HAW HAMBURG

BACHELOR THESIS

Bennet Blischke

Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices

Faculty of Engineering and Computer Science
Department Computer Science

HOCHSCHULE FÜR ANGEWANDTE WISSENSCHAFTEN HAMBURG
Hamburg University of Applied Sciences

Bennet Blischke

Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices

Bachelor thesis submitted for examination in Bachelor´s degree
in the study course *Bachelor of Science Informatik Technischer Systeme*
at the Department Computer Science
at the Faculty of Engineering and Computer Science
at University of Applied Science Hamburg

Supervisor: Prof. Dr. Thomas Schmidt
Supervisor: Prof. Dr. Franz Korf

Submitted on: 14. Dezember 2023

Bennet Blischke

Thema der Arbeit

Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices

Stichworte

RISC-V, PMP, RIOT, Speicherschutz, Sicherheit, eingebettete Systeme

Kurzzusammenfassung

RISC-V ist eine aufkommende Befehlssatzarchitektur, die bereits in eingeschränkten Geräten verwendet wird. Da diese Geräte heute mehr denn je miteinander verbunden sind, ist die Sicherheit des Betriebssystems von grundlegender Bedeutung. Die RISC-V Physical Memory Protection (PMP)-Einheit bietet Hardwareunterstützung für Speicherschutzmechanismen auf Betriebssystemebene. Können eingebettete Betriebssysteme die PMP nutzen, um zusätzliche Sicherheitsmechanismen zu implementieren? Stellt die begrenzte Verfügbarkeit von Rechenressourcen ein Hindernis dar? Um dies zu untersuchen, habe ich einen Prototyp für das IoT-Betriebssystem RIOT entwickelt. Da RIOT quelloffen ist, konnte ich Teile dieses Prototyps in den Quellcode einbringen. Ich habe in diesem Prototyp die Verhinderung der Datenausführung und die Erkennung von Thread-Stack-Überläufen implementiert. Ich habe festgestellt, dass die RISC-V PMP für eingeschränkte Geräte geeignet ist, aber die Integration in bestehende Software-Stacks eine Herausforderung darstellt. Daher sollten bei der Entwicklung von eingebetteter Software hardwarebasierte Sicherheitsverfahren in Betracht gezogen werden.

Bennet Blischke

Title of Thesis

Evaluation of RISC-V Physical Memory Protection in Constrained IoT Devices

Keywords

RISC-V, PMP, RIOT, Memory protection, Security, Constrained devices

Abstract

RISC-V is an emerging instruction set architecture that is already in use in constrained devices. As these devices are now more interconnected than ever before, the need for operating system (OS) security is fundamental. The RISC-V physical memory protection (PMP) unit offers hardware assistance for memory protection schemes at an OS level. Can embedded OSes utilize the PMP to implement additional security schemes? Does the limited availability of computation resources present an obstruction? To investigate, I built a prototype for the IoT operating system RIOT. Since RIOT is open source, I was able to contribute parts of this prototype into its source tree. I implemented data execution prevention and thread stack overflow detection within this prototype. I found that the RISC-V PMP is suitable for constrained devices, but the integration into existing software stacks is challenging. Consequently, hardware-based security schemes should be considered during the design of embedded software.

Contents

List of Figures

List of Tables

Listings

1 Introduction

Microcontroller units continue to gain more power and sophistication as advanced features and peripherals are added, while maintaining a similar silicon size [3, p. 100]. With the integration of the memory management unit (MMU), important security features such as privileged execution and memory protection are now available on some of these systems. Employing memory protection schemes for tasks, such as detecting software faults and stack overflows, is vital to ensure the security of connected and constrained devices.

1.1 Motivation

Listing 1.1: Example code that contains a possible buffer overflow. The function concatenates two strings "hello: " and a provide name, e.g. "Bob" to obtain "hello: Bob". The buffer is then printed to the screen. Since the length of the buffer is fix and the length of the name is not checked, a very long name will overflow the buffer during memory copy.

```
void print_name(char * name) {
    const char * HELLO = "hello: ";
    char buffer[100];
    memcpy(buffer, HELLO, strlen(HELLO));
    memcpy(buffer+strlen(HELLO), name, strlen(name));
    puts(buffer);
}
```

To this day, memory corruption is the most prominent used type of exploitation [4]. A significant reason is the use of memory-unsafe languages with manual memory management, such as C or C++ [5, p. 51]. Human error is unavoidable, therefore, it is unsurprising that a majority of vulnerabilities are caused by memory unsafety. E.g. for 2021, the Google

Project Zero [6] concluded that attackers have no need for new exploitation methods as the common memory corruption-based exploits are still sufficient - even though we have known about these attack surfaces, like buffer overflows, for over 55 years [7]. Further, Project Zero showed that 67% of the vulnerabilities they found are memory-corruption-based.

Shown in Listing 1.1 is a typical C code that contains a buffer overflow. In C it is the task of the developer to check bounds of buffers. Accessing (reading or writing) out of bounds memory is undefined behaviour in C [5, p. 51]. Here a required length check of the input parameter was overlooked, making it possible to overflow the intermediate buffer by providing a very long input name to the function.

Modern programming languages like Rust [8] provide certain memory safety guarantees, which makes it very hard to recreate traditional buffer overflow based vulnerabilities [4, Sec. IV]. Using these memory-safe languages is considered as a strong step forward towards more secure systems [5, p. 48].

Since RIOT is mostly written in C, memory-corruption is a major concern. Therefore it is important to deploy security mechanisms to reduce the surface and impact of such defects.

1.2 Thesis Objective

In this thesis, the RISC-V physical memory protection is used to deploy memory protection schemes on constrained devices. Due to the limitations of constrained devices and the new but scarce availability of the PMP, the feasibility of this deployment on top of an embedded operating system is unclear. This thesis provides a first insight into this feasibility.

For this purpose, a case study is made by implementing a driver for the RISC-V PMP within the RIOT operating system. The implementation of various memory protection schemes are explored using this driver and their effectiveness is evaluated. This thesis focuses on the 32-Bit RISC-V instruction set architecture (ISA) RV32 only.

1.3 Research Questions

`Data execution prevention` and `thread stack overflow detection` are security mechanisms capable of reducing the impact of certain memory-corruption defects. This leads to the following research questions in the context of constrained devices:

1. Can data execution prevention be implemented with RISC-V PMP? Which challenges need to be overcome and what kind of overhead remains?
2. Can stack overflow detection be implemented with RISC-V PMP? Which challenges need to be overcome and what kind of overhead remains?

1.4 Outline

First, required background information will be introduced in chapter 2. This encompasses specifying what is meant by 'Constrained Devices' and offering a broad outline of RISC-V. The relevant memory protection schemes are introduced as well. In chapter 3, an overview of related work is given. The conceptual preparations such as selecting an adequate testing platforms are discussed in chapter 4. The implementation of the selected memory protection schemes is described in chapter 5. Thereafter the effectiveness and shortcomings are evaluated in chapter 6. The conclusion is provided in chapter 7. Lastly, a short outlook into future work is provided in chapter 8.

2 Background

This section gives an introduction to constrained devices and the RISC-V ISA. The focus is on the privilege architectures and related features. Additionally, the RIOT operating system is introduced.

2.1 Constrained Devices

In RFC 7228 [9], constrained devices are described as “small devices with limited CPU, memory, and power resources”, which can be found in the Internet of Things. The IoT is only loosely defined, but typically describes physically and virtually connected devices, often using wireless technologies while powered by batteries. Hence the IoT is one of the influential factors for the design and development of constrained devices today.

Constrained devices usually consist of a microcontroller unit with attached sensors and other peripherals. Typical CPU clock speeds range from a few MHz up to a few hundred MHz and memory capacities in the order of kilobytes for both long term storage and working memory. Theses constrained computing systems often lack periphery that is considered standard in the conventional computing world. One of these peripherals are MMUs, which provide virtual memory. Virtual memory is universally found on smartphones, laptops and gaming consoles [10, p. 253], but is missing in constrained devices [11, p. 17].

2.2 RISC-V

RISC-V is an open standard Instruction Set Architecture [12]. As such, it is freely available to academia and industry. It is a general purpose ISA, suitable for hardware implementation. One of the core design principles is simplicity. It allows ease of use for research and educational purposes as well as for implementations. Unlike competing ISAs, e.g. ARM, RISC-V is not the intellectual property of a single company. This makes it easier for small organisations and companies to develop a chip using RISC-V [13]. Another core aspect of the ISA is extensibility, allowing companies to add their own special purpose hardware for domain specific tasks [14].

An executing hardware thread in a RISC-V CPU is called a hart.

2.2.1 RISC-V Extensions

A RISC-V base integer ISA is defined as the minimal ISA that must be present in any implementation. It is a minimal set of integer only instructions, sufficient to provide a ‘reasonable‘ execution environment for compilers, assemblers, linkers and operating systems [12, p. 4]. The RISC-V family consists of four base ISAs:

- RV32I, 32 Bit
- RV64I, 64 Bit
- RV32E subset of RV32I with limited register count
- RV128I, 128 Bit

As a ’reduced instruction set computing’ architecture, RISC-V features fewer instructions than established ISAs such as x86.

