A Memory Protection Strategy for Resource Constrained Devices in Safety Critical Applications

Mario Barbareschi
DIETI
University of Naples Federico II
Naples, Italy
mario.barbareschi@unina.it

Salvatore Barone
DIETI
University of Naples Federico II
Naples, Italy
salvatore.barone@unina.it

Valentina Casola
DIETI
University of Naples Federico II
Naples, Italy
valentina.casola@unina.it

Pasquale Montone
DIETI
University of Naples Federico II
Naples, Italy
p.montone@studenti.unina.it

Alberto Moriconi
DIETI
University of Naples Federico II
Naples, Italy
alberto.moriconi@unina.it

2022 6th International Conference on System Reliability and Safety (ICSRS) | 978-1-6654-7092-6/22/$31.00 ©2022 IEEE | DOI: 10.1109/ICSRS56243.2022.10067350

***Abstract*—In modern safety-related applications, software has achieved an increasingly critical role. Their safety-critical nature, however, requires special attention: industry-specific functional-safety standards guide designers, developers, integrators, and testers during all phases of the software life-cycle and the final artifacts undergo a rigorous certification process.**

In the field, it is not uncommon to find very resource-constrained devices performing real-time sensing and actuating tasks. Although these devices, typically microcontroller units, offer a rich plethora of on-chip devices for communication, sensing, and interaction with the physical world, they often have quite reduced computational capabilities, and barely provide memory protection functionalities, relying solely upon rudimentary Memory Protection Units (MPUs). In this perspective, guaranteeing fault-confinement through spatial isolation – i.e., the isolation between the memory used by each of the tasks, as mandated by in force regulations – is quite challenging.

In this paper, we present an MPU-based memory management and protection strategy that enables achieving spatial isolation in multi-application real-time operating systems (RTOS) tailored for safety-critical domains, while allowing a good degree of flexibility and combinability. Furthermore, we discuss the implementation of the proposed strategy as part of a RTOS from the industry domain, in order to provide a case-study pertaining to its actual implementation.

***Index Terms*—safety-critical systems, memory protection, real-time systems, spatial isolation, fault isolation**

I. Introduction

Critical systems – i.e., those for which a failure or malfunction can harm people, the environment, or even cause severe economic losses – are increasingly common in many application domains, including the heavy industry, the medicine field, and public transportation, just to mention a few examples.

Developing safety-critical applications requires compliance towards a number of regulations and standards, that although different between the different domains, such as electromedical, railway, aeronautics, and automotive, all resort to the widely known Software Integrity Level (SIL) concept [6], [10]. The sector regulations are aimed at use in any area where there are safety implications, and they consider that modern application design often re-uses generic software that is suitable as a basis for various applications, since it undoubtedly accelerate the development cycle.

Concerning the (re-)use of Real-Time Operating Systems (RTOSs) in safety-critical systems specifically, one particularly relevant aspect to cope with is to guarantee that faults do not propagate through the system. According to the mentioned regulations, several measures and specific techniques must be adopted to enhance reliability and safety. The International Electrotechnical Commission (IEC) 61508, for instance, strongly encourages temporal and spatial isolation as means to guarantee independence of execution and to avoid the propagation of faults [10].

The RTOS itself must provide techniques, such as priority and preemption schemes, and run-time deadline checks, that, in conjunction with static analysis tools, allows achieving temporal isolation. On the other hand, spatial isolation, i.e., the guarantee of isolation between the memory used by each task, requires hardware support. The latter, in modern computing systems, is provided by the Memory Management Unit (MMU), that has all memory references passed through itself to effectively perform virtual-memory management and memory-protection.

Full-functionalities provided by the MMU, however, may be unavailable in low-cost, low-power, and energy-efficient microprocessors, such as the increasingly widespread ARM Cortex-M family of microprocessors, that typically embed a low-resource applicant Memory Protection Unit (MPU), rather than a full-fledged MMU. Nevertheless, it is not uncommon to find low-cost and very resource-constrained devices being employed in safety-critical applications, since these units offer a rich plethora of on-chip devices for communication, sensing, and interaction with the physical world.

In this paper, we present a memory management and protection strategy that enables to achieve spatial isolation in

978-1-6654-7092-6/22/$31.00 ©2022 IEEE

multi-application real-time systems tailored for safety-critical domains, while allowing a good degree of flexibility and combinability.

