Artifact Evaluated NDSS
Available
Functional
Reproduced

Pitfalls for Security Isolation in Multi-CPU Systems

Simeon Hoffmann
CISPA Helmholtz Center for Information Security

Nils Ole Tippenhauer
CISPA Helmholtz Center for Information Security

***Abstract*—In embedded systems, the integration of multiple CPUs into one system on a chip (SoC) allows greater performance, and separation of tasks into independent firmwares and optimized architectures. For example, an ARM Cortex-M4 core could run the main firmware, and a Cortex-M0 core could run a real-time operating system (RTOS). Security implications of such integrations are still unclear, e.g. if an attacker with code execution on one CPU can fully compromise the second CPU, or leak protected data.**

In this work, we systematically identify security issues resulting from this integration, in particular related to memory and peripheral access control. These issues stem from re-use of single-CPU security mechanisms such as memory protection units (MPUs) in the new multi-CPU system. We identify four major attack vectors that can be present in such systems, and find that a significant number of systems on the market appear to be vulnerable. The attack vectors can lead to arbitrary read and write in protected memory of the other CPU, and even to code execution. In addition, we find that the communication mechanism of a popular open source RTOS, FreeRTOS [17], which is suggested as communication mechanism among firmwares on a multi-CPU system, introduces code execution vulnerabilities in the multi-CPU scenario. Then, we verify our theoretical predictions by implementing four attack vectors and demonstrate their practical efficacy. In addition, we find that in one case, the discovered attack surface may lead to the compromise of a custom trusted execution environment (TEE) implementation. We responsibly disclosed our findings to the vendors, resulting in a security advisory and a fix to a proprietary network stack implementation.

I. Introduction

Embedded systems are an integral part of everyday, modern life. As such, security vulnerabilities have an immediate impact on its users' well-being and can cause physical harm [15], [11]. Consequently, research created a plethora of techniques to analyze the security of embedded devices [40], [53], [62].

In recent years, we see a rising popularity in embedded devices that employ multiple CPUs integrated into one microcontroller unit (MCU) as a SoC [34], [42]. This allows the developer to split the tasks of the MCU in two firmwares, e.g., one firmware that performs safety-critical tasks and one firmware that performs the remaining tasks. In addition, developers can use this firmware split to introduce a security boundary, as each CPU runs its own firmware [47], [29]. While firmware is individual per CPU, remaining hardware such as memory and peripherals are shared among CPUs via various buses [2], [3], [4]. Depending on the MCU architecture, the CPUs have nearly unlimited access to all its components via those buses. Such architectures introduce a host of novel attack surfaces that currently is not well understood by researchers.

State-of-the art security analysis techniques for embedded systems are limited to analysis of single CPUs or firmwares. For example, recent advances in rehosting, a popular technique that aims to emulate the firmware hardware-less, focuses on peripheral interaction as wrong peripheral behavior blocks firmware execution [53]. However, rehosting analyzes each peripheral in isolation and does not capture inter-peripheral dependence, e.g., inter-CPU communication interfaces [16]. For a specific attack, Classen et al. analyzed two Bluetooth-WiFi chip architectures by Broadcom, in which Bluetooth and WiFi functionality is split between two CPUs [12]. The authors explore the communication channel specific to this architecture and identify vulnerabilities that can be exploited if one of the cores is malicious due to an inherent trust between both chips.

In this paper, we provide the first systematical assessment of security issues introduced by multi-CPU architectures in embedded devices. We investigate the general components of traditional, single-CPU embedded systems. We then analyze each component in the context of multi-CPU embedded devices and find that the introduction of more processors (each running individual firmware) introduces new attack surface. We distill the results into 4 general attack vectors.

To confirm the validity of these 4 attack vectors, we first perform a theoretical analysis of multi-CPU devices. To that end, we collect a list of 11 device families that have at least two CPUs. We theoretically investigate their vulnerability to the newly identified attack vectors. We find that 6 out of 11 devices are vulnerable to at least one of the attack vectors.

We implement our attacks on one of the vulnerable devices and practically verify our theoretical findings. We show that our attack vectors introduce arbitrary read, arbitrary write, and code execution primitives on the co-located CPUs or even in TEEs. Then, we demonstrate the practical relevance with a case study on two commercial products: the network stack of the STM32WB, and the security architecture of the Samsung Galaxy Ring. We find fundamental issues in both cases, and demonstrate practical exploits for a reproduction of the Galaxy Ring architecture. We disclosed our findings to the vendors, resulting in a security advisory and a patch.

Network and Distributed System Security (NDSS) Symposium 2026
23-27 February 2026, San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240971
www.ndss-symposium.org

Finally, we discuss countermeasures and conclude that software countermeasures cannot protect against these types of attacks, but they can help reduce the attack surface.

In summary, we make the following contributions:

- We systematically analyze cross-core attack surfaces in multi-CPU systems. We identify 4 novel attack vectors.
- We theoretically assess those vulnerabilities for the 11 most relevant multi-CPU SoCs on the market. We find that 6 of them are vulnerable to at least one attack.
- We practically implement our attack vectors on a vulnerable device and prove that all attacks work in practice. We find issues in two commercial products and responsibly disclosed our findings to the vendors.

To foster further research in this area, we publish our source code at https://github.com/scy-phy/multicpu-pitfalls.

II. Background

A. Memory-Mapped I/O

To allow interaction on a software level, peripherals connected to an embedded system are accessible via memory-mapped I/O (MMIO). Each peripheral is assigned a memory region, and each location in this memory region is called a MMIO register. The firmware can interact with the MMIO registers via regular load and store instructions. Unlike in normal memory, however, a read from a MMIO register returns a value provided from the peripheral of this memory range. A write to an MMIO register writes data to the peripheral of this memory range.

B. Bus Architecture

The actual communication from the CPU running the firmware and its peripherals happens via the bus architecture. The bus architecture is the physical connection of all the resources on the MCU. Modern MCUs employ multiple buses to group resources together. A single bus connects multiple resources. Each resource is connected either as *bus maintainer* or as *bus subordinate*. A bus maintainer can issue a request on the bus, a bus subordinate can merely respond to requests of bus maintainers. Typical bus maintainers are CPUs, while typical bus subordinates are most peripherals, such as, Universal Asynchronous Receiver-Transmitter (UART).

C. Direct Memory Access and Memory Protection Unit

Two features of modern MCUs deserve special attention: direct memory access (DMA) and the MPU.

DMA. DMA is an asynchronous communication mechanism on modern MCUs. It is a technqiue to copy memory without CPU involvement. Consequently, all DMA-enabled peripherals are connected as bus maintainers.

MPU. An MPU is a mechanism of modern MCUs to limit access to certain memory areas. The MPU comes as an optional part of the CPU, but firmware interacts with the MPU in the same way as the firmware interacts with peripherals, via MMIO. The MPU is disabled at firmware start, providing no protection. In privileged mode, the firmware can write the MPU configuration and enable or disable the MPU. The

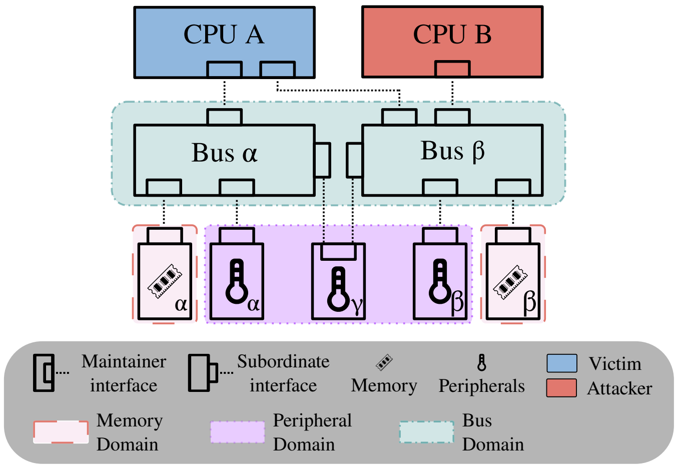

Fig. 1. A simplified multi-CPU system. CPU A is connected to bus α, the single-connected memory α and the single-connected peripheral α. In addition, both CPU A and CPU B are connected to bus β, the multi-connected memory β and the multi-connected peripheral β.

MPU configuration can define a fixed number of memory regions and access permissions for these memory regions. For example, the firmware may employ an RTOS which can access privileged mode. The RTOS runs tasks in unprivileged mode. The RTOS can configure a memory region as privilege access only. This creates a memory area that only the RTOS can access, e.g. to store cryptographic secrets.

D. Multi-CPU Systems

A multi-CPU system is an MCU that employs at least 2 CPUs. Each CPU runs its own firmware, independent of the firmware on the other CPUs. In addition, each CPU has its own bus architecture, connected to its own resources, such as peripherals and the system's memory. Usually, some parts of these bus architectures are bridged. This allows a bus maintainer to access the bridged resources as if they were connected to the maintainer's own bus.

Consider the following example. The MCU consists of CPU A and CPU B. CPU A is connected to a bus, and a UART peripheral is also connected to this bus. This bus is bridged to CPU B. CPU B can thus interact with the UART peripheral as if it were connected to its own bus architecture. This UART peripheral occupies a dedicated area in MMIO memory, in the same way a UART connected to CPU B would.

III. Attack Vectors On Multi-Processor Embedded Systems

We now analyze the architecture of multi-processor embedded systems. We investigate each part of the architecture (categorized as peripherals, buses, and the memory), and additionally, examine the inter-CPU communication channels. As result of our analysis, we present 4 attack vectors.

A. System Architecture

We assume a multi-CPU embedded device with at least two CPUs. All CPUs are ARM Cortex-M [6] CPUs. All

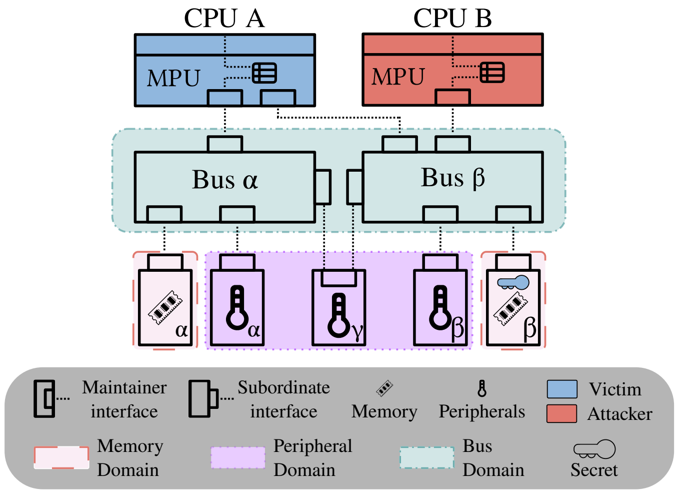

Fig. 2. An example of the MPU policy desynchronization attack vector. The attacker reconfigured the MPU on the attacker CPU to allow access to the victim's secret in memory β. Now the attacker can read the secret.