Base ISAs can be extended with a couple of standard definitions:

- M: Integer Multiplication and Division
- A: Atomic Instructions
- F: Floating-point Support
- D: Double-precision Floating-point
- C: Compressed Instructions
- Zicsr: Control and Status Registers
- Zifencei: Instruction-fetch Fencing

Number of Levels	Supported Modes	Suggested Use-case
1	M	Simple embedded systems
2	M, U	Secure embedded systems
3	M, S, U	Conventional computing

Table 2.1: Supported combination of privilege modes as defined in the RISC-V privileged architecture [1, 1.2 p. 3]

Level	Encoding	Name	Abbreviation
0	00	User/Application	U
1	01	Supervisor	S
2	10	Reserved	
3	11	Machine	M

Table 2.2: Privilege levels as defined in the RISC-V privileged architecture [1, 1.2 p. 3]

If an implementation supports all standard extensions, as for example `RV32IMAFD_-Zicsr_Zifencei` it is called a 'general-purpose' ISA and can be shortened to 'RV32G'.

2.2.2 Privileged Architecture

The RISC-V privileged architecture introduces privilege levels as shown in Table 2.2. A hart is always executing in exactly one privilege level at any given point in time. These levels provide protection of different components in the software stack [1, 1.2 p. 2]. If software attempts to perform an operation that is not allowed in the current privilege level, an exception will be raised by the hart. The highest privileges are available to code running in Machine-Mode (M-Mode), followed by the Supervisor-Mode (S-Mode) and the least privileges has code in the User-Mode (U-Mode). A minimal RISC-V implementation must at least implement the Machine-Mode. The systems of interest for this thesis provide Machine- and User-Mode, but not the Supervisor-Mode (compare Table 2.1).

System and hart specific settings such as the aforementioned privilege levels are controlled via the control and status registers (CSR).

2.2.3 Privileged Instructions

Certain instructions require elevated privileges to execute as they affect the overall system integrity and security. These include all instructions which read or write the CSRs as well as the `ecall/ebreak`. The later is used to trigger an exception by software on the current executing hart. This is usually used to initiate a system call, often called syscall.

To switch from the most privileged M-Mode to the least privileged U-Mode, the Machine-Previous-Privilege (MPP) bits in the `mstatus` register are set to `0b11` (representing Machine-Mode) and the exception return instruction `mret` is executed. During `mret`, the privilege mode is switched depending on the `MPP` bit field and execution is continued on the address stored in the `MEPC` (machine exception programm counter) register. The switch is typically done during a context switch in scheduling.

2.2.4 Physical Memory Protection

RISC-V Physical Memory Protection is described in the privileged specification section 3.7 `Physical Memory Protection`. A direct comparison with the more common ARM MPU is shown in Table 2.3. The PMP is capable of setting the memory area access attributes read, write and execute. Areas can be as small as 4 bytes in size. Each access rule consist of one configuration register and a corresponding address register as shown in Figure 2.1. The ISA allows to implement either 16 or 64 such rules. Each rule can also be locked, which marks the configuration registers as read-only until the next hart reset. Additionally, a locked rule is not only enforced on user-mode level threads, but also applies to Machine-Mode privilege threads. The PMP configuration registers are part of the Control and Status Registers and are therefore not mapped into the physical address space. Instead, special CSR instructions need to be used to access them. These instructions are available to Machine-Mode only. PMP violations are trapped precisely by the hart and erroneous accesses never succeed.

Any address stored in the PMP address registers must be shifted two bits to the right. As the smallest granularity is four bytes, this does not lead to loss in accuracy. Instead, it is used to allow defining access rules for a 34 bit address space. The address register supports three addressing modes: Top of range (TOR), Naturally aligned four-byte region (NA4) and Naturally aligned power-of-two region, $>=$ 8 bytes (NAPOT). When choosing TOR,

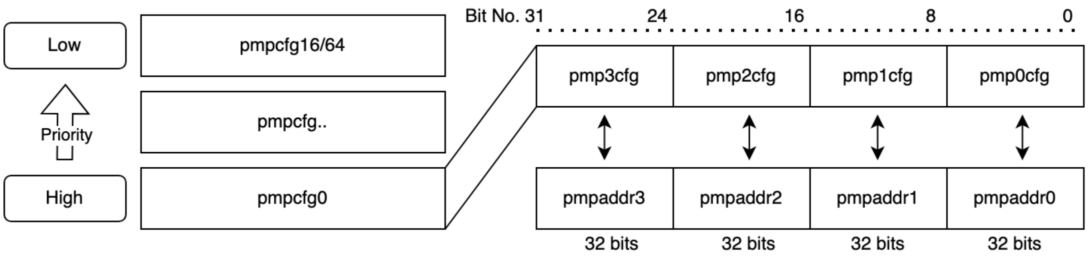


Figure 2.1: The organisation of the PMP registers in the RISC-V privileged specification. The priority is descending, therefor the lowest PMP configuration register (`pmp0cfg`) has the highest priority. Each `pmpXcfg` is 8 bit wide and has one corresponding 32 bit `pmpaddrX` register. An access to the configuration register is 32 bit wide and is able to manipulate four `pmpXcfg` at once. Hence the existence of the combined `pmpcfgX` register.

the address register forms the upper address (excluded) of the memory region, while the previous address register forms the lower (included) boundary. If TOR is chosen for the first address register, which has no preceding address register, the lower bound is 0x00000000. In the NA4 mode, which is a special case of NAPOT, the address must be aligned to four byte and the memory region is exactly 4 bytes in size. In order to use the NAPOT mode, the address must be aligned to the requested/intended area size, where the size must be a power of two, but at least 8 bytes. As the result of the alignment, the lower bits of the address are always zero and are repurposed to encode the size of the memory region.

2.3 Operating System Security

Operating system security needs to be distinguished from general computer and information security. In this thesis I focus on operating system security, as a subsection of computer security which emphasises the protection of operating system assets such as memory, processes and I/O control. In this context, 'protection mechanisms' are specific mechanisms provided by the operating system used for safeguarding the information on- and the integrity of the computing system [10].

	RISC-V PMP	ARM MPU
Smallest region size	4 Bytes	32 Bytes
Maximum region size	32 GB	4 GB
Region granularity	Configurable (>= 4 Bytes)	32 Bytes
Privileged / unprivileged settings	Hybrid / Mixed	Independent
Supported memory attributes	R/W/X	R/W/X
Maximum number of regions	16	16 (8 for privileged, 8 for unprivileged)

Table 2.3: Comparison of RISC-V PMP and ARM MPU main features following Lu [2, p. 9].

2.4 MMU and MPU

In the conventional computing area, memory management units (MMUs) are ubiquitous. They provide inter-process memory protection by separating processes into different address spaces [10, p. 186], realising process isolation. This is typically achieved by using virtual memory with a technique called paging [10, p. 194]. However, these MMU systems come at the cost of complexity both at the operating system integration level and in the hardware design of the CPU. This makes them unsuitable for constrained devices. Instead a smaller subset of the MMU is used: The MPU.

Memory protection units (MPU) protect the system from certain faults like buffer overflows. They do this by providing special hardware which oversees memory accesses and validates them according to preset rules. If a violation of these rules is encountered, an exception is generated and a kernel panic stops the execution[1]. Imagine, the memory protection unit gets configured by the operating system to disallow write access to a given buffer. If a write attempt into this buffer is encountered, an exception is generated and the buffer is not written to.

Isolation of process memory is needed for stability, security and privacy. Segregation into privilege modes/levels enhances security by containing faults and malicious actors in none critical areas.

[1]Typical behaviour, the exact response depends on the specific hardware and software in use.

2.5 The RIOT Operating System

RIOT [15] is a real-time multi-threaded operating system that targets microcontrollers with an emphasis on the IoT. One of its core requirements is to maintain a low memory footprint, which is an important aspect for most constrained devices. The development of RIOT is undertaken by an international open source community, which is not linked to individual vendors. The software is licensed under the copyleft license LGPLv2.1. An other attribute of RIOT is the modular nature of the underlying code base. This makes it easy to integrate new features and extensions.

RIOT is actively used in academic research. As such, it has often been used for security related research. The on-going activity in academia is highlighted with a short selection of recent security related publications:

- *Automated Detection of Spatial Memory Safety Violations for Constrained Devices* from 2022 discovered seven memory safety violation within RIOTs network stack [16].
- *Usable Security for an IoT OS: Integrating the Zoo of Embedded Crypto Components Below a Common API* from 2022 implemented an abstract cryptography API in RIOT [17]. With it, the usability, portability and performance overhead of cryptographic support in the IoT was evaluated.
- *PUF for the Commons: Enhancing Embedded Security on the OS Level* from 2023 designed and analyzed the integration of physically unclonable functions into RIOT [18].

2.5.1 Problems in Deploying Memory Protection in RIOT

As of today, RIOT rarely uses advanced memory protection schemes and in particular "RIOT is lacking essential protection mechanisms known from conventional operating systems" [19, p. 63-64]. On platforms where the ARM memory protection unit (MPU) is available, stack overflow detection and data execution prevention schemes are available although their effectiveness is limited [20]. Additionally, these mechanisms might not be active automatically thereby requiring supplemental effort and awareness from the executing developer [19, p. 63].