Briefly, our strategy leverages the MPU for implementing memory-protection functionalities, and partitions real-time tasks in multiple applications. The latter are compiled and linked separately, and each has its isolated address space. We allow shared-memory based cooperation between tasks belonging to the same application. Conversely, the use of message-passing is mandatory. Furthermore, access to memory-mapped devices is also restricted. In order to define memory-regions, applications undergo an analysis step which purpose is to gather their memory-requirements. The latter are collected in a binary configuration file that serves to the RTOS kernel, at runtime, to properly configure the MPU each time a task is scheduled for execution.

The remainder of the paper is organized as follows. Section II reviews contributions from both the industry and the scientific literature, while Section III provides preliminary technical background. Our memory-protection strategy is discussed in full-details in Section IV, and Section V discusses its implementation as part of relevant industrial application, as a case-study. Finally, Section VI draws the conclusions.

II. Related Work

While still considered somehow a niche feature, memory protection is becoming more popular in modern RTOSs.

FreeRTOS [1] – one of the most famous open-source and adopted RTOSs – provides only a limited support to memory-protection through using the MPU. For both the ARMv7-M and ARMv8-M architectures, in facts, unprivileged tasks can access just their stack, and up to tree additional protected memory regions can be configured for it [2]. Such regions can be parameterized individually, they are assigned to tasks when the task is created, and they can be reconfigured at run time if required.

A similar approach is adopted by the eChronos RTOS [4]. By default, tasks have permission to access only to their stack, yet they can be associated to protection domains, i.e., memory regions, defined in a way that is congruent with the underlying architectural constraints. Protection domains are configured at boot in the MPU, and selectively enabled based on the currently running task. This, however, poses a limit on the number of the concurrently active regions, since the number of definable regions.

The scientific literature also provides plenty of contributions concerning memory protection strategies. In [15], for instance, the concept of *arena* is introduced to represent memory ranges and permission sets which are associated to components; it then uses a greedy heuristic to find a memory layout that satisfies the constraints and to produce a linker script.

SAFER SLOTH [8] is a member of the SLOTH family of operating system which provides enhanced memory protection capabilities; it provides multiple execution models, one of which uses traps and the MPU support to obtain an execution model similar to that of the other industrial RTOSs; however, variables shared between tasks need to be explicitly associated to multiple memory domains.

Another possible approach is to avoid entirely the need of special-purpose hardware; the Amulet [11] platform for *mHealth* research provides a compiler for an ANSI C dialect that disallows pointers and recursion, and inserts bound checks at runtime. As shown in [9], the Amulet approach can be enhanced by using the MSP430 MPU to reduce the number of compile-time and run-time checks needed; however, as this MPU is not capable of protecting from accesses below the area range, some support from the tool is needed nonetheless. The concept of combining hardware support and language feature is also explored in the Tock OS [12], [13] operating system; while in Amulet the memory safety is obtained by restricting a potentially unsafe language, Tock OS is written in Rust and exploits a combination of hardware support and language features such as built-in bound checking to achieve memory safety.

The approach we present in this paper is mostly supported by the hardware; while the C language is deemed potentially unsafe and a strong push to consider safer alternatives for safety-critical systems permeates the literature [7], in practice it is still largely used because of the vast availability of tooling and libraries, and is still considered suitable by industry regulations [5].

Our work is therefore focused on obtaining spatial isolation without the need for additional syntax and language usage restrictions.

Another important point is to minimize the need for user intervention; for some applications, all the relevant memory regions to be protected, and the sharing patterns can be identified by simply inspecting the executables; when more complex memory access patterns are needed, as shown in Section V, there is no need to write additional code but some additional configuration in XML format is needed.

III. Technical Background

In this section, we briefly provide some preliminary technical background for the rest of the paper.

We introduce the basic concepts of memory protection hardware in the context of Microcontroller Units (MCUs), with particular attention to the MPUs used in resource-constrained devices.

We also introduce two simple execution models for RTOSs, namely the “library RTOSs” and the “separated-executable RTOSs”.

A. Memory Protection Hardware

While often underestimated in embedded settings, memory protection is an important feature for functional safety and, to a lesser degree, for system security.

The typical memory model for embedded devices does not include any memory protection hardware: the Central Processing Unit (CPU) can load and store from/to any memory address without restrictions.

Modern microcontroller units often provide simple memory-protection hardware, namely MPUs, while more complex protection hardware, e.g., MMUs, are rarely found. This is due to several reasons, including the low-power requirements and the execution models of typical applications. The latter, indeed, makes virtualizing the address-space less relevant.