CPUs are connected to peripherals and memory via the bus structure. Figure 1 shows a simplified overview of a multi-CPU system. CPU A and CPU B are both connected to bus β. Bus β connects to memory β and peripherals β and γ, and consequently, both CPU A and CPU B can access memory β as well as peripherals β and γ. We call memory and peripherals that are accessible by both CPUs *multi-connected* memory or peripherals. In addition, CPU A is connected to bus α and can thus access the connected memory α and peripheral α. CPU B cannot access memory α or peripheral α. If a memory or a peripheral can only be accessed by a single CPU, we term this memory or peripheral *single-connected* memory or peripheral.

B. Threat Model

We assume that an attacker was able to compromise the firmware of one CPU (CPU B in Figure 1) of a multi-CPU system during runtime. They are able to execute arbitrary code and can access all resources that the respective CPU has. In particular, the attacker can access all multi- and single-connected peripherals and memory that are connected to the compromised CPU. The attacker cannot, however, access single-connected memory or peripherals that are not connected to the compromised CPU. The other CPU (CPU A in Figure 1) obtains a secret and stores this secret in the system's memory. The goal of the attacker is to learn this secret.

Related work uses a similar attacker model [12]. In addition, other work that crosses security boundaries, e.g., by gaining access to a TEE, require the same attacker model [52], [26].

In the sample architecture provided by Nordic [55], one CPU takes care of network communication and the other CPU runs the application. This networking CPU needs to execute Nordic's proprietary, closed-source network stack implementation. Our attack scenario corresponds to a network attacker that compromised the network stack or a rogue network stack. This attacker now wants to access secret application data. Note that some manufacturers explicitly advertise multiple CPUs as a security feature: for example, ST Microelectronics claims in their security note that "In dual-core products, one core can act as secure while the other is nonsecure" [29]. Our attack scenario challenges this claim.

In the following, we consider a dual-CPU MCU, similar to Figure 1. We assume that CPU A is not compromised, while the attacker controls CPU B. We use CPU B and attacker (as well as CPU A and victim) interchangeably.

C. Attacks Against The Memory

Malicious access to memory is a known problem on (single-CPU) embedded devices [1]. In particular, as the memory containing the stack is often executable (even today [28]), overwriting a single return address with a pointer to attacker-controlled stack memory immediately results in code execution. We find many existing defenses against these attacks in the literature [39], [14], [1].

Accessing Connected Memories Directly. If the stack of the victim CPU A is in a multi-connected memory, this "jump to shellcode" attack also works cross-CPU. Once an attacker obtained code execution on CPU B, they can inject code into the multi-connected memory of CPU A, and modify a return address to execute it.

To defend against these types of attacks on single-CPU devices, vendors introduced MPUs. The defenses introduced above rely on these MPUs. They leverage the MPU to restrict access to peripherals and memory areas. However, MPUs are implemented as part of a CPU. Consequently, the MPU on CPU A only restricts accesses of CPU A-executed firmware, and the MPU on CPU B moderates accesses of CPU B-executed firmware. Consequently, to achieve similar guarantees in a multi-CPU system compared to a single-CPU system, all CPUs need to provide an MPU. Furthermore, all MPU policies need to be semantically equivalent with respect to multi-connected memories or peripherals and changes to one policy need to be reflected in all policies.

Consider Figure 2. CPU A is connected to single-connected memory α and single-connected peripherals α via bus α. In addition, CPU A is connected to multi-connected memory β and multi-connected peripherals β via bus β. CPU B is only connected to multi-connected memory β and multi-connected peripherals β via bus β. Both CPU A and CPU B employ an MPU. Assume that CPU A stores a secret, e.g., cryptographic keys, in multi-connected memory β. CPU A restricts access to the secret via an MPU policy. CPU B starts with a semantically equivalent policy. This policy denies all access from CPU B. Consequently, the semantic meaning of both policies is that only CPU A can access this memory area. If the attacker tries to access the secret, the MPU denies access to it.

We find that this setting allows the following attack. If the attacker on CPU B can reconfigure CPU B's MPU (which requires privileged code execution, default in many firmwares [14]), the attacker can change its policy to enable access to the memory that the victim uses to store its secret –

without a possibility for CPU A to notice this change in MPU configuration. The attacker effectively introduced a semantic inequality of the MPU policies that allowed them to access previously inaccessible memory. We term this attack MPU policy desynchronization attack.

Attack vector 1: MPU policy desynchronization.

Attack vector 1 allows an attacker to access a secret in multi-connected memory. It does not allow an attacker to access a secret in single-connected memory. We introduce other vectors to allow access to a secret in single-connected memory later.

Communication. We find that multi-CPU embedded devices commonly use multi-connected memory to share data between the CPUs. This matches with the state of the art in general-purpose computing. The two predominant methods of communication in modern systems are message passing and shared memory. Shared memory requires a shared memory space, a synchronization primitive (that is, locks or semaphores), and the mutual agreement to not modify the memory while not in possession of the synchronization primitive. Consequently, the memory region containing the shared memory needs to reside in a memory region that is available to all communication participants.

In our multi-CPU embedded device setting, this requires the implementation of shared memory to use multi-connected memory. To synchronize shared memory access, all communication participants need to respect the synchronization primitive. The communication can only work as intended in either a cooperative setting (all communication participants are benign and wait for their turn) or a setting where the access locking mechanism is enforced by a third party, like an operating system. In our threat model, the communication participants are not guaranteed to be cooperative (as the attacker chooses if they want to cooperate or not) and there is no operating system across all CPUs. Consequently, if an attacker controls CPU B, the attacker can freely access the shared memory. This enables the attacker to perform typical time-of-check-to-time-of-use (TOCTOU)-style attacks.

Assume CPU A wants to query CPU B, which is attacker-controlled, on some data to use in CPU A's processing. CPU B acquires the lock, writes to the shared memory channel, and releases the lock again. Now CPU A acquires the lock, performs a check on the data, and then uses it to perform a sensitive action. If CPU B ignores the lock, it can modify the data after CPU A performed the check, but before the data is used. This allows the attacker to provide *unchecked data to the sensitive action of CPU A*. Hence, we find that shared memory cannot enable reliable communication in our scenario, as neither hardware nor an operating system can enforce the synchronization primitive.

Message passing, on the other hand, utilizes a hardware-enforced queue-like structure. Communicating parties can only perform operations on the queue, and not access the underlying storage space. We find that the only way to implement message passing in multi-CPU MCUs is via a dedicated peripheral. Unlike shared memory, this can serve as a secure communication channel. CPU B, the attacker, can choose to write arbitrary data to the queue. But once the data is written to the queue, the attacker loses control over the data. Hence, CPU A can take the data out of the queue, perform its checks and act accordingly without CPU B, the attacker, interfering after the checks. In summary, we identify unsynchronized communication channels as our second attack vector.

Attack vector 2: Unsynchronized communication channels.

This attack allows an attacker to leverage data in multi-connected memory to potentially access data in single-connected memory. The exact capabilities of an attacker depend on the implementation of the shared memory communication mechanism.

D. Attacks Against Peripherals

In this part, we analyze the second category, peripherals. As mentioned in Section II-A, the CPUs interact with peripherals via memory operations. If multiple CPUs want to access the same peripheral, they thus need to coordinate.

Resource Contention. As mentioned in Section II-A, peripherals are mapped to fixed memory regions. The firmware interacts with these peripherals via memory reads and writes. Consequently, we find that multi-connected peripherals suffer from a similar problem as multi-connected memory: shared usage requires cooperation. If the victim wants to use a multi-connected peripheral, it requires the agreement that the attacker does not use it during that time. Otherwise, similar TOCTOU-style attacks are possible.

Assume CPU A wants to use a multi-connected, DMA-enabled Ethernet peripheral to receive a secret. After CPU A setup the peripheral, the attacker can overwrite the destination pointer to point to an attacker-controlled area. The multi-connected Ethernet peripheral will then write the secret to the attacker-controlled area and consequently, the attacker is in possession of the secret. We identify the possibility of changing the configuration of a peripheral from one CPU while the other CPU is using it as third attack vector.

Attack vector 3: Non-exclusive peripheral access.

E. Attacking the Bus Layout

As explained in Section II-B, bus maintainers issue transactions to the devices they are connected to. Typical bus maintainers include the CPUs, but also DMA-enabled devices. Those are the DMA peripherals themselves, but also different peripherals with dedicated, integrated DMA capabilities, such as Ethernet, USB or SD card controllers [28].

Confused Peripherals. Consider again Figure 1, the simplified multi-CPU system. Resource access is not symmetrical:

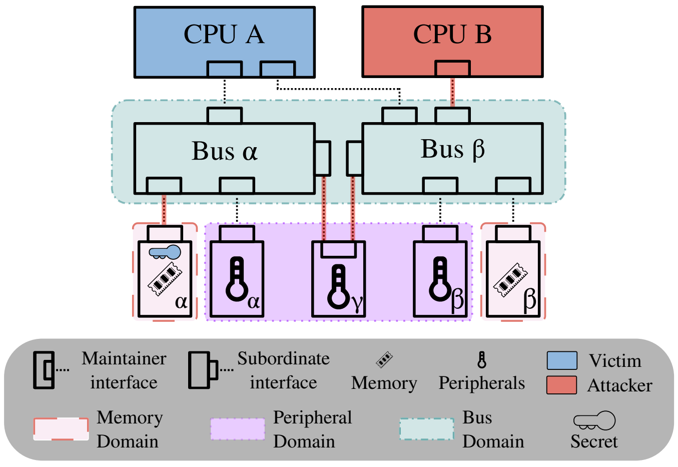

Fig. 3. An example of the confused deputy peripheral attack vector. The attacker cannot access the secret in memory α directly, as memory α is a single-connected memory. However, the attacker can configure peripheral γ, which has bus maintainer access to the memory, to access the secret on the attacker's behalf.

TABLE I
ATTACK VECTOR OVERVIEW. EACH ATTACK CAN ACHIEVE DIFFERENT EFFECTS GIVEN DIFFERENT REQUIREMENTS.

Attack vector	Target	Requirement	Effect
MPU policy desynchronization	MPU-prot., multi-conn. memory	privileged code execution	Read/write multi-connected memory
Unsynchronized comm. channels	multi-conn. memory	no hardware comm. mech.	Read/write single-/multi-conn. memory
Non-exclusive peripheral access	multi-conn. peripheral	none	Read/Write data from/to peripheral
Confused deputy peripheral	single-conn. memory	peripheral with diff. bus access	Read/write single-connected memory data

CPU A can access resources that CPU B cannot access. CPU A can access bus α, which is connected to the single-connected memory α and single-connected peripherals α and γ. CPU B is physically not connected to bus α and as such, has no direct access options to memory α and peripheral α. However, CPU B has access to peripheral γ, which is connected to bus α. As explained in Section II-B, all bus maintainers on a bus can access the resources on this bus. Some peripherals, such as DMA-enabled peripherals, have a maintainer interface on busses that they connect to. We find that a security problem may arise if the attacker cannot access a bus directly (bus α in this example), but can access a peripheral that is a bus maintainer on this bus.

Consider Figure 3. CPU A is connected to bus α, which connects to single-connected memory α. CPU A and B are connected to multi-connected memory on bus β. The single-connected memory α is thus not accessible to CPU B. The victim leverages the fact that the single-connected memory α is not available to CPU B and stores the secret in its memory. The attacker cannot access this memory, as CPU B, which the attacker controls, it not connected to the memory.