Currently, RIOT does not support the RISC-V physical memory protection (PMP) specification, even though multiple system on a chip (SoC) supported by RIOT implement the PMP. Additionally, RIOT also fails to utilise privileged execution environments on all platforms where this security feature is available. This includes the User- and Machine-Mode privilege level provided in the RISC-V privileged specification.

2.6 Data Execution Prevention

Buffer overflows remain one of the most prominent types of exploitation. A common way to exploit buffer overflows is the direct execution of the attacker-controlled payload using an overwritten return address. This can be prevented using the data execution prevention (DEP) security scheme. DEP is also known as `executable space protection` [19] or, when in the context of conventional computing platforms like x86, the `No-eXecute-Bit (NX-Bit)` [10, p. 644]. This protection disables instruction fetches from configurable addresses ranges including RAM. As such, hardware and software support is needed for this protection mechanism.

It is possible to use the PMP to mark the whole address space of the RAM as non-executable. On constrained devices, the operating system typically boots from Flash/ROM and keeps executing form it. Within RIOT, the execution is never passed to RAM. During the exploitation of a vulnerable buffer overflow, the malicious payload, that is overflowing the buffer, is stored on the stack in RAM [21]. Upon success of the attack, execution is typically passed to the attacker-controlled payload in RAM. On a system that marked the RAM as non-executable, this will trigger an exception at the CPU, bringing the control flow back to the operating system. In this situation, RIOT typically performs a kernel panic - stopping the attack immediately.

2.6.1 Security Limitations

Data execution prevention can be circumvented with Return-To-Libc or more generally with Return-Oriented-Programming (ROP). ROP is more difficult to execute on RISC-V compared to x86, as the program counter (PC) is not directly writeable and there are no distinct stack-return instructions, but the threat remains [22]. Since the attacker has some or full control of the stack, he can overwrite the return addresses of functions inside the stackframes. The attacker then returns the flow of control to, e.g., the driver of the

Example address	Function
0x80000100	Canary
0x800001...	Thread data
0x800001D0	Stack pointer sp
0x800001FF	Thread control block

Table 2.4: Example thread stack composition on RIOT. Note how the stack grows toward the canary as the stack space is filled.

memory protection unit with the argument registers setup to disable the data execution prevention.

2.7 Detection of Thread Stack Overflows

Listing 2.1: RIOT stack C-struct as found in `thread.h`. Various optional extensions not shown. The stack canary, located at `char *stack_start`, is only included when needed. For example, when the PMP-based stack overflow detection is in use, as indicated by the `MODULE_PMP_STACK_GUARD` define.

```
struct _thread {
    char *sp;                   /**< thread's stack pointer  */
    thread_status_t status;     /**< thread's status         */
    uint8_t priority;           /**< thread's priority       */

    kernel_pid_t pid;           /**< thread's process id     */

    clist_node_t rq_entry;      /**< run queue entry         */

#if defined(DEVELHELP) || IS_ACTIVE(SCHED_TEST_STACK) \
    || defined(MODULE_MPU_STACK_GUARD) \
    || defined(MODULE_PMP_STACK_GUARD) \
    || defined(DOXYGEN)
    char *stack_start;          /**< stack start address     */
#endif

#if defined(DEVELHELP) || defined(DOXYGEN)
```

```
    int stack_size;              /**< thread's stack size       */
#endif
};
```

It is desirable to guard the stacks of each thread against overflowing. RIOT currently does this by adding a canary value at the end of the stack as shown in Table 2.4. When the software-based overflow detection is used (`DEVELHELP`), the canary is the address of the canary itself. This reduces the chance that an attacker can guess the canary value and trick RIOT into not recognising the overflow. Additionally, the canary check can be calculated on the fly and does not need a compile time static magic value or otherwise access to any other values except the canary itself. The thread control block (TCB) is located at the start of the stack, followed by setup data such as function arguments. The thread usable stack space starts at the stack pointerr (SP) and spans up to the canary value. The location of the canary is also saved within the thread control block, as shown in Listing 2.1. If a fault starts overflowing the stack, it will attempt to write beyond the end of the stack, overwriting the canary. At the next scheduling, RIOT checks the correctness of the canary value and issues a kernel panic, if it is incorrect.

2.7.1 Security Limitations

The utility of stack overflow detection predominantly manifests during the development of software. Its primary benefit lies in easing the development and debugging processes by promptly identifying issues such as inadequate stack space or memory leaks. While it offers a certain security advantage, it is imperative to acknowledge that this benefit is constrained. The canary can potentially be bypassed, and its effectiveness is notably specific, rather than constituting a comprehensive security measure.

2.8 Thread Isolation

In traditional general purpose computing, processes are isolated from each other via virtual addressing [10, p. 194], often utilising paging. Isolation provides “a protection against unwanted information flow” as described by Biskup [23]. This is a safe and practical solution but comes at the cost of significant software and hardware complexity. On constrained devices, this is undesirable as low cost and small size of the MCU are significant requirements for the typical use case/deployment. Instead, embedded devices often feature only a flat physical address space and only support threads as a lightweight alternative for processes [10, p. 97]. With memory protection units, it becomes feasible to isolate threads from each other even without processes. During scheduling, the real-time operating system (RTOS) would only mark the next thread memory region as read-/writeable in the memory protection unit and disallow all other system memory access. This isolates threads from each other.

3 Related work

3.1 Usage of the ARM MPU in Embedded Systems

In *Good Motive but Bad Design: Why ARM MPU Has Become an Outcast in Embedded Systems* [20] the authors Wei Zhou et. al. show that the physical presence of the ARM MPU alone is insufficient to achieve additional security on embedded devices. During their investigation they identified some non-technical reasons as limiting factors in its wide spread adoption.

- The ARM MPU increases the transistor count and complexity of a given system on a chip. This in turn increases the price and the power demands, which contradicts common requirements of the IoT.
- Increased time-to-market pressure for the manufacturers as the IoT market grows, which encourages the reuse of existing code bases without ARM MPU integration.
- Developing new software leveraging the ARM MPU may cause compatibility issues.
- Most companies are reluctant to invest into security, when the existing code still works.

Technical reasons include the limited number of available regions, making it unnecessarily hard to use the MPU efficiently. Additionally, without further and deep OS integration, the security benefits can be easily circumvented as embedded OS typically do not offer process isolation. The performance overhead is considered too high for frequent context switches.

Relation to this work

While the authors had a focus on the FreeRTOS operating system, they believe that their observations apply to other operating systems because the demonstrated pitfalls root in the fundamental design drawbacks of MPU [20, 7, p. 4].

3.2 Memory Safety Using ARM MPU in RIOT

In 2014, Famulla explored the integration of the ARM MPU into RIOT in his thesis [24]. His proposed implementation featured memory protection through thread isolation and privileged execution. Significant changes to RIOT were required, including memory management and system call infrastructure. These heavy changes in the core component of RIOT prevented the integration of Famullas proposal at time.

Relation to this work

While this thesis does not implement thread isolation nor privileged execution, some of the challenges encountered during the integration of the RISC-V PMP into RIOT might be comparable to the challenges overcome by Famulla.

3.3 Checked C

The security of embedded devices can also be enhanced without special hardware features. One approach could be the usage of spatial memory safe programming language, such as Rust or the C dialect *Checked C*. The Checked C programming language enables a more precise articulation of a programmers intention regarding pointer utilization and the memory range encompassed by the data to which a pointer refers. Subsequently, this provided information serves as a basis for implementing runtime checks to identify instances where inadvertent access to incorrect data occurs, thus preventing errors from transpiring silently and evading detection. This comes at the cost of runtime performance and code size. Tempel and Bruns showed how *Checked C* can be integrated into RIOT [25].

Relation to this work

As the attack vector of this thesis assumes the usage of a memory unsafe programming language, this related work highlights one potential defense strategy and its possible integration into RIOT. It does not replace the protection mechanisms introduced in this thesis, but can be used as an additional measure.

3.4 Operating System Security in the Internet of Things

In his master thesis, Tempel showed through experimental testing and evaluation, whether the protection mechanisms provided by RIOT are sufficient [19]. He concluded that the lacking utilisation of security mechanisms (including hardware based) is a key contributor to his finding of multiple potentially software issues - which could have partially mitigated, if RIOT had implemented such protection mechanisms.

Relation to this work

Memory protection is among the protection mechanisms that were found by Tempel to be missing in RIOT [19, p. 26]. In this thesis, the memory protection unit based, data execution prevention and thread stack overflow detection are implemented for the RISC-V platform. Thereby partially addressing the shortcomings identified by Tempel.

3.5 Improvement of Stack Canaries

Hardware assisted buffer protection mechanisms can be an effective tool against malicious attacks as shown by Asmit De and co-authors [26]. They employed a Physically Unclonable Function (PUF)-based randomized canary generation technique to sidestep common challenges found in canary based protection mechanisms.

Relation to this work

One of the challenges is to choose a canary that cannot be guessed easily or predicted by an attacker. In this thesis, the software-based stack overflow detection using a canary is replaced by an hardware-based approach using the RISC-V PMP. This eliminates the concern of choosing a suitable canary.