From the memory-protection perspective, MPUs mediates memory accesses and regulates read/write operations leveraging memory-regions and attributes, as shown in Figure 1. The complexity of such attributes varies greatly with that of the underlying hardware, encompassing cache behavior, if applicable.

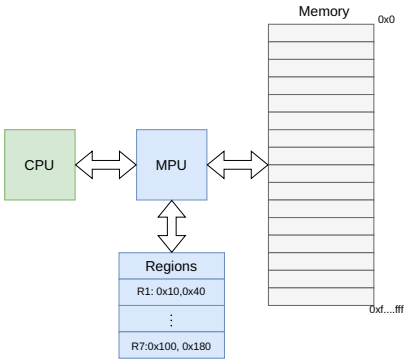


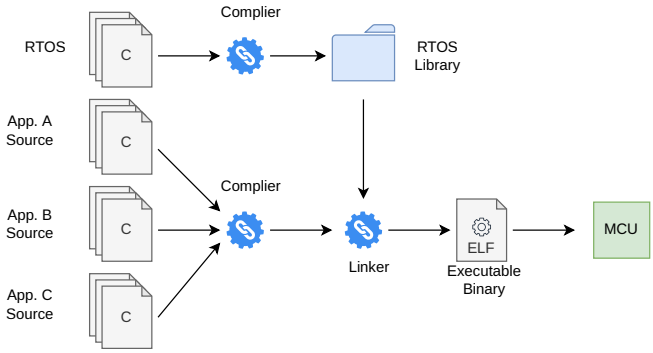
Fig. 1: Access to memory in devices with a MPU.

It's interesting to note that MPUs can be used to implement even more sophisticated schemes, as in some versions of Embedded Linux; in many systems, access to a memory region not covered by a MPU region will trigger an exception, similarly to page faults in MMU based one. In practice, this allows to overcome the limitation imposed by the small number of regions (usually between four and sixteen) in commercially available MPUs; the cost, however, is an unpredictable overhead due to the memory faults that undermines the time determinism needed in real-time systems.

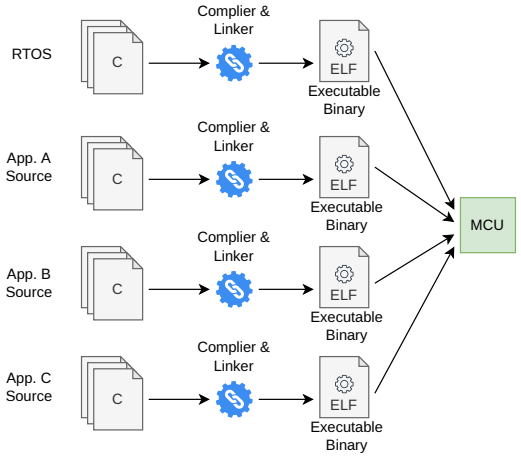
B. Real-Time Operating Systems Execution Model

The foremost objective of a RTOS is to meet the individual timing-requirements of each task. From an architectural standpoint, however, the organization of such tasks may vary greatly from system to system.

Most of the commonly used RTOSs are based on the so called "library RTOSs" architectural paradigm: the user defines the tasks with specific system calls, and assigns them a user-defined function as entry-point. The RTOS code is either compiled together with the user one, or provided as a static library and linked to the latter, providing a single binary that can then be loaded and executed on the target, as shown in Figure 2a. An important downside of this paradigm is that adding a task requires a full system recompilation; often, even a simple modification to the system configuration, such as changing the period of a task or its phase, needs a modification to some specific system call parameter and, therefore, a recompilation.



(a) Example of library RTOS



(b) Example of separated-executable RTOS

Fig. 2: Examples of RTOS execution models

A different architectural paradigm we may call "separated executables RTOS" is shown in Figure 2b. In this model, the RTOS is provided as a full, standalone binary executable, that can even be loaded and ran by itself on the target. The tasks are organized in multiple applications, based on architectural/conceptual needs, and eventually shared memory between them. Each application defines an entry point, that can be denoted as *main* – as in C programming conventions – and a number of conceptually correlated tasks, that are executed continuously, in order to perform application logic. For what pertains to the *main* function of each application, it can either define a task on its own, or be executed only once at system startup, in order to perform initialization and configuration procedures.