However, the attacker can access peripheral γ. Peripheral γ is connected to both bus α and bus β with a maintainer interface. Peripheral γ can thus access connected peripherals on both bus α and bus β. This includes the single-connected memory α. Consequently, as the attacker has access to a maintainer interface on bus β, the attacker can configure peripheral γ to access the secret on the attacker's behalf. This violates CPU A's assumption that CPU B cannot access the single-connected memory α, and may lead to security vulnerabilities, e.g., a compromise of the secret stored in the single-connected memory α. We identify access discrepancies among different, connected maintainer peripherals as our fourth attack vector.

Attack vector 4: Confused deputy peripheral.

F. Attack Vector Combination

We discovered four attack vectors with different requirements to achieve different effects. The attacks are summarized in Table I.

We find that in a setup where the secret is stored in multi-connected memory, and the attacker has privileged code execution, attack vector 1 is sufficient to steal the secret. We note that bare-metal firmware usually runs in privileged mode [14]. However, in a setup where the secret is stored in single-connected memory (or the attacker has no privileged code execution), attack vector 1 alone does not suffice. Instead, the attacker may require a combination of attack vectors to achieve the goal of stealing a peripheral-received secret.

For example, there may be an MPU policy protecting the peripherals, and the secret is stored in single-connected memory. An attacker may need to perform an MPU policy desynchronization attack to gain access to the peripheral. Afterwards, the attacker employs a confused deputy peripheral attack against the peripheral to steal the secret from single-connected memory.

IV. VULNERABILITY ASSESSMENT OF DEVICES ON THE MARKET

In this section, we survey relevant multi-CPU MCUs available on the market, and assess their potential vulnerability to our four attack vectors based on architectural choices made. To that end, we first collect a dataset of MCUs of the architecture we presented. Then, for each attack vector introduced in Section III, we investigate the vulnerability of the device against the attack vector.

A. Dataset Collection

We identify the 11 most important MCU vendors [24] and collect all multi-CPU MCU families where all CPUs implemented the ARM Cortex-M standard. Additionally, we add two more manufacturers popular in the community. Raspberry Pi is (as part of the Raspberry Pi family) particularly popular among end users [63]. Nordic Semiconductor is a popular

TABLE II
MCUs WITH MULTIPLE CPUs, RELEASE (MONTH AND YEAR), THEIR EMPLOYED CPU CONFIGURATION AND THEIR POTENTIAL VULNERABILITY TO THE ATTACK VECTORS FROM SECTION III. ● MEANS THE MCU IS VULNERABLE TO THE ATTACK (AND ○ MEANS IT IS NOT VULNERABLE). ◐ DENOTES A DEVICE WE DEEM VULNERABLE, BUT THE MCU EMPLOYS A COMPLETELY SYMMETRICAL MEMORY MAP. ◑ INDICATES THAT NOT ALL OF THE CPUs EMPLOY AN MPU. ◓ MEANS PARTIALLY VULNERABLE.

MCU Name	Vendor	Release	Cortex-M CPUs	V1: MPU	V2: comm.	V3: peripherals	V4: conf. dep. periph.
NXP K32L3 [44]	NXP	09/2019 [43]	1x M4 & 1x M0	●	●	●	●
NXP LPC43xx [41]	NXP	12/2012 [45]	1x M4 & 1x M0	●	●	◐	◑
STM32H745/755 [28]	ST	06/2019 [35]	1x M7 & 1x M4	●	●	●	●
STM32WB55 [38]	ST	02/2019 [32]	1x M4 & 1x M0	◑	●	●	●
STM32WL5x [37]	ST	01/2020 [36]	1x M4 & 1x M0	○	●	○	○
XMC7000 [21]	Infineon	11/2022 [23]	1-2x M7 & 1x M0	○	○	○	○
Traveo T2G [20]	Infineon	2017[1]	1x M7/M4 & 1x M0	○	○	○	○
PSoC6 [19]	Infineon	09/2017[2]	1x M4 & 1x M0	○	○	○	○
RP2040 [48]	Raspberry Pi	01/2021 [46]	2x M0	●	○	◐	●
RP2350 [47]	Raspberry Pi	08/2024 [47]	2x M33	○	○	○	○
nRF5340 [57]	Nordic	11/2019 [56]	2x M33	○	◓	○	○

[1]: Due to Infineon's takeover of Cypress, much information was lost. The earliest entry in the wayback machine for this MCU is from 2017 [54]
[2]: Due to Infineon's takeover of Cypress, much information was lost. The wayback machine seems to date the public release to 09/2017 [22]

manufacturer in the wireless domain [62]. As platform, we choose ARM due to its wide adoption in practice [8] and focus on the Cortex-M standard as this is the standard for low-power embedded devices [6]. Notably, there are also devices that employ Cortex-M and Cortex-A cores on the same MCU. However, the Cortex-A standard supports more advanced security features, such as an MMU [5], and the Cortex-A cores are generally strong enough to run a general-purpose operating system, such as Linux. Consequently, it requires a different analysis which we leave for future work.

We collected a total of 11 MCU families across 5 different vendors, summarized in Table II. Among these 11 families, we find 4 different Cortex-M CPUs (Cortex-M7, Cortex-M4, Cortex-M33, and Cortex-M0). Most of them provide exactly 2 CPUs, with the XMC7000 [21] being the only exception (it employs up to 3 CPUs). The most common scenario is an asymmetrical setup of a more powerful and a less powerful processor, such as a Cortex-M4 and a Cortex-M0, or a Cortex-M7 and a Cortex-M4. Three device families use a symmetrical setup: the RP2040 [48] employs 2 Cortex-M0 CPUs, the RP2350 [49] and the nRF5340 [57] provide 2 Cortex-M33 CPUs each. We note that we analyze device *families*. The LPC43xx device family alone comprises 61 actively supported devices [41]. Additionally, we observe that *all* of these board families, except for the RP2040, advertise security as a feature, with statements such as "enhance application security" [28] or "advanced security" [44]. ST Microelectronics published an application note that states that if a device employs multiple CPUs, one of them can be a designated secure CPU while the other is a non-secure CPU [29].

B. Prevention Characteristics

In order to identify if a device from our data set is vulnerable to one of the identified attack vectors, we identify *prevention characteristics* that are required to prevent the identified attack vectors. First, we perform a review of all devices from our data set. We analyze the features of the individual boards and investigate for each feature if it can prevent one of the previously identified attack vectors. This results in a set of mechanisms that can eliminate one or multiple attack vectors introduced in Section III. We deem a device vulnerable to an attack vector if it does not implement any mechanism to defend against the attack vector. Note that the existence of such a characteristic does not make a device secure by default. There is still room for misconfiguration. Consequently, our evaluation is an upper bound of the achievable security against the attack vectors introduced before.

C. AV 1: MPU Policy Desynchronization

We abuse the fact that MPU policies are individual to and programmable from their respective CPUs. Consequently, an attacker that controls CPU B can reconfigure its MPU policy to nullify all protection given by this MPU (see Figure 2).

We distinguish between 2 different situations. In some multi-CPU systems, all CPUs are MPU-protected while others only implement an MPU on a subset of their CPUs. While this attack vector considers MPU policy desynchronization, a non-existent MPU is functionally equivalent to no MPU policy. Thus, the attack vector also applies to multi-CPU MCUs where only a subset of the CPUs employs an MPU.

Prevention Characteristic. This attack vector is caused by bringing multiple MPU policies out of sync with respect to multi-connected peripherals or memory. In order to address this problem, we need a single MPU-like peripheral that keeps a policy for the whole device. As this mechanism needs to manage permissions for multiple bus maintainers, it cannot be implemented as part of a single bus maintainer. We find that a memory protection mechanism implemented on the bus defends against this attack vector. This memory protection mechanism defines access permissions of all bus maintainers uniformly on one peripheral external to the CPUs. These peripherals are usually initialized during a secure boot [37], or only configurable by exactly one bus maintainer [57]. Consequently, CPU B cannot choose to reconfigure the MPU to elevate its privileges.

Findings. Consider Table II. In our dataset, we observe that 5 out of 11 CPU families appear vulnerable to this attack vector. The vulnerable devices comprise of the NXP devices, the ST devices minus the STM32WL5x, and the RP2040. 2 out of these 5 do not implement MPUs on all of their cores (NXP LPC43xx and STM32WB55 family). As discussed before, a missing MPU is functionally equivalent to no MPU policy.

The 6 families that are not vulnerable are all Infineon families, the nRF5340, the STM32WL5x, and the RP2350. They all implement an access control mechanism on the bus level that assigns permissions to bus maintainers. These access control mechanisms work similar to a regular MPU: they are configured and enabled once, and this configuration can later only be changed from a secure access context (the exact implementation is manufacturer-specific). This allows for fine-grained configuration of access permissions per bus maintainer. We find a slight variation on the RP2350: this device employs ARM TrustZone [7] on both CPUs. Unlike an MPU per CPU, one instance of TrustZone per CPU can be configured securely, as the access permission check happens on the bus level. Consequently, enabling TrustZone on one CPU and disabling TrustZone entirely on the second CPU prevents the attack.

D. AV 2: Unsynchronized Communication Channels

In this attack vector, we attack the communication among the MCUs. We observe two communication paradigms in our dataset: communication via shared memory and communication via processor queues, corresponding to the mechanisms introduced in Section III.

Hardware Locking. The shared memory paradigm uses a semaphore peripheral. CPU A and CPU B assign a specific meaning to each semaphore, e.g., semaphore 1 protects memory 1. If any CPU performs a write access, it marks the semaphore, and after the write access, it unmarks the semaphore again. This unmarking can be configured to trigger an interrupt on the other CPU to notify it of the updated memory. Crucially, there is no connection between memory area and its protecting semaphore. Both CPU A and CPU B implicitly connect it to a memory area. If one of the CPUs is attacker-controlled, the attacker does not need to respect this agreement and can change the memory content at will. Consider the following example: CPU A and CPU B communicate via a queue in shared memory. The attacker controls CPU B and changes the pointer to the next element. The next time CPU A inserts an element in the queue, it is written to an attacker-controlled address. While CPU A can perform a sanity check on the queue's next pointer, this only narrows the time window that the attacker on CPU B has to write the pointer, as CPU A can never get exclusive access, resulting in a TOCTOU-style attack.

Processor Queues. The processor queues communication paradigm uses a queue mechanism in hardware. If CPU B wants to send a message to CPU A, it inserts the data in this queue. CPU A receives an interrupt that new data is there. At this point, CPU B has handed over its control over the communication data to the hardware queue mechanism. A modification is no longer possible. CPU A can now perform its sanity check and CPU B cannot change the data in the window between validation and use of the data.

Prevention Characteristic. The prevention characteristic that we identify for this attack vector is the existence of a hardware queue mechanism. If an MCU implements a queueing mechanism, firmware can implement secure communication among the CPUs. If there is only a shared memory-based approach, secure communication is not possible. Note that if both exist, secure communication is possible by choosing the queue mechanism, and consequently, we deem the device not vulnerable.

Note that protection from an external MPU, similar to the defense from Section IV-C, is no defense in this scenario, as the purpose of the shared memory is to be readable and writeable by both CPUs. If CPU B cannot write to the shared memory, the channel immediately becomes unidirectional, which is insufficient in the general case. Consequently, we deem an external MPU protection mechanism an insufficient protection mechanism against this attack vector.