4 Concept

Certain requirements need to be solved before the memory protection schemes can be implemented. When selecting an appropriate development target, it is critical to accurately assess the shortcomings of this target. If not accounted, such shortcomings can be mistaken for deficits in the PMP specification itself.

4.1 Evaluation Platforms

One ore more evaluation platforms need to be selected in order to verify and test the PMP-based memory protection schemes. At time of this thesis, RIOT supports multiple RISC-V based systems. Of those systems, only the HiFive1 Rev B and the ESP32-C3 feature a PMP.

4.1.1 HiFive1 Revision B by SiFive

The HiFive1 Revision B, manufactured by SiFive, is an evaluation board for the SiFive FE310-G002 microcontroller. It implements the RV32IMAC ISA, has PMP, and contains 16 kB of SRAM as well as 8 kB of flash memory [27, 28]. It is pictured in Figure 4.1.

The PMP of the FE310-G002 does not comply with the RISC-V specification: First of all, it implements only 8 regions instead of the 16 or 64 regions mandated by the specification [28]. Second, the FE310-G002 has a silicon bug that causes the locking of a region to always prohibit writing to the address register of the previous region as well, independent of the selected mode. This is the correct behaviour if the `TOR` mode had been selected, but is invalid for all other modes. This bug can be worked around by leaving an unused region before any locked region [29, FU-885].

The Hifive1 was selected as a testing platform.

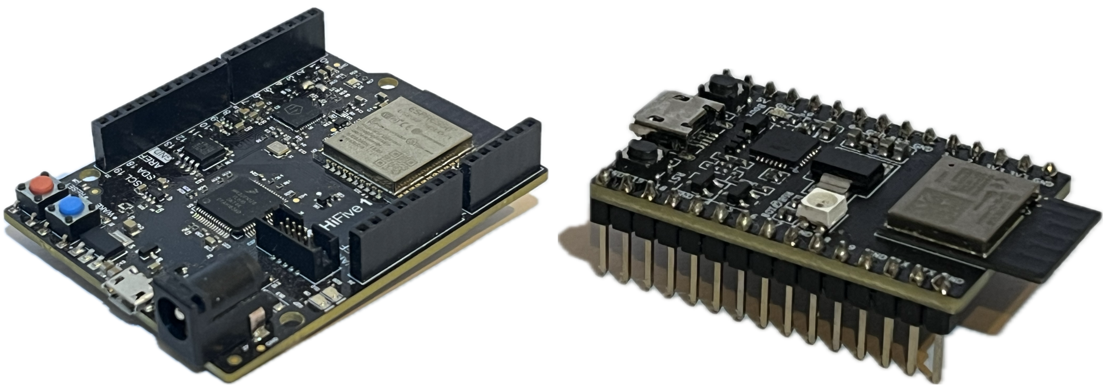


Figure 4.1: On the left: The HiFive1 Rev B development board. On the right: The ESP32-C3 development kit.

4.1.2 ESP32-C3-DevKitM-1 by Espressif Systems

The ESP32-C3-DevKitM-1 [30], devoloped by Espressif Systems, implements the RV32IMC ISA in it's SoC, including a PMP with 16 regions. It provides 400 kB of SRAM and 386 kB flash ROM. Care must be taken in regards to the PMP, as the manual clearly states that Espressif Systems' implementation differs from the ISA: Static priorities of the regions are not supported, meaning overlapping or concurrent regions behave erroneous compared to the ISA specification. Additionally, the maximum NAPOT range is limited to 1 GB.

The bootloader provided by Espressif Systems already uses and locks all available PMP regions. In order to use them from within RIOT, the bootloader must be patched accordingly.

Limitations of the ESP32-C3 integration of RIOT

RIOT supports various CPUs and boards through a modular framework to reduce duplicate code. In particular, the module `cpu/riscv_common` contains code needed to support CPUs implementing the RISC-V ISA.

However, the RIOT support for the ESP32-C3 does not use this module, but is implemented in the module `cpu/esp32` instead. This module adds support for all CPUs made by Espressif through the vendor-provided software development kit (SDK), called "Espressif IoT Development Framework (ESP-IDF)" [31]. Espressif manufactures CPUs implementing either the RISC-V or Extensa ISA, so the SDK abstracts over this ISA difference. As the module `cpu/riscv_common` is not used, the PMP driver must be made available to a ESP32-C3 RIOT project manually by the developer.

In addition, the SDK sets up the hardware, including the PMP, before handing off the control to the RIOT initialisation. The rules set in the PMP are fairly reasonable, for example disallowing writing into ROM areas. Because all entries are locked, they prohibit (re-)configuration of the PMP by my code. In order to run my implementation on the ESP32-C3, the SDK must be patched to not lock up the PMP configurations.

Fixing these integration issues is out of scope for this thesis. Due to this and the incomplete PMP implementation, the ESP32-C3 was not used for the evaluation of this thesis later on.

4.1.3 RISC-V Virtual Prototype

The RISC-V Virtual Prototype (VP) [32], developed by the Arbeitsgruppe Rechnerarchitektur (AGRA) of the University Bremen, implements RV32GC and RV64GC. Recently, experimental PMP support was implemented for which I got early access. The VP can simulate different hardware setups. I chose the Hifive1 Rev b, to keep my setups more comparable.

During a short experimentation phase, the VP frequently crashed and often exhibited incorrect behaviour in the PMP aspects. Due to this and the fact that the PMP support was not yet public (and with this, not final & reproducible), the VP was not used for the evaluation of this thesis later on.

4.2 Survey of PMP Specification Compliance

Both implementations of the PMP (by SiFive and Espressif Systems) are not compliant with the RISC-V PMP specification. The differences are listed in Table 4.1. Most notably is the deviation of the behaviour in case of overlapping regions, in which the ESP32-C3 does not behave correctly according to the specification. In addition, the HiFive1 Rev b does not support 16 (or 64) regions but 8, which is prohibited by the specification.

	RISC-V Spec	Hifive1 Rev b	ESP32-C3
Smallest region size	4 Bytes	4 Bytes	4 Bytes
Maximum region size	32 GB	32 GB	1 GB
Region granularity	>= 4 Bytes	4 Bytes	4 Bytes
Number of regions	16/64	8	16
Overlapping regions	Yes	Yes	No
Static priority by number	Ascending	Ascending	No

Table 4.1: Comparison of RISC-V PMP specification and different implementations. The behavior of a compliant PMP is described in detail in subsection 2.2.4.

4.3 The PMP Driver

During the development of the PMP driver, I encountered a potential shortcoming in the RISC-V architecture. Per ISA specification, it is not possible to select the target's register address of CSR read/write via a source register. Instead, the targets CSR address is encoded as an immediate value directly in the instruction. The immediate value can not be set at runtime via the default assembly and C ABI. Hence, a workaround was chosen, where all possible CSR read/write accesses needed for the PMP driver were collected within a `switch` statement as shown in Listing 4.1. Here, the compiler is emitting all required instructions with appropriate immediate values, which can then be selected at runtime.

The driver got merged into RIOT in pull request #19712 [33].

Listing 4.1: Example code showing the usage of a `switch` statement to choose the CSR register at runtime. Note how the CSR register address (0x3b0..) is statically provided at compile time within a string.

```
void write_pmpaddr(uint8_t reg_num, uint32_t value)
{
    assert(reg_num < NUM_PMP_ENTRIES);
    switch (reg_num) {
    case 0:
        __asm__ volatile ("csrw 0x3b0, %0" :: "r" value);
        break;
    case 1:
        __asm__ volatile ("csrw 0x3b1, %0" :: "r" value);
```

```
            break;
    /* Up to 15/63 ... */
    }
}
```

4.4 Caching for PMP Regions

During the evaluation, as later explained in subsection 6.3.3, I experienced a certain performance bottleneck when deploying stack overflow detection using the PMP. In order to minimise the runtime penalty, I implemented a caching algorithm for the PMP region configurations, specific to RIOT OS scheduling.

The caching is a classical last-recently-used (LRU) algorithm. During scheduling, it is checked if the next thread is the same as the previous thread (not the current). If so, no reconfiguration of the PMP regions is necessary. The algorithm marks this caching entry as recently used and the current entry is now considered the oldest entry. Should the next thread not be cached, the oldest entry is reconfigured for the next thread. Pseudo code of this algorithm is shown in Listing 6.6.

4.5 Thread Isolation

An isolated thread should be unable to access (reading/writing) memory outside its own stack space. While the necessary PMP rule is trivial, which disallows all memory access except for the current stack, the necessary infrastructure needed to enable lower privileged threads is not. For a rudimentary proof of concept of thread isolation, RIOT must be extended with a system call infrastructure as well as a way to store the information which threads are lower privilege. The information is needed during scheduling to determine the next privileges after the context switch completed.

Another issue with unprivileged execution within RIOT is the use of privileged instructions within numerous kernel functions. These instructions and with them their containing functions cannot be executed by the unprivileged thread and will throw an exception when attempting to execute them. While some functions can be avoided others are necessarily required by RIOT's design such as in the `thread_yield()` function. The function is used to yield the currents thread execution to the scheduler and uses privileged instructions to write certain CSRs with the goal of disabling IRQs before entering the scheduler. For such functions, syscalls must be implemented.