This model is slightly more complex than the library-RTOS one. Nevertheless, it provides benefits not limited to the conceptual and organizational perspective. Indeed, it allows adding functionalities to a system by simply adding a new application executable, and modifying the system configuration accordingly. This point is particularly important in a safety-critical context because some parts of the system may be composed of pre-existing and already certified applications, such as middlewares for communication or replication management, and being able to add them unmodified can significantly speed

up the certification process.

Using a separated executable model also lends elegantly to the definition of a memory protection scheme; while not necessarily encouraged in some programming styles, the sharing of global variables between tasks can be managed in a controlled way by only allowing it between tasks belonging to the same application.

To the best of our knowledge, RTOSs designed for resource-constrained devices such as those presented in Section II are invariably based on the library RTOS execution model. The emergence of powerful MCUs, such as the ARMv7-M and ARMv8-M families, may however motivate an interest on alternative paradigms when safety is of concern.

IV. Memory Protection in Resource Constrained Devices for Safety Critical Applications

Besides guaranteeing spatial and temporal isolation, further requirements have been taken into consideration during the development of our proposed memory-protection technique. These include managing statically allocated resources while supporting configurability w.r.t. running applications, and assuring compatibility towards position-dependent code, relying solely on hardware available in low-power, low-resource processors, e.g., ARMv7-M and ARMv8-M CPUs. In facts, the particular application domain mandates, among other things, no dynamic memory allocation after the system startup phase (at least not in user code). Furthermore, compatibility towards position-dependent code must be guaranteed, since many libraries targeting embedded systems, e.g., are distributed under this particular format. Last, the memory-virtualization mechanism usually provided by MMUs is not available, as an MPU is adopted in place of the latter in the above-mentioned low-power, low-resources processors, providing only the memory-protection functionality.

The programming model for our RTOS is based on applications and tasks: the latter are the basic unit for scheduling, while the former are sets of tasks sharing the same virtual address space. Applications, including the RTOS kernel, are compiled and linked separately; hence they independently manage their own address-space. Tasks are executed concurrently, and those belonging to the same application can cooperate through the shared-memory communication paradigm. Conversely, those belonging to different applications can communicate only using the message-passing mechanism offered by the real-time kernel. With such a kind of architectural organization, tasks can be partitioned into applications, and faults of a task can propagate only to other tasks belonging to the same application. This is true, as it is easy to foresee, whether isolation is guaranteed to data, stack, heap and to each memory-area of running applications. Furthermore, access to memory-mapped devices has to be restricted to the only applications being authorized.

A. Defining protected memory-regions

The memory-protection strategy we propose assumes (i) that protected memory regions cannot be defined at runtime, rather they have to be determined offline; (ii) that tasks execute at user level, and (iii) that the RTOS kernel is the only one managing the MPU, fulfilling its configuration each time a task is scheduled.

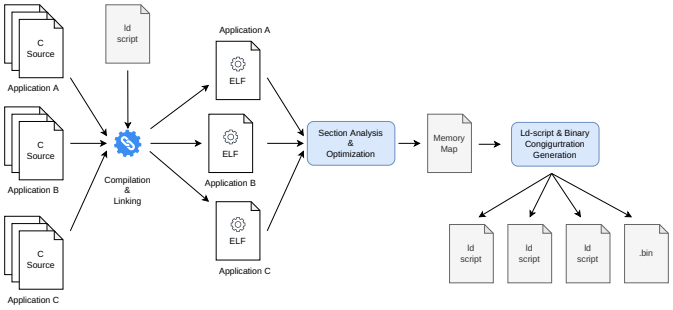


Fig. 3: Workflow for defining protected memory-regions

As depicted in Figure 3, in order to define memory-regions, applications part of the given system configuration (compiled and linked separately) undergo a preliminary analysis step, during which we inspect the following sections and their size: (i) the *text* section, where code live; (ii) the *data* section, where global tables, variables, etc. live; (iii) the *rodata* section, for read-only data; (iv) the *bss* section, which is home for uninitialized data; and, finally, for (v) the *init* and *fini* sections, which hold entry-points of initialization and finalization functions. All these sections, their size in particular, allows defining requirements of a given application, which, in turns, are needed to define protected memory-regions while optimizing for efficiency of memory usage, i.e., while minimizing the waste of memory space. In facts, when defining protected memory-regions, additional hardware limitations may lead to waste of memory space. MPUs of ARMv7-M and ARMv8-M CPUs, for instance, require the base-address to be size-aligned with the size of regions, i.e., if we consider 1KB large sections, then the 10 least significant bits of their base-addresses must be zero. Furthermore, none of the tasks should require more memory regions than those physically made available by the MPU.