Findings. Consider Table II. Out of the 11 investigated board families, we find that 5 appear vulnerable. These vulnerable boards implement only communication via shared memory and hardware locks. We note that the nRF5340 [57] has an IPC peripheral with a "GPMEM" field. This field stores general purpose data. However, the peripheral employs 16 individual IPC structs and only 2 "GPMEM" fields. It is unclear if the "GPMEM" fields can be used to pass data. Even if they can be used, the number of channels is limited to 2, while the number of IPC structs is 16. Thus, 14 of the 16 structs are insecure, 2 can be used securely. We assign ◓.

We find that all 3 Infineon families (the XMC7000, PSoC6 and Traveo T2G series) present a special case: they do not explicitly introduce queue structures, but they introduce an IPC struct that implements a locking mechanism with 2 data fields. These data fields are only accessible as long as the CPU owns the respective IPC struct lock. Consequently, Infineon implemented a hardware bond between the data fields and the lock. These data fields can hence be used similar to a queue mechanism with queue size one and allow secure data exchange. The Infineon devices are thus not vulnerable to the unsynchronized communication channel attack vector, even without implementing a true queueing mechanism.

The RP2040 and the RP2350 are only MCUs that implement a true queueing mechanism. Each CPU has a read and a write register, and writing to the write register in CPU A results in data in the read register in CPU B, and vice versa. The moment that CPU A writes data to the write register, it gives up control over the data and can no longer change it. Consequently, the RP2040 and the RP2350 are not vulnerable to the unsynchronized communication channel attack vector.

E. AV 3: Non-exclusive Peripheral Access

In this attack vector, we aim to change peripheral configuration data of the victim between configuration and usage.

Prevention Characteristic. The prevention characteristic for this attack vector is the existence of an access control mechanism for peripherals that works simultaneously for all bus maintainers. Therefore, this access control mechanism must be implemented as part of the bus architecture, unlike, e.g., an MPU which is implemented as part of the CPU.

Findings. Consider Table II. We find that 5 device families do not implement such a control mechanism and are thus likely vulnerable to the attack. Again, the NXP device families are vulnerable, as well as the ST boards except the STM32WL5x. The RP2040 is also vulnerable, while the RP2350 is not.

The 6 families that are not vulnerable are again all 3 Infineon families, the nRF5340, the RP2350 and the aforementioned STM32WL5x. They implement an access control mechanism that allows to create an access policy for bus maintainers. If a bus maintainer wants to access a resource, the mechanism moderates bus maintainer access at runtime according to the policy. This mechanism is configured and enabled once, similar to an MPU, and later only accesses from a secure access context can change the policy again. The exact details are implementation-specific.

F. AV 4: Confused Deputy Peripheral

In this attack vector, we abuse the fact that all bus maintainers can read/write from/to the bus, and some can do so on behalf of others. It can thus happen that the attacker CPU cannot access a particular resource, e.g., the single-connected memory α in Figure 3, but another bus maintainer (multi-connected peripheral γ) can access said resource, and the attacker CPU can configure it. Thus, the attacker CPU can still access the resource (indirectly) via the other bus maintainer.

Prevention Characteristic. We identify two prevention characteristics for this attack vector. First, a similar mechanism as the one introduced in Section IV-D, access control for peripherals, can protect against this attack vector. The access control mechanism can ensure that all accessible peripherals for a CPU have the same access as the original CPU. Consequently, some peripherals are made unavailable for a CPU.

The second prevention characteristic is also a prevention characteristic for the attack in Section IV-C: a memory access control mechanism. This memory access control mechanism must be implemented external to all CPUs and must not be configurable by the CPUs. Now, the attacker can no longer use the different privileges of other bus maintainers to access otherwise inaccessible memory regions as each bus maintainer has its own set of access privileges.

Findings. Consider Table II. In our analysis, 5 out of 11 boards allow this style of attack, and the board families that appear vulnerable are the same as in Section IV-E. The vulnerable families are the NXP boards, the ST boards without the STM32WL5x, and the RP2040. Out of these 5 boards, 3 provide resources that are only available to one of the CPUs. For example, the STM32H755 [28] provides two memory regions exclusively to the M7 core. A firmware engineer intuitively assumes that this memory is only available to one of the processors, thus it can store secrets that are exclusive to this processor. The M4 cannot read it, but can configure the DMA controller to copy data from this memory to M4-accessible memory and vice versa. The attacker can leverage this power to steal confidential information from otherwise unaccessible memory.

The 6 families that are not vulnerable (again, all Infineon families, the nRF5340, the RP2350 and the STM32WL5x) implement both a peripheral protection mechanism as well as a memory protection mechanism. If configured correctly, these mechanisms can protect against the confused deputy peripheral attack vector.

V. Practical Validation of our Attack Vectors

We practically validate our theoretical findings by implementing an attack for each of the attack vectors that we identified. We provide the source of our experiments at https://github.com/scy-phy/multicpu-pitfalls. For each attack, we first introduce the questions we aim to answer with the implementation, then we describe our attack in detail, and summarize the results of our evaluation.

Experimental Setup. We evaluate our attacks in two steps: first, in this Section, we implement our experiments on a development board from ST Microelectronics. Second, in Section VI, we present two case studies on real-world firmware.

For our experiments, we choose the NUCLEO-H755ZI-Q, a development board for the STM32H755 [28]. We choose this board because ST Microelectronics makes the strongest claims about security of their multi-CPU devices [29]. This board is one of the devices from Section IV that is vulnerable to all attack vectors, and it has a single-connected memory area. In addition, it provides a good development environment with many open-source samples. To load an existing example, we use the STM32CubeMX [58], select the NUCLEO-H755ZI-Q as the target board and pick one of the examples in the example selector. To generate an empty example, we select the STM32H755 in the MCU selector and configure the peripherals as required. In our experiments, the M4 is the attacker core and the M7 the defending core. We choose this configuration because the M7 has access to memory regions that the M4 cannot access. We use the terms M4 and attacker, as well as M7 and victim, interchangeably.

A. AV 1: MPU Policy Desynchronization

With this attack vector, we want to introduce semantic discrepancies among multiple MPU configurations. In this part of the evaluation, we want to answer the following question:

Q1 Can we modify the MPU on the attacker CPU to introduce semantic discrepancies such that the attacker can access previously protected memory?

Experimental Setup. We answer this question with two experiments. Both experiments are based on an empty sample. The victim stores a secret value in a memory region and marks this memory region as inaccessible by anyone. The attacker nonetheless wants to access this memory region.

Experiment 1. In the first experiment, which serves as ground truth, the attacker firmware also configures the MPU in

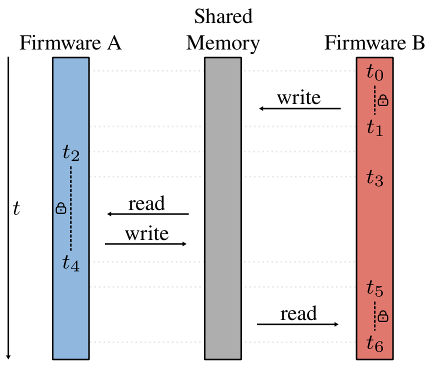

Fig. 4. Overview of an unsynchronized communication channels attack. The firmwares hold a synchronization primitive in the intervals indicated by the lock. In Experiment 1, the attacker tries to obtain the synchronization primitive at t_3 and fails. In Experiment 2, the attacker ignores the synchronization primitive, and can thus modify the data at t_3 without taking the synchronization primitive.

a semantically equivalent way: the MPU policy prohibits any access to this memory area. Now the attacker tries to access this memory area. This results in a fault. Thus, semantically equivalent MPU policies can protect a memory region.

Experiment 2. In the second experiment, the attacker introduces discrepancies in the MPU configuration on its CPU. The attacker uses their power to execute arbitrary code to reconfigure a peripheral. In this experiment, this attacker-controlled code disables the MPU, allowing full access to all regions for anyone. In our experiment, we simulate this scenario by disabling (not configuring) the MPU. This results in no restrictions for all code running on the attacker CPU. The attacker then tries to access the secret value that is protected by the M7 MPU policy. The attacker can freely retrieve this value. Consequently, this introduced discrepancy between the individual MPUs compromises protection.

Summary. As projected in Section IV, the attacker can introduce MPU access discrepancies to gain access to memory regions that are protected by the victim MCU (**Q1**).

B. AV 2: Unsynchronized Communication Channels

In this section, we want to answer the following questions:

Q2 Can we modify the shared memory from the attacker-controlled CPU while the victim CPU holds the associated synchronization primitive?

Q3 If yes, what is the impact on security?

Experimental Setup. To answer these questions, we design 2 experiments. The experiments base on a modified version of the `FreeRTOS_AMP_Dual_RTOS` example. This example uses the message buffer implementation [18] of the popular RTOS FreeRTOS [17] to send data from one CPU to the other CPU. This implementation uses a queue mechanism in shared memory to exchange data. We modify this example to an echo example: the attacker sends a message to the defender, and the defender replies with the same message. Figure 4 shows the general structure. The attacker writes a message to the queue in shared memory. The victim reads the message and sends it back to the queue. The attacker now reads the message again. The attack, in this scenario, is a modification attempt at time t_3. At this point in time, the victim owns the synchronization primitive, but the attacker tries to access the resource.

Experiment 1. The first experiment assumes cooperating firmwares. In a cooperative setting, both CPUs use a synchronization mechanism to assert that no concurrent accesses happen. In this experiment, we implement this synchronization mechanism with a hardware lock. To access the shared memory, a CPU needs to hold this hardware lock (**Q2**). This serves as ground truth to prove that synchronization is possible in a cooperative scenario.

Refer to Figure 4. The attacker first acquires the lock at t_0, writes data to the shared queue, then releases the lock at t_1. The victim firmware acquires the lock at t_2 to read from the queue. Now, at t_3, the attacker tries to access the shared memory. The victim firmware continues by reading the resource and writing it back. After writing it back, the victim firmware releases the lock at time t_4. Now, the attacker firmware can acquire the lock again and reads the shared memory location.

As this experiment assumes a cooperative setting, the attacker first tries to obtain the lock to access the shared resource at t_3, but fails to do so. Consequently, the attacker cannot modify the shared resource at t_3, because they do not hold the lock. Thus the attack is not possible.

Experiment 2. In the second experiment, we change the assumption that the attacker cooperates. We use the same firmware as in Experiment 1, that is we add locking logic around accesses to the shared resource. However, this time, we do not assume cooperation, thus the attacker does not respect the lock. Consequently, we remove the lock-holding logic from the attacker firmware when trying to access the shared resource at time t_3.

Consult again Figure 4. The attacker releases the lock at time t_1, then the victim acquires the lock at t_2. Any checks to the consistency of the state of the shared resource happen now, as the victim has sole access to the shared resource. However, our attacker does not respect the lock, thus tries to access the shared memory without holding the lock at time t_3. As there is no entity enforcing the lock, the attacker can freely modify the shared resource (**Q2**).

To answer **Q3**, we first investigate the shared resource in more detail. Inspecting the FreeRTOS message buffer implementation shows that it uses a queue in shared memory. Sending data to the message buffer writes data to a buffer area at the write pointer and advances the write pointer, receiving data from the message buffer reads data from the buffer area at the read pointer and advances the read pointer.