5 Implementation

This chapter explains my implementation of the the data execution prevention and the thread stack overflow detection.

5.1 Data Execution Prevention using PMP

Listing 5.1: The RISC-V initialisation function of RIOT. If the PMP based data execution prevention is enabled, the PMP driver is used to permanently revoke the execution permission of the RAM address space.

```
void riscv_init(void)nm
{
    riscv_fpu_init();
    riscv_irq_init();
#ifdef MODULE_PMP_NOEXEC_RAM
    /* This marks the (main) RAM region as non
     * executable. Using PMP entry 0.
     */
    write_pmpaddr(0, make_napot(CPU_RAM_BASE, CPU_RAM_SIZE));

    /* Lock & select NAPOT, only allow write and read */
    set_pmpcfg(0, PMP_L | PMP_NAPOT | PMP_W | PMP_R);
#endif
}
```

To prevent execution from RAM, the PMP driver was used to revoke the execution permission of the RAM memory addresses, for all privileged modes during startup of the system.

The implementation uses the RIOT provided RAM addresses. To make them always available at compile time, I slightly modified the build system [34]. The data execution prevention setup is located in the generic RISC-V startup code, as shown in Listing 5.1. The NAPOT addressing mode was chosen as many embedded systems feature a naturally aligned RAM address space. The configuration is locked to prevent further tampering as the locking bit (`PMP_L`) makes further writes to this configuration impossible until the next CPU reset. Additionally, the locking ensures that this configuration applies to all privilege levels, including Machine-Mode, which is RIOT's only used mode.

This implementation of data execution prevention does not work on the ESP32-C3 platform because RIOT's build system does not utilise the RISC-V common CPU modules (`cpu/riscv_common/`) for this platform. Instead, RIOT relies on the proprietary vendor provided software development kit for startup and basic hardware abstraction.

The data execution prevention got merged within the same pull request as the PMP driver which is #19712 [33].

5.2 Thread Stack Overflow Detection using PMP

Listing 5.2: Subsection of the RISC-V initialisation function of RIOT.

```
void riscv_init(void)nm
{
    ...
#ifdef MODULE_PMP_STACK_GUARD
    /* Make sure that none-locked rules also apply to M-Mode */
    clear_csr(mstatus, MSTATUS_MPP);
    set_csr(mstatus, MSTATUS_MPRV);
#endif
}
```

Listing 5.3: Subsection of RIOT's scheduling routine showing the configuration of the PMP based stack overflow protection.

```
thread_t *__attribute__((used)) sched_run(void)
{
    ...
#ifdef MODULE_PMP_STACK_GUARD
```

```
        write_pmpaddr(3, (uintptr_t)next_thread->stack_start);
        set_pmpcfg(3, PMP_NA4 | PMP_R);
#endif
    ...
}
```

As explained in section 2.7, the software based stack overflow detection in RIOT only acts when the scheduler is called. An obvious disadvantage of this is, that overflows can only be detected after the fact. This might already be too late to even gracefully display an error and stop the system. As soon as a defective code starts overflowing the stack it might overwrite critical data structures of the operating system. The RISC-V PMP can be used to detect stack overflows the exact moment code tries to write past the end of the stack, onto outside memory. This ensures that RIOT crashes gracefully as the overflowing code does not get a chance to corrupt critical data structures. This promptness eases the debugging of the underlying defect.

In this implementation the four byte canary value is marked as read-only. The canary is the same as in the case for the software based approach. This way, the thread control block struct does not need to be adjusted specifically for this implementation. See Listing 2.1 in section 2.7 for details. In case the stack overflows and attempts to write into the read-only canary, the PMP raises a store access-fault exception immediately [1, Sec. 3.7.1, p. 58]. Since the canary is specific to the current active thread, the stack guarding settings in the PMP need to be reconfigured during scheduling if the thread is switched. The configuration procedure of the PMP for the next active thread is done during scheduling, as shown in Listing 5.3. Conveniently, the `PMP_NA4` addressing mode can be used, which specifies a naturally aligned, 4 byte memory area, which matches exactly the properties of the stack canary. The access right is set to `PMP_R`, which translates to read-only.

In addition, the effective memory privilege is set to User-Mode once during startup in the RISC-V initialisation function, as shown in Listing 5.2. This ensures that the PMP restricts memory accesses even though RIOT executes all code in Machine-Mode level privilege. By default, PMP rules only apply to the least privileged User-Mode. It is not possible to ensure that my stack guarding is in effect by locking the corresponding PMP rule, as I did for the data execution prevention: Locked rules can not be reconfigured until the RISC-V hart is reset, but the rule needed for stack overflow detection needs to be changed on every context switch. This leaves lowering the effective memory privilege as the only viable option for RIOT, as implemented here.

My use of the RISC-V PMP for stack overflow detection is similar to the RIOT feature `MODULE_MPU_STACK_GUARD`, which makes use of the ARM memory protection unit (MPU). Instead, the ARM MPU based stack overflow detection marks 32 bytes as read-only, presumably because it is the smallest possible granularity (see Table 2.3) of the ARM MPU. This is significantly more occupied memory compared to my PMP based implementation, which only marks the four bytes of the stack canary. While it often is a good idea to keep behavior consistent between platforms, I chose to limit the amount of bytes used for stack guarding in order to reduce the memory footprint.

My implementation of stack overflow detection using the RISC-V PMP for RIOT is available as GitHub pull request #19821, which has not yet been merged into RIOT as of December 8th, 2023. [35]

6 Evaluation

In this chapter, the evaluation of the run time performance impact of the two deployed memory protection schemes is shown.

6.1 Measurement Setup

Listing 6.1: Test program, which repeatedly turns a GPIO on and off indefinitely. Allows to measure the time it takes to toggle a GPIO.

```
int main(void)
{
    while (1) {
        gpio_set(GPIO_PIN(0, 22));
        gpio_clear(GPIO_PIN(0, 22));
    }
    return 0;
}
```

Listing 6.2: This measures the minimum (lower bound) run time overhead, but it is expected that it will increase under real world conditions.

```
int main(void)
{
    while (1) {
        gpio_set(GPIO_PIN(0, 22));
        set_pmpcfg(0, PMP_NA4 | PMP_R);
        gpio_clear(GPIO_PIN(0, 22));
    }
    return 0;
}
```

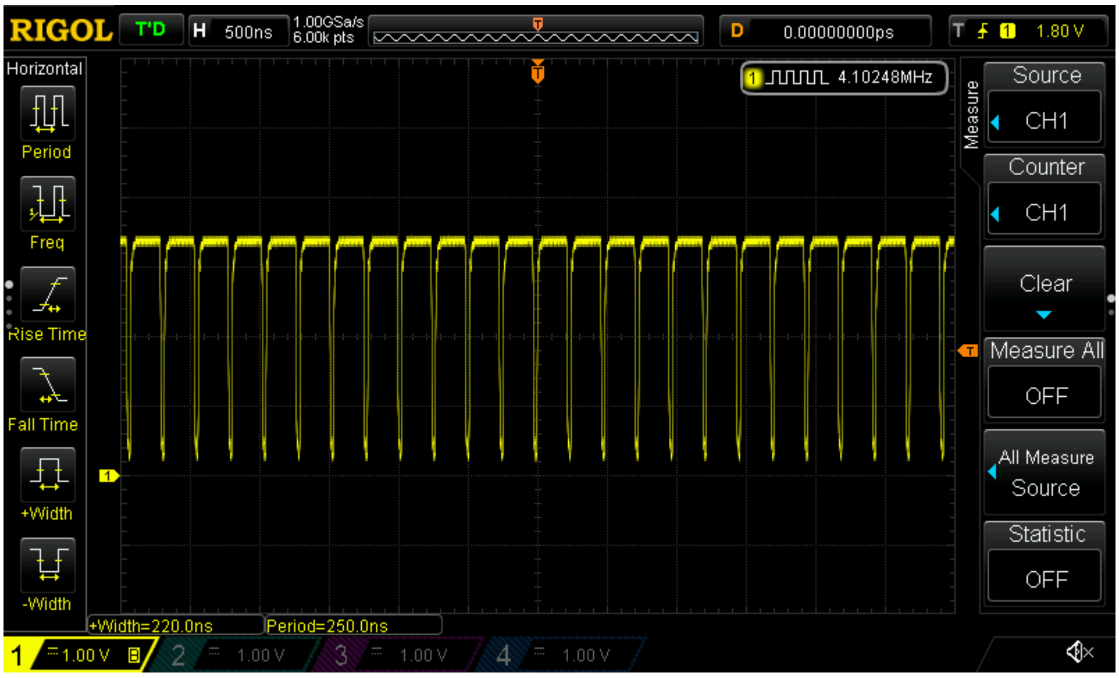
Listing 6.3: Based on the `IPC-Pingpong` example. The main thread creates a second thread before going into an endless loop in which the GPIO is first turned off and then the execution is yielded to the scheduler. The second thread also runs in an endless loop. It first turns the GPIO on and then yields its execution to the scheduler. With this program, the time spent inside the scheduler can be measured externally, by observing the time the GPIO is on high.

```c
void *second_thread(void *arg)
{
    (void) arg;
    while (1) {
        gpio_set(GPIO_PIN(0, 22));
        thread_yield();
    }

    return NULL;
}

char second_thread_stack[THREAD_STACKSIZE_MAIN];

int main(void)
{
    gpio_init(GPIO_PIN(0, 22), GPIO_OUT);

    thread_create(second_thread_stack, sizeof(second_thread_stack),
                  THREAD_PRIORITY_MAIN - 1, THREAD_CREATE_STACKTEST,
                  second_thread, NULL, "pong");

    while (1) {
        gpio_clear(GPIO_PIN(0, 22));
        thread_yield();
    }
}
```

The measurements were conducted using an oscilloscope (`RIGOL DS1054`) which observed the frequency that one GPIO pin could be toggled by any given setup, an example

Setup	GPIO high time
GPIO toggle only	29.20 ns
Writing a PMP config	220 ns
Writing a PMP address	1.860 µs
Thread context switch	940 ns

Table 6.1: Reference speed measurements on the Hifive1 Rev B. Compiled with `DEVELHELP = 0`.



