Nested compartmentalisation for constrained devices

Nicolas Dejon, Chrystel Gaber, Gilles Grimaud

Nested compartmentalisation for constrained devices

Nicolas Dejon
Orange Labs
Châtillon, France
Univ. Lille, CNRS, Centrale Lille, UMR 9189
CRIStAL - Centre de Recherche en Informatique
Signal et Automatique de Lille,
F-59000 Lille, France
nicolas.dejon@orange.com

Chrystel Gaber
Orange Labs
Châtillon, France
chrystel.gaber@orange.com

Gilles Grimaud
Univ. Lille, CNRS, Centrale Lille, UMR 9189
CRIStAL - Centre de Recherche en Informatique
Signal et Automatique de Lille,
F-59000 Lille, France
gilles.grimaud@univ-lille.fr

***Abstract*— This paper presents a framework and implementation guidelines to set up nested compartmentalisation in constrained devices. All memory spaces are protected by the Memory Protection Unit (MPU). Current MPU-based systems offer efficient memory protection but are mostly tied to the fixed permission model provided by their operating system, kernel, hypervisor or by code instrumentation. New use cases evolve with the rise of the Internet of Things (IoT) ecosystems where software components could benefit from locally and dynamically established permissions. This includes a temporary nested subspace with restricted memory access rights. Our framework integrates subspace creation and management for runtime dynamic changes of the permission model for any level of abstraction. Global security policies of fixed permission models are reflected in the software architecture and the implementation of the framework. We also demonstrate the feasibility of providing nested compartmentalisation by showing how to leverage the MPU features.**

***Keywords*-nested compartmentalisation, constrained devices, MPU**

I. Introduction

The rise of the Internet of Things (IoT) gives birth to increasing and novel use cases invested in various IoT verticals (smart grid, smart agriculture, smart city...). IoT devices are very heterogeneous, ranging from very constrained sensors to heavier embedded systems, in order to deal with all the new and various business opportunities. For instance, some new use cases incorporate the Cloud-to-Thing Continuum [1] and increased virtualisation [2] for the smallest devices to make the IoT device simpler to code, more portable, easier to deploy and in the end cheaper to produce with increased efficiency for the ecosystem.

However, constrained devices [3] are also very vulnerable to attacks when exposed to the outside world due to their limited resources and protection mechanisms. This is decoupled in IoT devices since they are connected and thus exposed to distant attackers. As many IoT devices are cyber-physical systems, it could end with disastrous consequences to the environment or with human damage. IoT cloud applications might as well receive erroneous data misleading decision makers. IoT cyberattacks may also lead to a snowball effect in the ecosystem and hit critical infrastructures by interfering with neighbour systems through lateral compromission [4] and with distant systems through Distributed Denial of Service (DDoS) attacks [5], [6].

Business opportunities and security challenges stress the development of IoT devices which are in stringent need to be supported. In this paper, we propose a framework which copes with the software dynamism requested by newer use cases, while at the same time protecting all software components' memory spaces by hardware protection mechanisms. We base our framework on the Memory Protection Unit (MPU) or vendor-specific alternatives available on low-end embedded systems which role is to protect memory regions. We propose a technical implementation of the framework to demonstrate the feasibility of nested compartmentalisation with an MPU without requiring hardware modifications. This allows to set up isolated subspaces where each software component can manage its own memory space like a separation kernel [7].

Paper structure Section II presents our target scenario and the current status of MPU-protected systems in constrained devices. Based on the conclusion that state-of-the-art solutions are not generic enough to address the targeted scenario, we propose in Section III a framework and its MPU-based API that dynamically set up any number of subspaces during runtime. Section IV leverages the MPU features and demonstrates the feasibility of setting up nested compartmentalisation with an MPU. We expose some identified limitations that we discuss in Section V. The paper ends with conclusion remarks in Section VI.

II. MPU-based memory protection in constrained devices: target scenario and current status

A. Target scenario

A running constrained IoT device would like to set up an isolated memory area in its own memory space, so to create a protected subspace. For example, it might be a third-party application dynamically downloaded and installed on a device, that has a doubtful origin or, in the contrary, a critical feature that must remain sandboxed. Thus, we want to control the subspace's memory accesses to protect it from unwanted external interferences or to block any of its attempts to access memory it doesn't own.

The target scenario considers a constrained IoT device (at least of type II in the IETF terminology [3]) with an MPU.