As our attacker does not respect the lock, our attacker can modify the shared memory at any point in time, but in particular at time t_3. To achieve an arbitrary read primitive, the attacker can manipulate the read pointer at time t_3. The next time the victim retrieves data from the message buffer, the victim instead reads from the manipulated pointer location. In our echo scenario, the victim then sends this read data

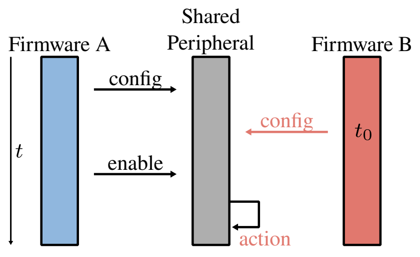

Fig. 5. A non-exclusive peripheral access attack. The attacker reconfigures the peripheral after victim configuration, but before victim usage. Consequently the peripheral performs an attacker-desired action.

to the attacker. Our attacker can achieve an arbitrary write primitive in the same way: by modifying the write pointer at time t_3. The victim receives (attacker-controlled!) data and writes the data back to the write pointer. However, as the attacker manipulated the write pointer, it instead writes to an attacker-controlled location. These two primitives are enough to gain code execution on the victim CPU. The attacker can thus take over the victim CPU in this experiment. **Beating Proprietary Defenses.** Interestingly, we find that on this specific MCU, the manufacturer provides 2 proprietary defense mechanisms: readout protection (RDP) and proprietary code readout protection (PCROP) [31]. These mechanisms allow the developer to mark a memory area of the device as a secure world. Only code running inside this secure world can access code and data inside the secure world, accesses from the outside return no data. Only the M7 CPU can use this secure world feature. These proprietary defense mechanisms thus introduce a TEE. The M7 enters this TEE after boot, but, once it exits the TEE, no reentry is possible except via reboot. This differs from other TEE implementations, such as Intel SGX [25] or Arm TrustZone [7].

Assume the following example: the victim acts as a cryptographic service provider. It owns cryptographic material, uses it to sign a given message, and sends the original message and the signature back to the attacker. Notably, to keep access to the cryptographic material inside the TEE, the victim firmware permanently runs inside the TEE. Consequently, the read and write primitive obtained via attack vector 1 allow arbitrary read and write *inside the TEE*. This defeats the purpose of the security mechanism and allows the attacker to both retrieve the cryptographic material as well as read out the proprietary code running in the secure world, the two scenarios that the defense was purpose-built to defeat.

Summary. We find that an attacker can access shared memory without holding the lock, as there is no relation between the lock and the memory. In our experiment, FreeRTOS stores a queue structure with a read and write pointer in this shared memory. Overwriting these pointers results in an arbitrary read primitive and an arbitrary write primitive, which lead to code execution on the victim CPU. If the victim firmware runs in a secure world, instead we gain a read and write primitive inside the secure world, which leads to code execution inside the secure world.

C. AV 3: Non-exclusive Peripheral Access

Here, we want to answer the following question:

Q4 Can we reconfigure a peripheral between configuration and usage?

Experimental Setup. We answer this question with one experiment. This experiment is based on an empty example. We add the configuration and usage of a single DMA peripheral to the victim firmware: the victim uses DMA to copy data from one place in memory to another place. The attacker firmware tries to modify the configuration between the configuration and enable points in time. The ground truth is trivially omitted: if the attacker does not modify the peripheral configuration, the peripheral acts as instructed.

Experiment 1. In this experiment, the attacker wants to modify the configuration at time t_0. Consider Figure 5. The victim firmware configures the DMA peripheral. Now, the attacker reconfigures the DMA peripheral under attack and in particular, reconfigures the source and destination address of the data transfer at time t_0. Then, the victim enables the peripheral. The peripheral performs its action. However, as the attacker modified the configuration, the action is under the attacker's control. Consequently, the DMA peripheral copies data from an attacker-controlled location to an attacker-controlled location (**Q4**). This results in an arbitrary read or write primitive, which an attacker can use to gain code execution on the victim CPU if executed multiple times, as shown in Section V-B. Note that the capabilities of this attack depend on the peripheral under attack. Attacking a DMA controller might result in code execution, while attacking an LED might not.

D. AV 4: Bus Maintainer Access Discrepancies

For this attack vector, we want to answer the question:

Q5 Can the attacker CPU use a bus maintainer to access a resource that it is physically not connected to?

Experimental Setup. We design two experiments to answer this question. Both experiments are based on an empty sample. The victim firmware sets a secret value in the memory region that only it can access, and the attacker firmware tries to retrieve this secret value. The NUCLEO-H755ZI-Q has 2 areas that only the Cortex-M7 can access: the data tightly coupled RAM (DTCM) and the instruction tightly coupled RAM (ITCM). We derive from the memory map that the attacker CPU has no other resource mapped at the address of the DTCM. The memory region that provides access to the ITCM is, however, mapped as an alias to the vector table instead. Our experiments test both areas.

Experiment 1. This first experiment serves as a ground truth. The attacker tries to directly access the secret value via its memory address. As expected, the access to the secret value in DTCM results in a hardfault. No resource is mapped here, thus the address cannot be resolved and the attacker cannot access the secret value.

The access to ITCM returns a value, but not the secret value. Instead, it matches the value that is at the same offset in flash memory. This is expected: the area provides an alias to the vector table, which is located at the start of the flash. Consequently, directly accessing it returns the value in flash at the given offset.

Experiment 2. In the second example, we use a bus maintainer peripheral that does have access to both memory regions and is available to the attacker CPU. In our scenario, we choose the MDMA controller. According to the reference manual, the MDMA controller is connected to both the attacker and victim CPU as well as both the DTCM and ITCM, to which the attacker CPU is not connected to [28]. In our experiment, we directly configure the controller with the malicious configuration to read the secret value and store it in a memory area that the attacker can access. As the attacker can execute arbitrary code on their CPU, the attacker can also reconfigure the MDMA controller with a malicious configuration. Our experiment represents the state after the attacker reconfigured the MDMA controller. As expected in Section IV, in both cases, the secret value is now accessible to the attacker in the configured attacker-accessible memory region. Indeed, the MDMA controller is connected to both the DTCM as well as the ITCM, unlike the attacker CPU. In addition, there is no remapping of the ITCM-mapped memory area. Consequently, the MDMA peripheral also accesses the correct secret value in the ITCM (**Q5**).

Summary. As expected in Section IV, we show that the attacker can access a single-connected resource that the attacker itself is not connected to, if the attacker is instead connected to another bus maintainer peripheral that *does* have access to the resource.

VI. Case Studies

In addition to practically evaluating the attack vectors on a development board, we demonstrate that our identified vulnerabilites are actually present in commercial products. For this, we use 2 commercial products: a proprietary networking firmware and a Samsung Galaxy Ring. The proprietary network firmware is taken from an *STM32WB55* [38], where it is used together with an application firmware. Each firmware runs on a dedicated CPU. The network CPU only runs ST-signed firmware, while the application CPU can run any.

The *Samsung Galaxy Ring* [30] is of particular interest as it processes health data on the application core, while the network core exposes this data via bluetooth.

A. STM32WB55 – Insecure Communication Channels

We chose this device because of two factors: first, according to Table II, this device is vulnerable to all attack vectors. Second, this device allows only signed and encrypted binaries on the network CPU. Consequently, production devices also employ the same network firmware as the development kit.

We investigated the communication mechanism between the network CPU and the application CPU and found a typical shared memory implementation that notifies the other processor via semaphores. The network processor manages the low-level Bluetooth handling, and relays higher-level Bluetooth events to the application processor. This event relaying is implemented with a queue mechanism: the network processor adds an element to the queue, and the application processor retrieves the queue element and subsequently appends it to a queue that manages free elements. Note that the pointers are not protected in any way. Indeed, the setup procedure expects the application processor to configure the shared memory region. Afterwards, the application processor wakes up the network processor, which parses the shared memory region data into the communication structures.

This presents a typical attack vector 2: changing the metadata in the shared memory communication structures may result in security issues. While a practical vulnerability confirmation was not possible due to the proprietary nature of the network core firmware, ST issued a patch that hardens this interface after we reported our attack vectors [33] in addition to the publication of the security advisory.

According to the ST application note 5156 “Introduction to security for STM32 MCUs” [29], “In dual-core products, one core can act as secure while the other is nonsecure”. However, it features a custom readout protection mechanism that prevents the application core from accessing some ram and flash regions. This does not impact the vulnerability against our proposed attack vectors, but practical attacks require more effort, as, e.g., code reuse attacks are not easily possible without knowing where certain instructions are located. Note that this is a typical case of security by obscurity.

B. Case study: Galaxy Ring – Zephyr Isolation Issues

The Samsung Galaxy Ring is a smart ring that measures several body functions and transmits the data to a smartphone via Bluetooth. It employs an nRF5340, which also advertises one CPU as a network core and the other CPU as application core. We obtained the firmware image of both the application processor as well as the network processor. Our analysis shows that both firmwares are based on the Zephyr RTOS [51]. The Zephyr RTOS advertises security as a core feature. Zephyr implements isolation on a thread level: User-Mode threads are untrusted, and thus isolated from Kernel-Mode threads and the kernel itself. Zephyr explicitly does not protect against kernel-level threats or threats from Kernel-Mode threads.

Zephyr implements wireless connectivity on the nRF5340 by implementing a mediator firmware. This mediator firmware interacts with closed-source, binary-only firmware provided by Nordic Semiconductor to access the wireless features of the hardware, and exposes these functionalities to the application core [50]. This allows the application firmware to make use of network functionality. The mediator firmware can communicate over various interfaces, such as an SPI interface, a UART interface or an IPC mechanism.

We analyze the network core application of the Samsung Galaxy Ring with binary similarity techniques and find it uses the sample IPC mechanism that Zephyr provides. In addition, we identify the thread start mechanism and find that all threads on the network core run in Kernel-Mode. Consequentially, also the network communication on the network core runs in Kernel-Mode.

Experimental confirmation. To confirm our theoretical findings on the Samsung Galaxy Ring, we design an equivalent experiment on an nRF5340 development board. In our experiment, Zephyr runs on both the network and the application core. As identified on the Samsung Galaxy Ring, we run the sample IPC mechanism from Zephyr on the network core, and we run the `peripheral` bluetooth sample on the application core. We modify the application core code to store secret data in a variable only accessible to the kernel and then execute a function pointer, also only accessible to the kernel.

Experiment 1. In this experiment, we show that an attacker on the network core can read data on the application core. To that end, we modify the network core application to read the secret data. This is equivalent to an attacker compromising the network core on the Galaxy Ring (as the Galaxy Ring seems to employ the sample IPC mechanism from Zephyr) and accessing sensitive health data that the application core collects. Indeed, the network core can access this data via attack vector 1. Zephyr protects this data via MPU, and as the application core and network core do not have the same MPU configuration, the attacker can access the secret data.