+Width=220.0ns
Period=250.0ns

Figure 6.1: Measurement of the reference setup where only a single PMP configuration is written. The screenshot is taken on a RIGOL DS1054 oscilloscope. Writing one PMP configuration takes approximately 220 ns.

measurement is shown in Figure 6.1. This indirectly measures the time spent on a given task by the system. Assessing performance by measuring the time taken to complete a task (task completion time) is the only viable way of evaluating and comparing the speed of computing systems as shown by Patterson et al “Execution time is the only valid and unimpeachable measure of performance” [36, p. 54]. Other indicators have been proposed such as ’instruction count’ or ’instructions per clock cycle’ but those are inadequate [36, p. 32f]. For example, two tasks might have the exact same instruction count, but one regularly stalls the CPU pipeline, resulting in much poorer performance compared to the other task.

For reference in Table 6.1, the minimal time requirements for GPIO switching, PMP access and thread context switches were measured. The measurements contain the time it takes to loop from the end of a task back to the beginning. Since these time spans are minimal and are also constant for all tasks and setups, they do not influence the relative comparison of task completion times later on.

6.2 Data Execution Prevention

Listing 6.4: A unit test that tests the correct operation of the data execution prevention. First, a small buffer is allocated and filled with values that, when interpreted as RISC-V instructions, are illegal instructions. Next, inline assembly is used to craft a jump instruction which targets this buffer. When the data execution prevention works, an exception will be raised by the RISC-V hart due to the memory access fault. Otherwise, an illegal instruction fault exception will be raised.

```
#define JMPBUF_SIZE 3

int main(void)
{
    uint32_t buf[JMPBUF_SIZE];

    /* Fill the buffer with invalid instructions */
    for (unsigned i = 0; i < JMPBUF_SIZE; i++) {
        buf[i] = UINT32_MAX;
    }
```

	text	data	bss
Without data execution prevention	8766	884	2312
With data execution prevention	9148	1072	2312
Overhead	382	188	0
LTO; Without data execution prevention	8054	872	2312
LTO; With data execution prevention	8126	996	2312
Overhead	72	124	0

Table 6.2: Size in bytes of the `text`, `data` and `bss` section of the `pmp_noexec_ram` unit test when compiled for board `hifive1b` with or without data execution prevention.

```
    puts("Attempting␣to␣jump␣to␣stack␣buffer␣...\n");
    __asm__ volatile ("jr␣%0" :: "r" ((uint8_t*)&buf));

    return 0;
}
```

6.2.1 Unit Test

To ensure that the data execution prevention works as intended and actually provides the security benefit discussed in section 2.6, a unit test was written and introduced to RIOT as shown in Listing 6.4. The unit test forces execution from a buffer placed in RAM, that is filled with illegal RISC-V instructions. Assuming that the data execution prevention works, the RISC-V hart will immediately raise a memory access fault exception resulting in a RIOT panic. In case that the data execution prevention does not work, the hart will instead raise an illegal instruction fault exception and the test will fail.

6.2.2 Memory Consumption

The memory footprint was measured by building the unit test twice, once with data execution prevention enabled and once without. Then the size difference of the relevant ELF sections (`.text`, `.data` and `.bss`) was calculated as shown in Table 6.2. In this case, the total size difference was 570 bytes and 196 bytes when utilising link time optimisations (LTO). A compiler can use LTO to infer which code paths are actually executed in between compilation units. The memory consumption mainly results from the inclusion of the PMP driver. The overall size difference is lower when RIOT utilise LTO. LTO is currently not enabled by default. In this specific case, only one PMP region is being used and no format strings are needed, leaving a lot of room for link time optimisations.

6.2.3 Performance Limitations of Data Execution Prevention

The code needed, to implement the data execution prevention, is executed only once on startup. Therefore the run time overhead is expected to be negligible and was not measured.

6.2.4 Summary

Data execution prevention is a widely used security measure in general purpose computing (see section 2.6). DEP was successfully implemented using the RISC-V PMP and it's behaviour was tested to be correct on a constrained device. No run time overhead could be observed. The memory footprint is minimal (see Table 6.2). As shown, this measure can be used on constrained devices as well and enabling DEP is strongly recommended.

6.3 Stack Overflow Detection

Listing 6.5: A unit test that tests the correct operation of the PMP-based stack overflow detection. An infinite recursion is used to repeatedly push saved registers onto the thread's stack. When the stack overflows, the overflow detection should immediately detect it and force RIOT into a kernel panic based on a memory access fault. In all other cases, the test is considered to have failed.

```
#define CANARY_VALUE 0xdeadbeef

static struct {
    char overflow_buffer[128];
    unsigned int canary;
    char stack[THREAD_STACKSIZE_MAIN];
} buf;

static inline unsigned int __get_SP(void) {
    unsigned int __tmp;
    __asm__ volatile ("mv␣%0,␣sp" : "=r"(__tmp));
    return __tmp;
}
static int recurse(int counter) {
    printf("counter␣=%4d,␣SP␣=␣0x%08x,␣canary␣=␣0x%08x\n", counter,
            (unsigned int)__get_SP(), buf.canary);

    if (buf.canary != CANARY_VALUE) {
        printf("canary␣=␣0x%08x\nTest␣failed.\n", buf.canary);

        for (;;)
            thread_sleep();
    }
    counter++;

    /* Recursing twice here prevents the compiler from optimizing-out the recursion. */
    return recurse(counter) + recurse(counter);
}
static void *thread(void *arg) {
    (void) arg;
    recurse(0);
    return NULL;
}
int main(void) {
    puts("\nPMP␣Stack␣Guard␣Test\n");
    puts("If␣the␣test␣fails,␣the␣canary␣value␣will␣change␣unexpectedly");
    puts("after␣~50␣iterations.␣If␣the␣test␣succeeds,␣the␣MEM␣MANAGE␣HANDLER");
    puts("will␣trigger␣a␣RIOT␣kernel␣panic␣before␣the␣canary␣value␣changes.\n");
#ifdef MODULE_PMP_STACK_GUARD
    puts("The␣pmp_stack_guard␣module␣is␣present.␣Expect␣the␣test␣to␣succeed.\n");
#else
    puts("The␣pmp_stack_guard␣module␣is␣missing!␣Expect␣the␣test␣to␣fail.\n");
#endif

    buf.canary = CANARY_VALUE;
    thread_create(buf.stack, sizeof(buf.stack), THREAD_PRIORITY_MAIN - 1, 0,
            thread, NULL, "thread");
    return 0;
}
```

	text	data	bss
Without stack overflow detection	9028	1236	3724
With stack overflow detection	9332	1396	3724
Overhead	304	160	0
LTO; Without stack overflow detection	8402	1224	3724
LTO; With stack overflow detection	8522	1352	3724
Overhead	120	128	0

Table 6.3: Size in bytes of the `text`, `data` and `bss` section of the `pmp_stack_gurad` unit test when compiled for board `hifive1b` with or without stack overflow detection.

6.3.1 Unit Test

In order to verify the proper functionality of the stack overflow detection and its ability to promptly identify the occurrence of an overflow as soon as it occurs, I have incorporated a unit test. As shown in Listing 6.5, the unit test utilities an infinite recursion of function calling. In particular, the `recurse()` function calls itself before terminating, resulting in an infinite call tree. Each time a function is called, the currently used registers are pushed onto the threads stack to prevent the called function from overwriting them. This process is often called 'saving'. Upon return of the function, the 'saved registers' are restored by popping them from the stack. However, because the recursion is infinite, no function ever returns and so this step is not performed (in this particular case). Each time registers are saved, the stack grows to accommodate the space needed. Eventually, the threads stack space is exhausted and the process of saving the registers overflows the stack. This should be immediately detected by the PMP-based stack overflow detection, forcing RIOT into a kernel panic based on a memory access fault. In that case, the unit test passed. All other behaviours are considered failed.

6.3.2 Memory Consumption

Again, as with subsection 6.2.2, the memory footprint was measured by building the unit test twice, once with stack overflow detection enabled and once without. Then the size difference of the relevant ELF sections (`.text`, `.data` and `.bss`) was calculated as shown in Table 6.3. In this case, the total size difference was 464 bytes and 248 bytes with LTO enabled. Since the overhead is close to the observation with DEP, it is clear that most of the memory usage originates from the PMP driver. The run time memory consumption in RAM is minor.