This requires solving the optimization problem in which all applications belonging to the given system configuration must fit into the memory, while the waste of memory space due to the definition of protected regions must be minimized. We solve the above-mentioned optimization problem by resorting to a greedy heuristic from [14], which, briefly, sorts protected regions based on their size, and, from the largest to the smallest, assigns them a memory address. Then, we exploit the defined memory-map for the given system configuration while performing a second linking phase of applications, in order to reposition applications into memory. Furthermore, a binary configuration file is generated to be written in memory and read by the RTOS kernel at boot so that the latter can fulfill to the MPU configuration each time a task is scheduled for execution. Indeed, the RTOS kernel must be aware of the memory address-space of each application, task, and software library which are part of the given system configuration, along

with information concerning data frames, stack and heap sizes, memory-mapped devices, and so forth. Last, a loading script is also generated, in order to support device flashing.

It's important to note that, although the strategy is presented in the context of a separated-executable RTOS, similar effort is needed to implement it in a more traditional library RTOS. The main motivation for our choice is to provide a strategy that is appliable when altering existing executables is undesirable. For a library RTOS similar analysis, linker script generation and recompilation steps have to be undergone, but this time the entire system executable has to be regenerated. This is rarely a compilation time concern, as they are usually negligible for resource constrained devices, and should only be of interest in specific safety-critical scenarios.

V. An Industrial Case Study

The strategy presented in the context of this paper has been implemented in a custom hard RTOS for safety-critical applications, entirely developed from scratch. The target platform for such RTOS is the ARMv7-M based STM32H7 family of MCUs; however, the code can run with minimal modifications on many MCUs based on the Cortex-M0+/M3/M4/M7 cores, such as the STM32F/G/L families, the Tiva MCUs from Texas Instruments, the NXP K32 series, and the Microchip SAM series, as it only makes use of core facilities and no MCU-specific peripheral, the only dependence being the specific memory addressing layout.

The system has to be configured using a XML file that specifies (i) applications, (ii) their tasks, and (iii) associated resources such as inter-process communication facilities, attached peripherals and memory regions.

As an example, in Listing 4 we configure a system with a single application named IPERF and two tasks:

- `task_iperf` implements a simple network performance server compatible with the IPERF utility [3],
- `task_eth` is an interrupt service routine that manages incoming network data.

The configuration also contains information on specific memory regions that need to be accessed by the tasks; while global data, application heap and task stacks are automatically found and configured via compiled executable analysis, other buffers and peripherals need to be manually specified. For each memory region, in addition to the address and size of the region, a policy can be specified to ensure the appropriate memory system behavior; e.g.:

- cache can be disabled for specific memory regions by using the NON-CACHEABLE policy; this is particularly useful when the region need to be accessed via DMA;
- if cache is to be used, WRITE-THROUGH or WRITE-BACK behavior can be specified; DEVICE-ACCESS policy disables write buffering and reordering, ensuring the region is accessed in the same way as the parts of the address space dedicated to peripherals.

The STM32H7 MCU we target is equipped with 2 MB of flash and 1056 KB of RAM; however, the memory structure is organized in non-contiguous blocks:

```
...
<application name="iperf" elf="iperf.elf" heap
    -size="2048">
   <task xsi:type="periodic" name="task_iperf"
        stack-size="1024" priority="4" phase="
       0" period="10" deadline="0"/>
   <task xsi:type="isr" name="task_eth" stack-
       size="1024" priority="5" deadline="5"/>
</application>
...
<resource xsi:type="region" policy="non-
    cacheable" owner="iperf" name="
    RxArraySection" address="0x30042000" size=
    "8192"/>
<resource xsi:type="region" policy="write-back
    " owner="iperf" name="lwip_heap" address="
    0x30044000" size="2048"/>
<resource xsi:type="region" policy="device-
    access" owner="iperf" name="
    RxDecripSection" address="0x30040000" size
    ="128"/>
<resource xsi:type="region" policy="device-
    access" owner="iperf" name="
    TxDecripSection" address="0x30040080" size
    ="128"/>
<resource xsi:type="peripheral" name="eth"
    address="0x40028000" size="8192" user="
    task_iperf"/>
...
```

Fig. 4: Example kernel configuration

- two blocks of tightly-coupled memory, respectively 128 KB for data and 64 KB for instructions,
- three non-contiguous SRAM blocks of respectively 512 KB, 288 KB and 64 KB, that we conventionally call `RAM_D1`, `RAM_D2` and `RAM_D3`.