Experiment 2. In this experiment, the attacker on the network core hijacks the control flow of the application core. The network core modifies the function pointer executed by the application core, leading to attacker-controlled control flow. This is equivalent to an attacker compromising the network core on the Galaxy Ring and modifying the application core firmware, to, e.g., make sensors report wrong values. Similar to experiment 1, the attacker on the network core can modify the function pointer via attack vector 1, as Zephyr employs an MPU to protect this function pointer from untrusted access.

Note that we did not implement an exploit for the Samsung Galaxy Ring itself, as it requires malicious code execution in the network core. We consider obtaining code execution on the network core out of scope for this work.

We responsibly disclosed our findings to Samsung on August 5th 2025. We immediately got a (generic) confirmation of our message, but no further reply.

C. Summary

We have shown that the identified vulnerabilities are also relevant in practice. The proprietary network stack required for the STM32WB55 is vulnerable to attack vector 2, while attack vector 1 leads to the removal of security boundaries in the Samsung Galaxy Ring.

VII. Discussion

In this section, we discuss less powerful attacker models and possible countermeasures. In general, we divide countermeasures in two categories: software countermeasures and hardware countermeasures. Software countermeasures are retrospectively applicable, but can only help reduce risk and do not provide full protection. Hardware countermeasures *do* provide full protection, but require architectural redesign. Consequently, only devices manufactured after the architectural redesign are protected, leaving devices that were manufactured before that point in time vulnerable.

A. Attacks without Code Execution

Our attacker model assumes that an attacker can execute arbitrary code on one of the CPUs of the system. However, in many cases, code execution may not be required.

We find that all attack vectors can also work with only a write primitive (e.g. via a buggy peripheral configuration), if certain conditions are met. Changing the MPU configuration in attack vector 1 requires a write primitive with elevated permissions. Consequently, we can reduce the attacker capabilities to a privileged write primitive given the right circumstances. In attack vector 2, if the write primitive allows writing at the correct time (after sending data to the shared communication mechanism, but before the victim interacts with the mechanism), the attacker still obtains code execution on the victim CPU. Similarly, if the attacker has a write primitive that occurs after peripheral configuration but before peripheral enabling, attack vector 3 and 4 require no code execution.

B. Software Countermeasures

As mentioned above, software countermeasures cannot provide full protection, but can help reduce the risk. In the following, we suggest risk reduction strategies against the individual attack vectors.

Attack Vector 1. This attack vector introduces discrepancies in the MPUs of each CPU. However, each CPU can only access its own MPU configuration and cannot validate semantic equivalence of other CPUs' MPU configuration. Thus we see no software countermeasure for attack vector 1.

Attack Vector 2. One possible risk reduction strategy from the literature against attack vector 2 is to remove metadata in cross-CPU communication protocols [12]. Indeed, in our implementation of the attack vector, we exploit the queue structure that is used for communication. If, instead, the shared memory is fixed-size and data-only, the attack surface is reduced. The attacker can still modify the data in a TOCTOU-style attack, but the attacker is limited to writing data-only. If this data is not relevant to the control flow of the victim CPU, the attacker does not immediately gain code execution.

Another strategy is to copy the received data before use to a location only the victim has access to and operate on the copy. Combined with the first risk reduction strategy, this does invalidate attack vector 2.

Attack Vector 3. The best option we see to reduce the impact of attack vector 3 is to reduce the time between peripheral configuration and peripheral usage. This reduces the probability of the attacker to reconfigure the peripheral at the correct point in time. However, this strategy suffers from two problems. First, the attacker may be able to introduce artificial delays, e.g., via interrupts. If the attacker can trigger an interrupt on the victim CPU, this will invoke the interrupt service routine (ISR), suspending regular firmware execution.

The attacker can use this delay as an additional time window to reconfigure the peripheral. Furthermore, a peripheral may be long-running. In our example in Section V-C, we demonstrated the attack vector at the example of a DMA transaction. This transaction is configured once and runs once. Other peripherals may be used repeatedly with the same configuration. Consider an example of a UART console that runs indefinitely. The firmware configures the UART peripheral once, and subsequently, a user can interact with the console. The peripheral is not reconfigured after each console interaction but instead is configured once at the start of the operation. Consequently, timing is not an issue for an attacker in this scenario.

Attack Vector 4. As attack vector 4 relies entirely on the architecture of the MCU, we see no software countermeasure.

C. Hardware countermeasures

To mitigate the presented attack vectors, we instead suggest the implementation of additional hardware features. While this provides the most comprehensive defense against our attack vectors, additional hardware cannot be retrofit and thus only future chips can benefit. Existing chips are left vulnerable.

Attack Vector 2. To mitigate attack vector 2, the manufacturer needs to implement a hardware-enforced communication channel that requires the attacker to commit data to the hardware (immutable afterwards). The RP2040 [48] can serve as a reference: it implements a read-only read register and a write-only write register in each CPU. Writing to the write register passes the data to the read register of the other CPU. Thus, the attacker can no longer modify the data.

Attack Vectors 1,3 and 4. We recommend to fix the attack vectors 1, 3 and 4 with a bus-level permission management system. This system checks (before the access to a memory or peripheral happens) if the bus maintainer has the correct permissions to access the resource. One example implementation can be found in the investigated Infineon boards [19], [21], [20]. Infineon implements a peripheral protection unit, which is similar to an MPU but dedicated to protect peripheral accesses and tailored towards peripheral protection, such as more protection areas. Infineon implements a hardware feature called shared MPU. This acts as an MPU, but it is shared across all bus maintainers. This combination of hardware features defends against attack vectors 1, 3 and 4. The peripheral protection unit can protect the peripherals that an attacker manipulates, defending against attack vector 3 and 4, and the shared MPU protects against attack vector 1. The shared MPU protects the shared MPU itself, defending against attack vector 4, and defends memory areas that other bus maintainers are not supposed to access (attack vector 3).

VIII. Related Work

There is no prior work that considers the multi-CPU MCU scenario on embedded devices. A systematization lists no investigation in this domain [60]. The closest work is from Classen et al., which demonstrates cross-CPU exploitation for a specific MCU with dedicated WiFi and Bluetooth CPUs [12].

DMA Attacks. In early work on Firewire [10], Boileau demonstrated that flaws in the Firewire protocol can allow an external peripheral (connected via Firewire) to reconfigure DMA controllers to gain direct access to the entire addressable system memory. On commodity PCs, many defenses have been proposed [64], [65], [67]. On embedded devices, DMA attacks are still more prevalent. For example, the use of DMA-related sidechannels to exploit trusted execution architectures in embedded systems was also investigated in [9]. In particular, the gap between formal modeling-based security guarantees and practically achievable guarantees was examined, and novel DMA-based side-channel attacks are possible.

Embedded Exploit Mitigation. Many exploit mitigation techniques have been proposed in recent years. Abbasi et al. propose μArmor, an MPU-based defense for RTOS. They implement a protection mechanism for executable memory regions, a code reuse gadget reduction scheme and a data/pointer separation scheme [1]. The use of MPUs as access control mechanism was also investigated by [14]. In this work, the authors create overlays over regions and assign privileges. An instruction that accesses such an overlay is replaced with an ISR that checks access permissions first. Clements et al. propose ACES, a compiler extension that uses source code and a developer-specified security policy to enforce compartmentalization at runtime [13]. NesCheck, on the other hand, does not require a security policy. Instead, NesCheck leverages static analysis and runtime checks to enforce memory safety [39]. More recently, Mera et al. found that existing compartmentalization schemes do not consider DMA. They propose D-Box, a scheme that also secures against malicious DMA [27]. In addition, Zhou et al. point out that an MPU cannot restrict the maintainer access of peripherals [66].

Trusted Execution Environments. Another line of research aims to protect embedded devices with TEEs. Wang et al. propose RT-Tee, a TEE that can fulfill real-time CPU and MMIO requirements [61]. The authors of [59] use a TEE to measure the integrity of an embedded device. This employs a TEE to measure a novel metric, the operation execution integrity, a combination of control flow and critical data integrity, and attest it remotely. However, recent work has pointed out limitations of TEEs in embedded devices. Rodrigues et al. discovered a sidechannel in the MCU bus interconnect logic. The authors use this side channel to leak a secret from the TEE [52]. In [26], the authors abuse physical features of SRAM to retrieve secrets from TEEs. SRAM exposes the value it holds while under excessive stress. This allows the authors to extract secrets across protection domains.

IX. Conclusions

In this work, we systematically identified security issues in powerful next-generation embedded systems, resulting from the integration of multiple CPUs in a single SoC. We found that security mechanisms that are present and effective on single-CPU systems do not transfer to the multi-CPU setting. As a result, attackers can read and write protected data from other CPUs on the same system.

We identified four attack vectors, and found that a significant number of systems on the market are vulnerable. These attack vectors can lead to arbitrary read and write primitives, such that one CPU can force the other CPU to read and write memory locations on its behalf, and even to code execution. In addition, we found that the communication mechanism of a popular open source RTOS may introduce code execution vulnerabilities in the multi-CPU scenario. We verified our theoretical predictions by implementing our four attacks and demonstrated their practical efficacy. Our attacks allowed us to overcome proprietary TEE implementations that were supposed to defend against reading out data on one of the CPUs. We performed two case studies, showing the practical impact of our attack vectors, and discussed potential software and hardware countermeasures. We responsibly disclosed our findings to the vendors, resulting in one security advisory and updates to proprietary network stacks of one vendor.

Acknowledgments

We thank our anonymous reviewers for their valuable feedback. This work was funded by the Federal Ministry of Research, Technology and Space (BMFTR) under the grant ASRIOT (16KIS1901). The responsibility for the content of this publication lies with the authors.

X. Ethics Considerations

Our paper introduces 4 novel attack vectors that can only partially be mitigated via software. As a first step, we provide instructions how to minimize the attack surface via software, even in scenarios where software updates alone do not suffice.

Furthermore, we responsibly disclosed our findings to the device manufacturers as well as the FreeRTOS team shortly after finishing the experiments and provided assistance in judging and mitigating the impact.

In addition, we hope to use this paper as a communication tool to reach hardware manufacturers that consider multi-CPU designs in their products. For some of the presented vulnerabilities, only hardware changes fully mitigate the underlying issue. By raising awareness about the presented issues, we hope that hardware manufacturers consider hardware countermeasures as already implemented by some of the manufacturers.

References

[1] Ali Abbasi, Jos Wetzels, Thorsten Holz, and Sandro Etalle. Challenges in designing exploit mitigations for deeply embedded systems. In *Proceedings of the IEEE European Symposium on Security and Privacy (EuroSP)*, 2019.

[2] ARM. AMBA® AHB protocol specification. https://documentation-service.arm.com/static/6141bf0d674a052ae36ca811. Accessed: 2024-09-02.

[3] ARM. AMBA® APB protocol specification. https://documentation-service.arm.com/static/63fe2c1356ea36189d4e79f3. Accessed: 2024-09-02.

[4] ARM. AMBA® AXI protocol specification. https://documentation-service.arm.com/static/63ff0ebd56ea36189d4e7ee7. Accessed: 2024-09-02.

[5] ARM. Learn the architecture: A profile. https://www.arm.com/architecture/learn-the-architecture/a-profile. Accessed: 2024-08-14.

[6] ARM. Learn the architecture: M profile. https://www.arm.com/architecture/learn-the-architecture/m-profile. Accessed: 2024-08-14.