6.3.3 Run Time Overhead of Stack Overflow Detection

Adding stack overflow detection using the PMP is expected to add a run time overhead because the frequently called scheduler gets extended in complexity. During scheduling, the PMP gets reconfigured, adding at least the overhead observed in Table 6.1 to the run time. I conducted additional measurements to see if the observed (lower bound) overhead increases during scheduling. Secondly, the measurements enable the comparision of the PMP-based stack overflow detection with the software-based one.

For the measurements, the code from Listing 6.3 is used with different compilation options. The code is being used as an example task in which two threads are scheduled alternating. One thread pulls the GPIO to high, while the other clears it, thereby enabling measuring the GPIO high time externally using an oscilloscope to determine the time needed to perform one context switch. Three setups are being measured: The pure time it takes to do one context switch. This is used as a baseline for comparison. Second, the existing software-based stack overflow detection present in RIOT. It is enabled by passing `SCHED_TEST_STACK=1` to RIOT using the `CFLAGS` during compilation. Lastly, the PMP-based stack overflow detection is enabled using `USEMODULE += PMP_STACK_GUARD` and measured.

The results in Table 6.4 show a neglectable overhead of the software-based stack overflow detection and an overhead of 3.63 µs for the PMP-based. The relative overhead is calculated by subtracting the GPIO high time of the baseline from the GPIO high time of the individual task setup.

As the overhead is constant per context switch, the run time penalty strongly depends on the application in question. For example, if context switches are infrequent and a lot of CPU time is spend within a thread before yielding, the run time overhead can be considered minor or even be neglected all together.

One reason for the relative big overhead could be the design of the CPU present on the Hifive1 Rev B board. On this CPU, the FE310 G002, regular CSR reads have a three clock cycle result penalty. A CSR write additionally flushes the CPU pipeline with a five

Task Setup	GPIO high time	Relative overhead
No features	940 ns	baseline
Software-based stack overflow detection	940 ns	0.0 µs
PMP-based stack overflow detection	4.57 µs	3.63 µs

Table 6.4: Performance measurements of the testing task on the Hifive1 Rev B.

cycle penalty. This means that during each context switch the pipeline is flushed and with it the instruction fetch cache [28, p. 17, 3.3].

6.3.4 Caching

When using a simple last-recently-used (LRU) caching strategy, the performance can be enhanced. The caching works by having two PMP rules active at once: the next active thread and the last thread. When switching between threads, no reconfiguration of the PMP is needed when the next active thread is the same as the last thread. Otherwise, the PMP needs to be reconfigured. The cache is updated on each context switch. Pseudocode of this algorithm is shown in Listing 6.6.

Listing 6.6: Pseudo code showing the caching algorithm as executed during scheduling.

```
if next_thread != cached_thread:
    setup_pmp(next_thread)
cached_thread = last_thread
last_thread = next_thread
```

The question arises as to why not cache an even larger number of entries. This limitation is primarily due to the constrained availability of regions, necessitating their resourceful utilization. Furthermore, employing more intricate caching algorithms can introduce additional overhead during context switches, potentially negating the time savings achieved by abstaining from CSR read/writes. Additionally, there is the challenge of managing cache invalidation when application threads are terminated and their stack space is freed.

While a short proof of concept showed that such a caching algorithm can reduce the relative overhead, it was not further pursued in this thesis. Measuring the efficiency of caching strategies is far from trivial as care must be taken to disambiguate best-, worst- and average case scenario in relation to common workloads.

6.3.5 Summary

It is feasible to implement thread stack overflow detection. The memory overhead showed in Table 6.3 is within the common limits of constrained devices.

The promptness of the PMP-based approach is a great advantage over the software-based detection. Due to the direct intervention by the CPU (via an exception) as soon as the canary would be overwritten, it is no longer necessary to await a context switch to verify the correctness of the canary. Thereby saving time and increasing the chance that a fault is found early in development. In addition, execution is stopped (or the flow of control diverted) before the overflowing memory access can influence the stability of the overall system. This is especially important for embedded and real time application, which might have tight couplings with physical processes.

Depending on the application, the overhead introduced during scheduling might not be tolerable. In these cases the feature can be selective turned on during development and be disabled in production, thereby, completely removing any penalty on the constrained device in deployment.

7 Conclusion

In this work, I successfully built a prototype that utilises the RISC-V physical memory protection from within the RIOT operating system. The PMP has low demands in terms of processing speed and memory size. Using two exemplary memory safety schemes, it was shown in chapter 6 that the PMP can feasibly be utilised by constrained devices.

Data execution prevention was implemented with no negative impact on the performance of the constrained device. Together with the negligible size overhead and the added restriction to the exploitability of common vulnerabilities, it is a good candidate to be always enabled on default.

The RISC-V PMP was used to enhance the usability and accuracy of the stack overflow detection, thereby increasing its usability as a developer tool. Developer must take the added overhead during context switch into account, depending on the intended application.

7.1 Drawbacks of RISC-V PMP

The limitations associated with the RISC-V Physical Memory Protection predominantly result from non-technical factors. Its availability is limited, as only a small number of platforms offer support for it. Frequently, the physically realized PMP, as executed by vendors, exhibits substantial deviations from the specification. The integration of advanced, PMP-based security measures, such as thread isolation, proves to be a challenging, especially when incorporating it into pre-existing codebases. This observation matches the conclusion of Wei Zhou et. al. that operating system integration is one of the major limiting factors, in the adoption of the ARM MPU [20].

7.2 Limitations of Thread Stack Overflow Detection and Data Execution Prevention

Even with stack overflow detection and data execution prevention, there are numerous ways for an attacker to access private information or gain control over the flow of execution in the event of a memory corruption vulnerability. One prominent one, return oriented programming, is significantly harder on RISC-V, but as shown by Cloosters et. al. [22], it is still a feasible attack vector.

Proper thread isolation would add a significant barrier to the exploitation of common programming errors. This is especially important for RIOT as its code base is overwhelmingly written in the memory unsafe language C.

8 Future Work

This chapter provides an outlook on points of interest for future research.

8.1 Obstacles to Thread Isolation in RIOT

The RISC-V PMP, together with the privilege modes, is suited for implementing thread isolation. Thread isolation can be used in addition to DEP to further increase the security of the system. With it, the PMP is used to disallow any given thread to access other threads memory. This includes the stack overflow detection scheme as a side effect of the isolation.

RIOT misses certain building blocks which are needed in order to implement thread isolation:

8.1.1 Memory Maps not Generalised

The RIOT operating system does not have the memory layout per system programmatically available. Since e.g. the address and size of RAM sections are needed at runtime to configure the data execution prevention, it is unnecessary hard to implement these feature in a generalised way, that is easy to be used and adopted for different systems. With thread isolation this issue is even more challenging as access rights become stricter and more spatially segmented.

One solution could be devicetrees[1]. A devicetree is a data structure that describes the hardware of a device. As an example, the IoT focused operating system Zephyr[2] already utilities devicetrees to configure the operating system for a given device.

[1]See https://www.devicetree.org/.
[2]See https://docs.zephyrproject.org/latest/build/dts/index.html.

8.1.2 Missing Concept of Privilege Modes

The concept of different execution privileges is not present in RIOT. This means that even if a thread with lower privileges is run, it is impossible to know which kernel functions can be called in the given privilege level. Some functions might not execute in a lower privilege mode as they require access to special peripherals or depend on privileged instructions.

8.1.3 Missing Syscall Infrastructure

With system calls (syscalls), lower privileged threads could 'ask' higher privileged ones to execute certain operations for them. The RIOT operating system does not implement nor provide infrastructure for syscalls.

8.1.4 Ownership of Shared Memory

Shared memory and inter process communication between threads becomes a challenge when thread isolation is used. RIOT does not have a concept of memory ownership, making it difficult to implement thread isolation. In the current state, all threads *own* all memory simultaneously. At any given time a thread can send a message to any other thread by writing into its memory. This would not work under isolation, as the thread can no longer access other threads memory. RIOT's IPC mechanisms need to be reworked to accommodate for this.

Another issue that needs to be solved is globally shared variables in the `.data` segment of the binary. Global variables are typically located outside the current threads stack and are thus not accessible due to the isolation constraints.

8.2 Generic API Abstraction for Memory Protection Units

Given the similarities of the ARM MPU and the RISC-V PMP, the idea of implementing an abstraction API seems promising. Most tasks performed with a memory protection unit are logically and functional identical between different hardware implementations. A generic API would decouple the operating system design and the implementation of memory protection schemes from the underlying hardware, simplifying the assessment and transportability of RIOT's MPU-based security schemes on different hardware targets (boards).

Developing this generic API to serve as an abstraction layer for the ARM MPU and the RISC-V PMP might appear to be a straightforward. However, several key challenges need to be considered:

8.2.1 Dynamic Tracking of (Un-)Utilized Regions

The need to dynamically monitor, which memory regions were in use at runtime, poses a significant challenge. The necessity to combine static (compile-time) and dynamic rules within the API introduces an additional layer of complexity. Balancing these two types of rules while ensuring smooth interaction presents a non-trivial task.