We reserved the tightly-coupled memory for the kernel heap and stack, the `RAM_D1` for application heaps and task stacks, the `RAM_D2` for application `data` and `bss` sections. The `RAM_D3` is not used by the automatic memory layout facilities, but by configuring them opportunely, buffers can be placed there.

At system boot, the kernel reads from the configuration binary all the informations pertaining the memory regions associated to the tasks, and stores them in their task control blocks.

During the context switch, the memory regions associated with the exiting task are removed from the MPU registers and the ones associated with the task that is becoming active are written.

VI. Conclusions

In this paper, we discussed an MPU-based memory management and protection strategy that enables achieving spatial isolation in multi-application real-time operating systems (RTOS) tailored for safety-critical domains. The strategy is designed to be well suited for very resource-constrained devices performing real-time sensing and actuating tasks, which

are commonly adopted for in-field safety-critical applications. Furthermore, we discussed the implementation of such strategy as part of a RTOS from the industry domain, while considering ARMv7-M CPUs as target devices.

Briefly, the strategy leverages the MPU for implementing memory-protection functionalities, and partitions real-time tasks in multiple applications, which are compiled and linked separately, and have their own isolated address space. The strategy allows shared-memory based cooperation between tasks belonging to the same application, while message-passing is mandatory for tasks belonging to different applications. Furthermore, our strategy also allows restricting access to memory-mapped devices. Memory-regions for the MPU are defined based on memory-requirements of applications, which are collected in a binary configuration file exploited by the RTOS kernel, at runtime, to properly configure the MPU each time a task is scheduled for execution.

REFERENCES

[1] FreeRTOS - Market leading RTOS (Real Time Operating System) for embedded systems with Internet of Things extensions. https://www.freertos.org/index.html.

[2] FreeRTOS-MPU - ARM Cortex-M3 and ARM Cortex-M4 Memory Protection Unit support in FreeRTOS. https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.html.

[3] iperf - the ultimate speed test tool for tcp, udp and sctp. https://iperf.fr.

[4] eChronos RTOS. NICTA/Data61 and Breakaway Consulting Pty. Ltd., 2017.

[5] Railway applications - Communication, signalling and processing systems - Software for railway control and protection systems. Standard, European Committee for Electrotechnical Standardization, 2020.

[6] International Electrotechnical Commision. Functional safety - Safety instrumented systems for the process industry sector. Standard, International Electrotechnical Commision, 2018.

[7] WJ Cullyer, SJ Goodenough, and Brian A Wichmann. The choice of computer languages for use in safety-critical systems. *Software Engineering Journal*, 6(2):51–58, 1991.

[8] Daniel Danner, Rainer Müller, Wolfgang Schröder-Preikschat, Wanja Hofer, and Daniel Lohmann. SAFER SLOTH: Efficient, hardware-tailored memory protection. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 37–48, April 2014.

[9] Taylor Hardin, Ryan Scott, Patrick Proctor, Josiah Hester, Jacob Sorber, and David Kotz. Application memory isolation on {ultra-Low-power}{MCUs}. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 127–132, 2018.

[10] Functional safety of electrical/electronic/programmable electronic safety-related systems. Standard, International Electrotechnical Commission, 2010.

[11] David Kotz. Amulet: An open-source wrist-worn platform for mHealth research and education. In *2019 11th International Conference on Communication Systems Networks (COMSNETS)*, pages 891–897, January 2019.

[12] Amit Levy, Bradford Campbell, Branden Ghena, Daniel B. Giffin, Pat Pannuto, Prabal Dutta, and Philip Levis. Multiprogramming a 64kB Computer Safely and Efficiently. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 234–251, Shanghai China, October 2017. ACM.

[13] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. The Case for Writing a Kernel in Rust. In *Proceedings of the 8th Asia-Pacific Workshop on Systems*, pages 1–7, Mumbai India, September 2017. ACM.

[14] Silvano Martello and Paolo Toth. *Knapsack Problems: Algorithms and Computer Implementations*. John Wiley & Sons, Inc., 1990.

[15] Runyu Pan, Gregor Peach, Yuxin Ren, and Gabriel Parmer. Predictable Virtualization on Memory Protection Unit-Based Microcontrollers. In *2018 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 62–74, April 2018.