[7] ARM. Trustzone for Cortex-M. https://www.arm.com/technologies/trustzone-for-cortex-m. Accessed: 2024-09-04.

[8] Aspencore. Embedded markets study: Integrating iot and advanced technology designs, application development & processing environments. https://www.embedded.com/wp-content/uploads/2019/11/EETimes_Embedded_2019_Embedded_Markets_Study.pdf, 2019. Accessed: 2024-08-14.

[9] Marton Bognar, Jo Van Bulck, and Frank Piessens. Mind the gap: Studying the insecurity of provably secure embedded trusted execution architectures. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022.

[10] Adam Boileau. Hit by a bus: Physical access attacks with Firewire. Presentation at Ruxcon, 2006.

[11] Stephen Checkoway, Damon McCoy, Brian Kantor, Danny Anderson, Hovav Shacham, Stefan Savage, Karl Koscher, Alexei Czeskis, Franziska Roesner, and Tadayoshi Kohno. Comprehensive experimental analyses of automotive attack surfaces. In *Proceedings of the USENIX Security Symposium*, 2011.

[12] Jiska Classen, Francesco Gringoli, Michael Hermann, and Matthias Hollick. Attacks on wireless coexistence: Exploiting cross-technology performance features for inter-chip privilege escalation. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022.

[13] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer. {ACES}: Automatic compartments for embedded systems. In *Proceedings of the USENIX Security Symposium*, 2018.

[14] Abraham A Clements, Naif Saleh Almakhdhub, Khaled S Saab, Prashast Srivastava, Jinkyu Koo, Saurabh Bagchi, and Mathias Payer. Protecting bare-metal embedded systems with privilege overlays. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2017.

[15] Andrei Costin, Jonas Zaddach, Aurélien Francillon, and Davide Balzarotti. A {Large-scale} analysis of the security of embedded firmwares. In *Proceedings of the USENIX Security Symposium*, 2014.

[16] Max Eisele, Daniel Ebert, Christopher Huth, and Andreas Zeller. Fuzzing embedded systems using debug interfaces. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2023.

[17] FreeRTOS. FreeRTOS: Real-time operating system for microcontrollers and small microprocessors. https://www.freertos.org/. Accessed: 2024-08-24.

[18] FreeRTOS. FreeRTOS stream & message buffers. https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/04-Stream-and-message-buffers/03-Message-buffer-example. Accessed: 2024-08-22.

[19] Infineon. 32-bit PSoC 6 Arm Cortex-M4 / m0+. https://www.infineon.com/cms/en/product/microcontroller/32-bit-psoc-arm-cortex-microcontroller/psoc-6-32-bit-arm-cortex-m4-mcu/. Accessed: 2024-08-14.

[20] Infineon. 32-bit traveo t2g arm cortex microcontroller. https://www.infineon.com/cms/en/product/microcontroller/32-bit-traveo-t2g-arm-cortex-microcontroller/. Accessed: 2024-08-14.

[21] Infineon. 32-bit xmc7000 industrial microcontroller arm cortex-m7. https://www.infineon.com/cms/en/product/microcontroller/32-bit-industrial-microcontroller-based-on-arm-cortex-m/32-bit-xmc7000-industrial-microcontroller-arm-cortex-m7/. Accessed: 2024-08-14.

[22] Infineon. An228571 getting started with psoc™ 6 mcu on modustoolbox™ software. https://documentation.infineon.com/psoc6/docs/pvw1667470959605#revision-history. Accessed: 2025-11-06.

[23] Infineon. Infineon launches xmc7000 series for industrial applications with increased performance, memory, advanced peripherals, and extended temperature range. https://www.infineon.com/market-news/2022/infcss202211-021. Accessed: 2025-11-06.

[24] Infineon. Infineon second quarter fy 2022 quarterly update. https://www.infineon.com/dgdl/2022-05-09+Q2+FY22+Investor+Presentation.pdf?fileId=8ac78c8b808544e20180a4d32bb70009. Accessed: 2024-08-14.

[25] Intel. Intel® software guard extensions (intel® SGX). https://www.intel.com/content/www/us/en/products/docs/accelerator-engines/software-guard-extensions.html. Accessed: 2024-09-04.

[26] Jubayer Mahmod and Matthew Hicks. Untrustzone: Systematic accelerated aging to expose on-chip secrets. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2024.

[27] Alejandro Mera, Yi Hui Chen, Ruimin Sun, Engin Kirda, and Long Lu. D-box: Dma-enabled compartmentalization for embedded applications. *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2022.

[28] ST Microelectronics. High-performance and dsp with dp-fpu, arm cortex-m7 + cortex-m4 mcu with 2mbytes of flash memory, 1mb ram, 480 mhz cpu, art accelerator, l1 cache, external memory interface, large set of peripherals including a crypto accelerator, smps. https://www.st.com/en/microcontrollers-microprocessors/stm32h755zi.html. Accessed: 2024-08-22.

[29] ST Microelectronics. Introduction to security for stm32 mcus. https://www.st.com/content/ccc/resource/technical/document/application_note/group1/9f/0b/e4/b6/75/15/4f/e2/DM00493651/files/DM00493651.pdf/jcr:content/translations/en.DM00493651.pdf. Accessed: 2025-07-25.

[30] ST Microelectronics. Potential isolation issue between cpu1 and cpu2 on stm32wb5x, stm32wb3x, stm32wb1x, and stm32wl5x. https://www.st.com/resource/en/security_advisory/sa0024-potential-isolation-issue-between-cpu1-and-cpu2-on-stm32wb5x-stm32wb3x-stm32wb1x-and-stm32wl5x-stmicroelectronics.pdf. Accessed: 2025-07-25.

[31] ST Microelectronics. Proprietary code read-out protection (pcrop) software expansion for stm32cube (an4701, an4758 and an4968). https://www.st.com/en/embedded-software/x-cube-pcrop.html. Accessed: 2024-08-28.

[32] ST Microelectronics. Stm32 wireless: First mcus now available, first nucleo pack with usb dongle. https://blog.st.com/stm32wb55-stm32cubemonrf-p-nucleo-wb55/. Accessed: 2025-11-06.

[33] ST Microelectronics. Stm32cubewb mcu firmware package wireless coprocessor release notes. https://github.com/STMicroelectronics/STM32CubeWB/blob/dcc538339a30165ced95745969706b7423e3d96d/Projects/STM32WB_Copro_Wireless_Binaries/STM32WB5x/Release_Notes.html#L568. Accessed: 2025-07-25.

[34] ST Microelectronics. Stm32h7: First dual core version, more accessible single core models. https://blog.st.com/dual-core-stm32h7/. Accessed: 2024-09-02.

[35] ST Microelectronics. Stm32h7: First dual core version, more accessible single core models. https://blog.st.com/dual-core-stm32h7/. Accessed: 2025-11-06.

[36] ST Microelectronics. Stm32wl, the 1st mcu with embedded lora transceiver, a masterclass in chip design. https://blog.st.com/stm32wl/. Accessed: 2025-11-06.

[37] ST Microelectronics. Stm32wl5x. https://www.st.com/en/microcontrollers-microprocessors/stm32wl5x.html. Accessed: 2024-08-14.

[38] ST Microelectronics. Ultra-low-power dual core arm cortex-m4 mcu 64 mhz, cortex-m0+ 32 mhz with 1 mbyte of flash memory, bluetooth le 5.4, 802.15.4, zigbee, thread, matter, usb, lcd, aes-256. https://www.st.com/en/microcontrollers-microprocessors/stm32wb55rg.html. Accessed: 2024-08-14.

[39] Daniele Midi, Mathias Payer, and Elisa Bertino. Memory safety for embedded devices with nescheck. In *Proceedings of the ACM Asia Conference on Computer and Communications Security*, 2017.

[40] Marius Muench, Dario Nisi, Aurélien Francillon, and Davide Balzarotti. Avatar 2: A multi-target orchestration platform. In *Workshop on Binary Analysis Research*, 2018.

[41] NXP. Lpc4300 series: High-performance microcontrollers (mcus) based on arm® cortex®-m4/m0 cores. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/lpc4300-arm-cortex-m4-m0:MC_1403790133078#/. Accessed: 2024-08-14.

[42] NXP. Multi-core microprocessors in embedded applications. https://www.nxp.com/docs/en/white-paper/multicoreWP.pdf. Accessed: 2024-09-02.

[43] NXP. Nxp releases first k32 l microcontrollers to production. https://www.nxp.jp/company/about-nxp/newsroom/NW-RELEASE-K32L. Accessed: 2025-11-06.

[44] NXP. Nxp's energy efficient cortex-m4 mcu with cortex-m0+ and advanced security. https://www.nxp.com/products/processors-and-microcontrollers/arm-microcontrollers/general-purpose-mcus/k32-l-series-arm-cortex-m4-m0-plus/nxps-energy-efficient-cortex-m4-mcu-with-cortex-m0-plus-and-advanced-security:K32-L3. Accessed: 2024-08-14.

[45] NXP. Product longevity. https://www.nxp.com/products/nxp-product-information/nxp-product-programs/product-longevity:PRDCT_LONGEVITY_HM. Accessed: 2025-11-06.

[46] Raspberry Pi. Meet raspberry silicon: Raspberry pi pico now on sale at $4. https://www.raspberrypi.com/news/raspberry-pi-silicon-pico-now-on-sale/. Accessed: 2025-11-06.

[47] Raspberry Pi. Raspberry pi pico 2, our new $5 microcontroller board, on sale now. https://www.raspberrypi.com/news/raspberry-pi-pico-2-our-new-5-microcontroller-board-on-sale-now/. Accessed: 2025-11-06.

[48] Raspberry Pi. rp2040. https://github.com/raspberrypi/documentation/blob/develop/documentation/asciidoc/microcontrollers/silicon/rp2040.adoc. Accessed: 2024-08-14.

[49] Raspberry Pi. Rp2350 reference manual. https://datasheets.raspberrypi.com/rp2350/rp2350-datasheet.pdf. Accessed: 2025-08-04.

[50] Zephyr Project. Zephyr github. https://github.com/zephyrproject-rtos/zephyr/tree/main/samples/bluetooth/hci_ipc. Accessed: 2025-07-31.

[51] Zephyr Project. Zephyr project. https://www.zephyrproject.org/. Accessed: 2025-07-25.

[52] Cristiano Rodrigues, Daniel Oliveira, and Sandro Pinto. BUSted!!! Microarchitectural side-channel attacks on the MCU bus interconnect. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2023.

[53] Tobias Scharnowski, Nils Bars, Moritz Schloegel, Eric Gustafson, Marius Muench, Giovanni Vigna, Christopher Kruegel, Thorsten Holz, and Ali Abbasi. Fuzzware: Using precise {MMIO} modeling for effective firmware fuzzing. In *Proceedings of the USENIX Security Symposium*, 2022.

[54] Cypress Semiconductor. Cypress traveo 32-bit arm automotive microcontrollers (mcus). https://web.archive.org/web/20170316181430/http://www.cypress.com/products/cypress-traveo-32-bit-arm-automotive-microcontrollers-mcus. Accessed: 2025-11-06.

[55] Nordic Semiconductor. https://blog.nordicsemi.com/getconnected/why-does-the-nrf5340-have-two-cores. https://blog.nordicsemi.com/getconnected/why-does-the-nrf5340-have-two-cores. Accessed: 2024-12-05.

