded, avoid unbounded loops or blocking operations, and be scheduled with carefully chosen priorities that respect existing RTOS policies. Monitoring mechanisms (e.g., watchdogs or counters) can detect abnormal fault rates or repeated patch activations, enabling the system to enter a safe state or disable suspect patches.

Furthermore, in the current proof-of-concept, the patch is executed in the fault handler's privileged domain. We acknowledge that a vulnerability in the patch code itself could therefore have severe consequences, as it would execute with full system privileges. In accordance with our Threat Model (Section 5), we assume that patches do not introduce new vulnerabilities (i.e., regression bugs). If another memory fault occurs within the fault handler, this automatically triggers a *hard fault* of the system. Nevertheless, to reduce the impact of potential patch defects, future implementations could execute the patch code in unprivileged thread mode. However, any vulnerable patch, regardless of privilege mode, would still damage the system. The unprivileged mode can only help to contain this damage. On ARM Cortex-M, the change to unprivileged mode

can be achieved by leveraging the standard exception-return mechanism: when a HardFault or MemManage fault occurs, the processor stores the interrupted context (including the program counter) on the stack. The handler can then modify this stacked PC to point either to a RAM-resident patch or to a safe resume address past the vulnerable code. By ensuring that the link register holds a valid EXC_RETURN value denoting a return to unprivileged mode, a subsequent *bx lr* causes the core to pop the modified frame and continue execution outside the handler. This approach, however, introduces additional overhead due to the required stack manipulation and mode transition, representing a trade-off between security isolation and redirection latency.

Finally, the interaction with other security mechanisms, such as secure boot and firmware update frameworks, must be well defined. The boot chain should verify both the base firmware and the patching runtime, ensuring that no unauthorized modifications to the MPU helper library, fault handlers, or PATCH memory layout occur before system startup. Combined with a clear rollback strategy for faulty or compromised patches, these measures help ensure that hotpatching improves resilience to vulnerabilities without becoming a new, high-impact attack vector.

10 Conclusion

This paper presents an MPU-based hotpatching method for real-time embedded systems. By leveraging the Memory Protection Unit to block vulnerable instructions, our approach triggers a HardFault upon execution attempts, redirecting control flow through the HardFault handler to preloaded patch code in RAM. This mechanism effectively modifies program behavior at runtime, enabling patching without system downtime.

We validated our approach by implementing a security patch for a syringe pump system that contained an intentional vulnerability. Performance measurements revealed efficient patch activation (15 cycles) and acceptable redirection overhead (136 cycles, 46 without optional validation steps), confirming the method's feasibility for real-time environments.

While the results indicate the MPU's potential as a hotpatching mechanism, we acknowledge several limitations that must be addressed in order to develop a complete framework. These include practical deployment challenges and system-specific constraints that may affect applicability. Our evaluation suggests that, despite these considerations, MPU-assisted hotpatching is a viable solution for maintaining real-time system security through runtime updates.

Acknowledgments. This work was partially funded by Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) as part of Priority Programme RAINCOAT II (440059533) and SFB 1119 – 236615297, project S2.

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

References

1. Altekar, G., Bagrak, I., Burstein, P., Schultz, A.: OPUS: online patches and updates for security. In: USENIX Security Symposium. USENIX Association (2005)
2. ARM: Cortex-M4 Processor Technical Reference Manual. ARM (2015), https://static.docs.arm.com/100166/0001/arm_cortexm4_processor_trm_100166_0001_00_en.pdf
3. ARM: Cortex-m4 devices generic user guide (2024), https://developer.arm.com/documentation/dui0553/a/
4. Arnold, J., Kaashoek, M.F.: Ksplice: Automatic rebootless kernel updates. In: Proceedings of the 4th ACM European conference on Computer systems. pp. 187–198 (2009)
5. AWS open source (Amazon): Debugging hard fault & other exceptions on arm cortex-m3 and arm cortex-m4 microcontrollers (2026), https://freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/Others/Debugging-Hard-Faults-On-Cortex-M-Microcontrollers
6. Chen, H., Chen, R., Zhang, F., Zang, B., Yew, P.C.: Live updating operating systems using virtualization. In: Proceedings of the 2nd international conference on Virtual execution environments. pp. 35–44 (2006)
7. Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.C.: POLUS: A powerful live updating system. In: International Conference on Software Engineering (ICSE). IEEE (2007). https://doi.org/10.1109/ICSE.2007.65
8. Chen, Y., Li, Y., Lu, L., Lin, Y.H., Vijayakumar, H., Wang, Z., Ou, X.: Instaguard: Instantly deployable hot-patches for vulnerable system programs on android. In: 2018 Network and Distributed System Security Symposium (NDSS'18) (2018)
9. Chen, Y., Zhang, Y., Wang, Z., Xia, L., Bao, C., Wei, T.: Adaptive android kernel live patching. In: 26th USENIX Security Symposium (USENIX Security 17). pp. 1253–1270 (2017)
10. Evans, S.: Iot malware attacks jump 400% since 2022, report (2023), https://www.iotworldtoday.com/security/iot-malware-attacks-jump-400-since-2022-report
11. Fischmeister, S., Winkler, K.: Non-blocking deterministic replacement of functionality, timing, and data-flow for hard real-time systems at runtime. In: 17th Euromicro Conference on Real-Time Systems (ECRTS'05). pp. 106–114. IEEE (2005)
12. FreeRTOS: Freertos (2025), https://freertos.org
13. FreeRTOS: Over the air (ota) updates (2025), https://www.freertos.org/ota/index.html
14. He, Y., Zou, Z., Sun, K., Liu, Z., Xu, K., Wang, Q., Shen, C., Wang, Z., Li, Q.: Rapidpatch: Firmware hotpatching for real-time embedded devices. In: 31st USENIX Security Symposium (USENIX Security 22). pp. 2225–2242. USENIX Association (2022), https://www.usenix.org/conference/usenixsecurity22/presentation/he-yi
15. Jeong, H., Baik, J., Kang, K.: Functional level hot-patching platform for executable and linkable format binaries. In: 2017 IEEE International Conference on Systems, Man, and Cybernetics (SMC). pp. 489–494. IEEE (2017)
16. Kwon, J., Cho, J., Park, D.: 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). https://doi.org/10.1109/ISPACS48206.2019.8986373