B. Attacker model

We consider a powerful distant attacker who can install software components on an IoT device at reach. The adversary tries to access memory out of its memory space. Our attacker is nevertheless not able to perform any fault attacks from a distant location because we need integrity of the hardware components and we assume they are functionally correct and fully operational and thus physical attacks are out of scope. Side-channel attacks are also ruled out.

C. The Memory Protection Unit or MPU

The Memory Protection Unit (MPU) [8] is an optional component on many ARM Cortex-M processors. Its role is to restrict memory accesses according to a defined configuration. The configuration consists in a set of memory blocks called MPU regions having various permission rights (read, write, execute) and additional attributes (caching, buffering...). That is, a system usually has a default memory map that can be changed by configuring the MPU and each running process must stick to the active MPU configuration. A faulty memory access ends in a memory fault.

Systems protected by an MPU offer a higher level of security compared to their counterpart without MPU or not using it. Naturally, this goes along with a proper MPU configuration and self-protections in order to set up the protection measures as intended.

From a broad perspective, the MPU is for small embedded systems what the Memory Management Unit (MMU) is for general-purpose systems, since both share the memory protection role in a similar fashion. However, the MPU is much more constraint and systems with MPU support up to 16 MPU regions compared to millions of memory pages for an MMU.

D. State-of-the-art of MPU-based systems

We study in this section current MPU-based systems publicly available for application developers and which security characteristics are actively studied [9]–[11]. We do not include any bare-metal systems as they are tied to their platform while we consider wider applicable solutions. Hence, software components on the studied systems are passively protected by the memory protection mechanisms generally set up by the operating system, kernel, hypervisor or directly instrumented in the code. The study consisted in technical documentation and source code readings.

For our target device classes, memory protection is provided by the MPU or equivalent units. We set aside systems with only TrustZone [12] protection on purpose, since they only provide two protected areas while we strive for many more. While the protection mechanisms are most of the time use case specific, some criteria are shared between the systems. We differentiate them based on their runtime dynamism and flexibility (categorised in Table I), hardware configuration and exposed limitations due to design choices.

1) Static systems: Some systems only configure the MPU once and for all, so that the MPU offers a fixed segmentation throughout the device's lifetime, limited by the number of MPU regions. These systems benefit from a global protection mechanism. This is proposed by TrustLite [13] that configures each MPU protected trustlet at bootime and never touches the MPU configuration again.

2) Dynamic systems: Other systems permit instead a local memory protection at process or function level. The MPU is then dedicated to protect one software component at a time and enables many more protected domains by reconfiguring the MPU each time there is a context switch. The permission model can be either immutable (generally fixed at compile-time) or mutable (changed during runtime). Only a few systems also allow memory extension to their protection domains.

For dynamic systems with immutable permission models, the MPU configuration is fixed for each software component before boot time. However, the MPU is constantly reconfigured when a new protected domain is loaded, like with a new process. This is the case for many systems like μVisor, ACES, MINION, EwoK, Choupi-OS, TockOS [11], [14]–[18]. Most of the systems fall into this category and allow no user configuration. Their inherent differences on the isolation level, guarantees and security policies they enforce are translated in as many different ways to configure the MPU.

For dynamic systems with mutable permission models, the MPU configuration is not decided at boot time yet. For instance, FreeRTOS-MPU [19] and Zephyr (MPU) [20] offer some runtime defined regions that user applications or threads can configure. However, this is only a partial user configuration because of the limited number of user regions available. Furthermore, this feature can cause serious security issues if badly configured by the user.

For the vast majority of dynamic systems, software components are given a fixed size protection domain that cannot be extended at runtime. Exception is made with TockOS, but is very limited, that can dynamically allocate a bit of a process' memory by enabling subregions. These are initially disabled when the process is initialized. Of course, previously discussed customizable regions in FreeRTOS-MPU and Zephyr (MPU) can be seen as extension in addition to access permission changes, but are again limited in number.

3) Flexible systems: Flexible systems allow another dimension of adaptation to new use cases. Previously mentioned systems are not flexible in this sense. Indeed, software components are fully tied to their protection mechanism's set-up, which is based on the different abstraction layers (mainly flat isolation) they consider. There is no way to configure the MPU differently beyond the attribution scheme defined by the underlying operating system or instrumentation. That is, the MPU configuration is usually pre-defined with at least some reserved registers for a specific use.

4) Hardware base: The MPU architecture also dictates how the MPU can be used.

First, constraints differ on versions. The ARMv7 version embeds constraints like the multiple of the region size align-

ment and that the region size must be a power of two that the more recent ARMv8 version does not have.