[56] Nordic Semiconductor. Meet the nrf5340, nordic's new dual-core flagship soc. https://blog.nordicsemi.com/getconnected/meet-the-nrf5340-nordics-new-dual-core-flagship-soc. Accessed: 2025-11-06.

[57] Nordic Semiconductor. nrf5340. https://www.nordicsemi.com/Products/nRF5340. Accessed: 2024-08-14.

[58] ST. Stm32cube initialization code generator. https://www.st.com/en/development-tools/stm32cubemx.html. Accessed: 2024-08-22.

[59] Zhichuang Sun, Bo Feng, Long Lu, and Somesh Jha. Oat: Attesting operation integrity of embedded devices. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2020.

[60] Xi Tan, Zheyuan Ma, Sandro Pinto, Le Guan, Ning Zhang, Jun Xu, Zhiqiang Lin, Hongxin Hu, and Ziming Zhao. SoK:Where's the "up"?! a comprehensive (bottom-up) study on the security of arm Cortex-M systems. In *Proceedings of USENIX WOOT Conference on Offensive Technologies (WOOT)*, 2024.

[61] Jinwen Wang, Ao Li, Haoran Li, Chenyang Lu, and Ning Zhang. Rt-tee: Real-time system availability for cyber-physical systems using arm trustzone. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2022.

[62] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2020.

[63] Ashley Whittaker. Raspberry pi pico – what did you think? https://www.raspberrypi.com/news/raspberry-pi-pico-what-did-you-think/. Accessed: 2024-08-22.

[64] Paul Willmann, Scott Rixner, and Alan L Cox. Protection strategies for direct access to virtualized I/O devices. In *Proceedings of USENIX Annual Technical Conference (USENIX ATC)*, 2008.

[65] Miao Yu. *An I/O Separation Model and its Applications to On-Demand I/O on Commodity Platforms*. PhD thesis, Carnegie Mellon University, 2019.

[66] Wei Zhou, Zhouqi Jiang, and Le Guan. Understanding mpu usage in microcontroller-based systems in the wild. In *Workshop on Binary Analysis Research*, 2023.

[67] Zongwei Zhou, Virgil D Gligor, James Newsome, and Jonathan M McCune. Building verifiable trusted path on commodity x86 computers. In *Proceedings of the IEEE Symposium on Security and Privacy (SP)*, 2012.

XI. Artifact Appendix

A. Description & Requirements

This artifact currently contains sample attacks for each of the 4 vectors, and experiments on a setup similar to the setup of the Galaxy Ring. Detailed list of experiments:

a) Experiments on attack vector 1: We have two experiments on attack vector 1 (MPU policy desynchronization). In the experiment, the M7 configures a memory area as not readable by anyone. In Experiment 1, the M7 then tries to read from the memory area. In Experiment 2, the M4 tries to read from the memory area.

b) Experiments on attack vector 2: We have three experiments on attack vector 2 (unsynchronized communication channels). In the first experiment, the attacker (on the M4) tries to gain a read privilege. We have two experiments where the attacker tries to execute code on the M7: in the first experiment, the attacker respects the hardware synchronization primitive, in the second experiment, the attacker does not respect it.

c) Experiment on attack vector 3: In this experiment, the attacker on the M4 will reconfigure a peripheral after the victim configured it, leading to data leakage.

d) Experiments on attack vector 4: In these experiments, the attacker on the M4 wants to leak data from memory that is not accessible to the M4. In the first experiment, the attacker directly tries to access the data, and in the second experiment, the attacker uses a peripheral as confused deputy peripheral.

e) Experiment on Galaxy Ring scenario: In these experiments, the attacker on the network core accesses data on the application core and also change data on the application core.

f) Requirements: If you want to recreate the environment from scratch, you need to setup the ST Cube IDE [1].and the zephyr build environment [2]. You will also need access to a Nucleo-H755 and a nRF5340 development board.

1) How to access: The artifact archive is at https://doi.org/10.5281/zenodo.17524720. You can also find it on Github: https://github.com/smnhff-work/multicpu-pitfalls/

2) Hardware dependencies: ST Microelectronics Nucleo H755 development kit[3]None if you use the provided machine.

3) Software dependencies: STM32 Cube IDE[1] and an ARM toolchain[4] to build the ST samples.

4) Benchmarks: None.

B. Artifact Installation & Configuration

1) ST setup: Install the ST Cube IDE and the zephyr environment as mentioned above. For the ST samples, import the provided archive, and build the samples. For the zephyr samples, exchange the files as described in the readme.

C. Experiment Workflow

See the example in XI-E1.

[1]https://www.st.com/en/development-tools/stm32cubeide.html

[2]https://docs.zephyrproject.org/latest/index.html

[3]https://www.st.com/en/evaluation-tools/nucleo-h755zi-q.html

[4]https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads

D. Major Claims

- (C1) The **MPU Policy Desynchronization** attack allows an attacker to access protected memory from a different CPU.
- (C2) The **Unsynchronized Communication Channels** attack allows an attacker to read and write arbitrary memory locations on a different CPU.
- (C3) The **Non-exclusive Peripheral Access** attack allows an attacker to reconfigure a peripheral after it has been configured by the other CPU.
- (C4) The **Confused Deputy Peripheral** allows an attacker to access otherwise unavailable memory.
- (C5) We can use the **MPU Policy Desynchronization** attack on Zephyr to write to and read from the application core kernel memory from the network core.

E. Evaluation

In the general structure of the experiments, we store a secret value at a specific memory address. On the attacker CPU, we perform the attack and afterwards check that the received data matches the expected secret. If this holds, we print `<Experiment number> worked`. We note that we use Cortex M4, M4 and attacker interchangeably and Cortex M7, M7 and victim interchangeably.

1) Experiment E1+E2 (C1): [MPU Policy Synchronization] [15 human-minutes + 5 compute-minutes]: This experiment uses the memory protection unit on the Cortex M7 to remove read access to a specific memory area. In this experiment, we show that the Cortex M4 does not need to respect this policy. This experiment does two experiments at the same time: show the baseline, that is the M7 can no longer access the protected memory (E1), and show that the M4 can still access the protected memory (E2).

[How to] You flash the experiment and observe the output via a uart connection.

[Execution]

1. Click on the arrow of the **empty-h755-mpu** project. This opens the folder structure.
2. **Right click** on the **empty-h755-mpu_CM7** project, then run **Clean Project**. Afterwards, run **Build Project**.
3. Repeat 2. for the **empty-h755-mpu_CM4** project.
4. **Right click** on the **empty-h755-mpu_CM7** project, select **Debug As** → **Debug Configurations...**.
5. On the left hand side, you see numerous debug configurations. Select **empty-h755-mpu_CM7**, then press **Debug**. This downloads both the M7 and the M4 firmware to their respective CPU and starts a debugging session for the M7. It breaks on the first line in the M7 main function, where the M4 has not started yet.
6. Press **Resume** in the toolbar at the top of the window.

[Results] Inspect the UART output. To that end, figure out the port where the device is connected and connect to it, e.g., via `minicom`. Two things are required for successful reproduction: the absence of `Direct access worked!` and the presence of `E2 worked!`. If `Direct access worked` is

present, then the M7 could access the restricted memory, disproving the ground truth and thus disproving E1. `E2 worked` is only printed if the M4 could access the memory and found the expected values. Consequently, this string is required to show that E2 worked.

2) Experiment E3 (C2): [Unsychronized Communication Channels - Data Leakage] [15 human-minutes + 5 compute-minutes]: This experiment tricks the M7 to access and provide data on the behalf of the M4.

[How to, Preparation and Execution] See E1+E2, but use the `Communication-DataLeak` folder.

[Results] Inspect the UART output (the console window on the right) and verify that `E3 worked` is present. If `E3 worked` is printed, the M4 successfully modified the shared communication channel to leak the secret.

3) Experiment E4+E5 (C2): [Unsychronized Communication Channels - Code Execution] [15 human-minutes + 5 compute-minutes]: This experiment tricks the M7 into overwriting a return address and thus jumping to an attacker-controlled code location. E4 assumes that the attacker respects the synchronization primitive, showing that the attack is not possible if synchronization on the communication primitive is mandatory. E5 shows that the attacker does not need to respect the primitive, leading to code execution. By default, E4 is executed.

[How to, Preparation and Execution] See E1+E2, but use the `Communication-CodeExec` folder. To switch between E4 and E5, comment line 35 in `main.c` in the M4 source tree (under `Comm...-CodeExec_CM4/Core/Src/main.c`) and rerun steps 1-6. Check Results E5 at this point.

[Results E4] Inspect the UART output (the console window on the right) and verify that `E5 worked` is **not** present. If the attacker respected synchronization, then it could not take the lock and thus only modify metadata after the victim was done using the shared struct.

[Results E5] If `E5 worked` is printed, the attacker successfully modified the shared communication channel while the victim was using it. The attacker changed the victim's control flow to execute a function that prints the required string.

4) Experiment E6 (C3): [Non-exclusive peripheral access] [15 human-minutes + 5 compute-minutes]: In this experiment, the attacker modifies the configuration of a peripheral that the victim uses after the victim already setup the peripheral.

[How to, Preparation and Execution] See E1+E2, but use the `empty-h755-peripherals` folder.

[Results] Inspect the UART output (the console window on the right) and verify that `E6 worked` is present. If `E6 worked` is printed, the M4 successfully modified the peripheral configuration after the victim set it up. This results in different data being copied.

5) Experiment E7+E8 (C4): [Bus Maintainer Access Discrepancies] [15 human-minutes + 5 compute-minutes]: In this experiment, the attacker uses a confused deputy peripheral to access memory that it is physically not connected to.

[How to, Preparation and Execution] See E1+E2, but use the `empty-h755-dma` folder. To switch between E7 and E8, comment line 30 in `main.c` in the M4 source tree (under `empty-h755-dma_CM4/Core/Src/main.c`) and rerun steps 1-6. Check Results E8 at this point.

[Results E7] Inspect the UART output and verify that `E8 worked` is **not** present. Indeed, the attacker tries to access the memory directly. This memory is not mapped and thus the access results in a fault.

[Results E8] Inspect the UART output and verify that `E8 worked` is present. In this iteration, the attacker uses a peripheral to access the memory. This peripheral is connected to the memory and can thus access and copy it. If the value matches the expected value, the M4 code prints the required string.

6) Experiment E7+E8 (C4): [Zephyr MPU Policy Desynchronization][15 human-minutes + 5 compute-minutes]: In this experiment, the attacker uses an MPU Policy Desynchronization to read from or write to application core kernel memory from the network core. *[How to, Preparation, Execution]* Flash the experiment (network core firmware and application core firmware) to the board, and observe the uart output. *[Results E7]* Inspect the console output on the bottom left console. If it contains the string `secret data: secret`, the attack worked. In this experiment, the attacker tried to directly access the memory from the victim. We use the hardcoded string `secret` as secret. *[Results E8]* Inspect the console output on the bottom right console. If it contains the string `The attack worked!`, then the attack worked. In this experiment, the attacker overwrites a function pointer in victim memory. If this worked, the victim will execute a function that prints `The attack worked!`.