8.2.2 Ordering of Rules

The order, in which memory protection rules are applied and stored in hardware, is crucial due to the priority logic depending on it. However, the ordering differs for each specific hardware backend. Managing this variability adds intricacy to the APIs development.

8.2.3 Granularity and Alignment Variations

One additional challenge results from the disparities in minimal region size, granularity, and alignment requirements between different hardware architectures. These discrepancies require careful consideration and accommodation in the API design. For example, a common task could be to mark the smallest possible region as read-only, as it is the case for the canaries of the stack overflow detection. On the PMP this is typically four bytes, while the ARM MPU consumes at a minimum of 32 bytes.

In light of these complexities, the development of a generic memory protection API demanded additional planning and a deep understanding of the intricacies of the underlying hardware architectures and was not further pursued in this thesis.

Bibliography

[1] A. Waterman, K. Asanovic, and J. Hauser, *The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 20211203*, RISC-V International, 2021. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Priv-v1.12/riscv-privileged-20211203.pdf

[2] T. Lu, "A survey on risc-v security: Hardware and architecture," Marvell Semiconductor Ltd., Tech. Rep., 2021.

[3] K. R. Raghunathan, "History of microcontrollers: First 50 years," *IEEE Micro*, vol. 41, no. 6, pp. 97–104, 2021.

[4] M. Noseda, F. Frei, A. Rüst, and S. Künzli, "Rust for secure iot applications," in *Embedded World Conference, Nuremberg, Germany.* WEKA, 2022. [Online]. Available: https://digitalcollection.zhaw.ch/handle/11475/25046

[5] L. Szekeres, M. Payer, T. Wei, and D. Song, "Sok: Eternal war in memory," in *2013 IEEE Symposium on Security and Privacy*, 2013, pp. 48–62.

[6] M. Stone and G. P. Zero, "The more you know, the more you know you don't know," 2022. [Online]. Available: https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html

[7] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. Vijaykumar, and A. Jalote, "Detection and prevention of stack buffer overflows attacks," *Communications of the ACM*, vol. 48, no. 11, 2005.

[8] The Rust teams, "The rust programming language." [Online]. Available: https://www.rust-lang.org

[9] C. Bormann, M. Ersue, and A. Keranen, "Terminology for Constrained-Node Networks," IETF, RFC 7228, May 2014. [Online]. Available: https://doi.org/10.17487/RFC7228

[10] H. B. Andrew S. Tanenbaum, *Modern Operating Systems*. Pearson Education Limited, 2014.

[11] H. Xia, "Capability memory protection for embedded systems," University of Cambridge, Computer Laboratory, Tech. Rep. UCAM-CL-TR-955, Feb. 2021. [Online]. Available: https://www.cl.cam.ac.uk/techreports/UCAM-CL-TR-955.pdf

[12] A. Waterman and K. Asanovic, *The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 20191213*, RISC-V Foundation, Dec. 2019. [Online]. Available: https://github.com/riscv/riscv-isa-manual/releases/download/Ratified-IMAFDQC/riscv-spec-20191213.pdf

[13] M. Sharma, E. Bhatnagar, K. Puri, A. Mitra, and J. Agarwal, "A survey of risc-v cpu for iot applications," in *International Conference on Innovative Computing & Communication (ICICC) 2022*, feb 2022.

[14] S. Greengard, "Will risc-v revolutionize computing?" *Commun. ACM*, vol. 63, no. 5, p. 30–32, apr 2020. [Online]. Available: https://doi.org/10.1145/3386377

[15] E. Baccelli, C. Gündogan, O. Hahm, P. Kietzmann, M. Lenders, H. Petersen, K. Schleiser, T. C. Schmidt, and M. Wählisch, "RIOT: an Open Source Operating System for Low-end Embedded Devices in the IoT," *IEEE Internet of Things Journal*, vol. 5, no. 6, pp. 4428–4440, December 2018. [Online]. Available: http://doi.org/10.1109/JIOT.2018.2815038

[16] S. Tempel, V. Herdt, and R. Drechsler, "Automated detection of spatial memory safety violations for constrained devices," in *2022 27th Asia and South Pacific Design Automation Conference (ASP-DAC)*, 2022, pp. 160–165.

[17] L. Boeckmann, P. Kietzmann, L. Lanzieri, T. C. Schmidt, and M. Wählisch, "Usable security for an iot os: Integrating the zoo of embedded crypto components below a common api," in *Proceedings of the 2022 INTERNATIONAL CONFERENCE ON EMBEDDED WIRELESS SYSTEMS AND NETWORKS*, ser. EWSN '22. New York, NY, USA: Association for Computing Machinery, 2023, p. 84–95.

[18] P. Kietzmann, T. C. Schmidt, and M. Wählisch, "Puf for the commons: Enhancing embedded security on the os level," *IEEE Transactions on Dependable and Secure Computing*, p. 1–18, 2023. [Online]. Available: http://dx.doi.org/10.1109/TDSC.2023.3300368

[19] S. Tempel, "Operating system security in the internet of things," Master's thesis, Universität Bremen, 2020.

[20] W. Zhou, L. Guan, P. Liu, and Y. Zhang, "Good motive but bad design: Why arm mpu has become an outcast in embedded systems," 2019. [Online]. Available: https://arxiv.org/abs/1908.03638

[21] C. Cowan, C. Pu, D. Maier, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, Q. Zhang, and H. Hinton, "StackGuard: Automatic adaptive detection and prevention of Buffer-Overflow attacks," Jan. 1998. [Online]. Available: https://www.usenix.org/conference/7th-usenix-security-symposium/stackguard-automatic-adaptive-detection-and-prevention

[22] T. Cloosters, D. Paaßen, J. Wang, O. Draissi, P. Jauernig, E. Stapf, L. Davi, and A.-R. Sadeghi, "Riscyrop: Automated return-oriented programming attacks on risc-v and arm64," in *Proc. of 25th International Symposium on Research in Attacks, Intrusions and Defenses (RAID 2022)*, Limassol, Cyprus, oct 2022.

[23] J. Biskup, *Security in Computing Systems.* Springer Berlin, 2008.

[24] T. Famulla, "Entwurf und implementierung von speicherschutzmechanismen für das wsn/iot betriebssystem riot-os unter verwendung der speicherschutzeinheit des cortex m4 prozessors," Bachelor's Thesis, Freie Universität Berlin, 2014.

[25] S. Tempel and T. Bruns, "Riot-police: An implementation of spatial memory safety for the riot operating system," 2020. [Online]. Available: https://arxiv.org/abs/2005.09516

[26] A. De, A. Basu, S. Ghosh, and T. Jaeger, "Hardware assisted buffer protection mechanisms for embedded risc-v," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 12, pp. 4453–4465, 2020.

[27] SiFive, Inc., "Sifive hifive1 rev b getting started guide version 1.3," Mar. 2023. [Online]. Available: https://sifive.cdn.prismic.io/sifive/6e47e321-cd6b-4419-a5e7-bfd39d9bb05f_hifive1revb-gsg-v1p3.pdf

[28] ——, "Sifive fe310-g002 manual," Sep. 2022. [Online]. Available: https://sifive.cdn.prismic.io/sifive/034760b5-ac6a-4b1c-911c-f4148bb2c4a5_fe310-g002-v1p5.pdf

[29] ——, "Errata_fe310-g002_20210408," Apr. 2021. [Online]. Available: https://sifive.cdn.prismic.io/sifive/b9b9c901-d522-4ac4-99b9-fe585ade229d_FE310_G002_errata_20210408.pdf

[30] Espressif Systems, *ESP32-C3 Technical reference manual*, Espressif Systems, 2023. [Online]. Available: https://www.espressif.com/sites/default/files/documentation/esp32-c3_technical_reference_manual_en.pdf

[31] Espressif Systems. [Online]. Available: https://www.espressif.com/en/products/sdks/esp-idf

[32] V. Herdt, D. Große, P. Pieper, and R. Drechsler, "Risc-v based virtual prototype: An extensible and configurable platform for the system-level," *Journal of Systems Architecture*, vol. 109, p. 101756, 2020. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1383762120300503

[33] B. Blischke, "cpu/riscv: Add PMP driver," GitHub Pull Request, Jun. 2023. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/19712

[34] ——, "buildsystem: Always expose CPU_RAM_BASE & SIZE flags," GitHub Pull Request, Jun. 2023. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/19746

[35] ——, "cpu/riscv: Stack guarding via PMP," GitHub Pull Request, Jul. 2023. [Online]. Available: https://github.com/RIOT-OS/RIOT/pull/19821

[36] J. L. H. David A. Patterson, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 2nd ed., ser. The Morgan Kaufmann Series in Computer Architecture and Design. Morgan Kaufmann, 2020.

Erklärung zur selbstständigen Bearbeitung

Hiermit versichere ich, dass ich die vorliegende Arbeit ohne fremde Hilfe selbständig verfasst und nur die angegebenen Hilfsmittel benutzt habe. Wörtlich oder dem Sinn nach aus anderen Werken entnommene Stellen sind unter Angabe der Quellen kenntlich gemacht.

Ort	Datum	Unterschrift im Original