Second, two systems based on the same MPU architecture may not use the MPU features and constraints in the same manner and for the same purposes. For example, the ARMv7 architecture integrates subregions of equal size and independently enabled, and allows MPU region overlapping. This allows many combination for systems heavily relying on subregions for their protection domains, like TockOS and EwoK. Indeed, they both reconfigure the MPU at context switch to match the current process or application. However, TockOS loads one process at a time in its MPU configuration and uses the subregions for a process' inner workings, like expanding its memory or granting accesses to peripherals. In the contrary, EwoK makes coexist several applications at a given moment in the MPU configuration, only discriminating the active application by enabling associated subregions while the remaining disabled subregions contain the other applications.

5) Limitations and trade-offs: The MPU hardware constraints limit the memory protection mechanism to be used by system designers. The trade-off between hardware constraints, security and performances has often to be assessed manually even with some automated parts in the design pipeline.

Some systems had to rework their idealised protection to fit to the hardware constraints like ACES and MINION scaling down the final number of protection domains which may lead to less security.

For others systems like EwoK, no trade-off with security is allowed and the MPU is just enough to protect what is craved by their use case. However, any use case requiring more than the number of MPU subregions is not qualified for the protection offered by this kernel. This also shows how dependent some systems are with the underlying hardware.

Furthermore, most of these systems provide protection at some level of inner abstraction, like a process or thread, and are not protecting heterogeneous components.

Finally, some systems like TrustLite chose to modify the MPU and made it *execution-aware* in order to strengthen security but still have other limitations like a static runtime configuration.

Each system tailored their memory protection at best for a specific use case which is translated in different use of the MPU. As our target scenario requires another level of abstraction than the current systems are dealing with, we cannot directly reuse one of the operating systems nor their memory protection mechanisms without deeply modifying them.

If we chose to do so anyway, this new adapted system would suffer from the same inconveniences we find in current solutions: a specialised solution for few use cases, not deeply customizable and not reusable directly for other systems that are not similar, requiring users to understand concepts specific to the use case, as well as asking them to understand how the MPU is used and what protection it really offers, likewise the eventual hardware modifications we had to set up, and possibly trade-offs we could have made to stick to the hardware limitations and finally what choice made us eventually drop part of the idealised security solution by grouping isolated domains together.

Instead, we explore in the next section a generic approach that lets any component decide to create a subspace and which will inherit the same ability. This framework is in line with our target scenario and generic enough to be reused by any systems for any level of abstraction decided at runtime, with no trade-off in terms of security and based on the MPU without any hardware modifications. That is, any abstraction can leverage the hardware protection mechanisms to set up dynamical and locally decided protections.

III. Providing nested compartmentalisation with the MPU

A. Design principles

1) Ubiquitous MPU control: We argue that the MPU component is suited to reach the generic and strong protection mechanism we are looking for, without any hardware modifications.

We propose that a software component, whatever its level of abstraction (hypervisor, OS, thread, sandboxed application...), holds the ability to design its inner memory space at runtime and in particular creating and managing a sub-layer of abstraction (a subspace). In other words, any software component is empowered with the capacity to manage its memory space without following any fixed scheme of resource attribution. It can create sub-components having the same ability in a nested manner.

The dynamic attribution of resources is controlled by the MPU which is configured to reflect the component's available memory space. In particular, any system component is endowed the same rights to configure the MPU for its memory space, even if it is nested in another system component. Hence, we propose to delegate the power to control the MPU to any system component and enable them to resketch their memory space at will.

2) Centralized MPU configuration via system calls: The MPU is part of the privileged ISA (Instruction Set Architecture) which means the configuration should be done while in a privileged mode. For the software entities to reconfigure the MPU at will, it is not an option to give this level of privilege to all of them (that include not trusted components) for obvious security concerns. Thus, we propose the exclusive role of configuring the MPU is given to a single trusted entity lying in the (privileged) kernel space. Any other component should access the MPU and change the MPU configuration via system calls to this privileged entity.

In current dynamic systems, the customizable MPU reconfiguration is at most partially possible, but mostly tied to the scheduler by loading the MPU configuration associated to the current context. This proposition goes beyond these systems, letting the user decide for each component how the MPU should be configured, dynamically at runtime, and do

TABLE I
COMPARISON OF COMPARTMENTALISATION FEATURES IN MPU-BASED SYSTEMS.