17. LLVM Project: The llvm compiler infrastructure project (2025), https://llvm.org/
18. Mackensen, P., Niesler, C., Blanco, R., Davi, L., Moonsamy, V.: Kintsugi: Secure hotpatching for code-shadowing real-time embedded systems. In: 34th USENIX Security Symposium (USENIX Security 25). pp. 1847–1866 (2025)
19. Niesler, C., Surminski, S., Davi, L.: HERA: hotpatching of embedded real-time applications. In: 28th Annual Network and Distributed System Security Symposium (NDSS). The Internet Society (2021), https://www.ndss-symposium.org/ndss-paper/hera-hotpatching-of-embedded-real-time-applications/
20. Petrosyan, A.: Monthly number of iot attacks global 2022 (2024), https://www.statista.com/statistics/1322216/worldwide-internet-of-things-attacks/
21. Rajput, P.H.N., Doumanidis, C., Maniatakos, M.: ICSPatch: Automated vulnerability localization and non-intrusive hotpatching in industrial control systems using data dependence graphs. In: 32nd USENIX Security Symposium (USENIX Security 23). pp. 6861–6876 (2023)
22. Ramaswamy, A., Bratus, S., Smith, S.W., Locasto, M.E.: Katana: A hot patching framework for elf executables. In: 2010 International Conference on Availability, Reliability and Security. pp. 507–512. IEEE (2010)
23. Rizin Team: Cutter – a powerful multi-platform reverse engineering tool (2026), https://cutter.re/
24. Salehi, M., Pattabiraman, K.: Autopatch: Automated generation of hotpatches for real-time embedded devices. arXiv preprint arXiv:2408.15372 (2024)
25. Salls, C., Shoshitaishvili, Y., Stephens, N., Kruegel, C., Vigna, G.: Piston: Uncooperative remote runtime patching. In: Proceedings of the 33rd Annual Computer Security Applications Conference. pp. 141–153 (2017)
26. Schiller, E., Aidoo, A., Fuhrer, J., Stahl, J., Ziörjen, M., Stiller, B.: Landscape of iot security. Computer Science Review **44**, 100467 (2022)
27. Shin, K.G., Ramanathan, P.: Real-time computing: A new discipline of computer science and engineering. In: Proceedings of IEEE, Special Issue on Real-Time Systems. IEEE (1994)
28. STMicroelectronics: STM32F446xC/E Datasheet (2015), https://www.st.com/resource/en/datasheet/stm32f446mc.pdf
29. Wahler, M., Richter, S., Kumar, S., Oriol, M.: Non-disruptive large-scale component updates for real-time controllers. In: 2011 IEEE 27th International Conference on Data Engineering Workshops. pp. 174–178. IEEE (2011)
30. Xu, Z., Zhang, Y., Zheng, L., Xia, L., Bao, C., Wang, Z., Liu, Y.: Automatic hot patch generation for android kernels. In: 29th USENIX Security Symposium (USENIX Security 20). pp. 2397–2414 (2020)
31. Zephyr Project: Zephyr project – a proven rtos ecosystem (2025), https://zephyrproject.org/
32. Zhou, M., Wang, H., Li, K., Zhu, H., Sun, L.: Save the bruised striver: A reliable live patching framework for protecting real-world plcs. In: Proceedings of the Nineteenth European Conference on Computer Systems. pp. 1192–1207 (2024)