OS/kernel/hypervisor/tool	*Dynamism*			*Flexibility*	
	MPU reconfiguration	Permission model	Extendable memory	Compartmentalisation nature	Nested compartmentalisation
`TrustLite`	✗	immutable	✗	process	✗
`Choupi-OS`	✓	immutable	✗	process	✗
`EwoK`	✓	immutable	✗	process	✗
`µVisor`	✓	immutable	✗	thread	✗
`ACES`	✓	immutable	✗	generic	✗
`TockOS`	✓	immutable	(✓)	process	✗
`Zephyr(MPU)`	✓	(mutable)	(✓)	thread	✗
`FreeRTOS-MPU`	✓	(mutable)	(✓)	task	✗
`proposed framework`	✓	mutable	✓	generic	✓

not assume any reserved registers for a particular use or tied with specific permission rights.

We define then an API, detailed in subsection III-B, that the privileged MPU configuration entity should implement. This API exposes the system calls that create and manage the subspace to set up a local permission model.

3) Customized security policy: The components should also be restricted in their use of the MPU reconfiguration, otherwise it would be the same thing as giving them all the privileges. For example, malicious components could attack other components by expanding their own memory and by giving themselves more access rights than originally given. This means there should be a global restriction associated to the local permissions. Our proposition is then not only locally flexible, but at the same time globally restricted by a security policy. For example, one component could locally create a sub-component and give it some code and data for processing, while respecting a global strict rule to not add any supplementary memory access permissions to the given data. The global security policy is then reflected in each memory space definition and embedded within the implementation of the API.

In current systems, this global security policy would be reflected in their valid static configurations and pre-defined use of the MPU registers. However, we let the components decide themselves how they organise their memory in order to free them from the frozen MPU segmentations or limitations.

B. Dynamic and flexible MPU-based API

This proposition aims to extend the dynamic MPU reconfiguration of current solutions and enhance their flexibility. Our API is both dynamic and flexible.

1) Dynamic API: Reconfiguration of the MPU registers at runtime is already present in most dynamic solutions. It usually protects a specific process and the MPU is reconfigured at context switch. This allows to overcome the limitation of the MPU regions number with registers recycled to be used with other processes instead of making all processes coexist at the same time. However, we are interested in extending the memory of the processes during runtime to cope with their needs like adding memory if the process runs short of memory space or giving temporary access to a shared peripheral. There should be sufficient latitude to be independent of the number of MPU regions which limit current possibilities. This marks a significant difference with solutions choosing to combine MPU regions in a last effort to stick to the hardware constraint with a trade-off made on security, while we strive to be as close as possible of the ideal security solution. Memory blocks should then be extendable and modifiable.

The API integrates six system calls for dynamism:

- `add, remove`: these system calls extend, respectively restrict, the memory space of a software component. They can be used to move around memory chunks especially to sandbox some code or convey messages.
- `prepare, collect`: these system calls are for management. They are supposed to be called respectively before adding memory and after removing memory in order to split the act of extending memory from the management operations that make it possible.
- `cut, merge`: these system calls are specific to systems that don't have pre-defined chunks of memory, like in the case of MPU-based systems. As our target scenario is dynamic, memory blocks are expected to grow and shrink as required by the application. Cutting and merging blocks permit to redefine the memory blocks attributed to a software component.

2) Flexible API: We propose a major enhancement of the current solutions in terms of flexibility. Our target scenario does not require to know how many layers or enclaves at compile and boot-time in order to get the extra layer of abstraction when needed. This ability is not offered by any system we are aware of. Indeed, they all limit their abstractions to the hardware protection mechanisms like the limited number of MPU regions which combination is not flexible enough to nest any number of protection layers. Only one sub-layer of eight equally sized subregions is allowed in ARMv7 MPU.

The API integrates two system calls to enhance the flexibility and to create a subspace:

- `create, delete`: these system calls respectively create and destroy a subspace. The creation consists of declaring the new subspace. The destruction of this sublayer sets back any memory blocks given to the subspace into the current memory space.

This enables to break the flat memory model and, to the best of our knowledge, goes beyond the current MPU configuration use that only enables an arbitrary number of static layers (usually one to three) and cloned memory attribution schemes.

In the next section, we detail how the MPU features are leveraged to implement this API.

IV. Technical implementation by leveraging the MPU features and design choices

This section presents the design choices which have conducted the system calls' implementation and we discuss each procedure's complexity, as well as the use of the MPU and required structures to set up a memory space and subspaces.

A. Overview

A software component's memory space is composed of a list of memory blocks, each configured as MPU region to protect. A memory space can be composed of more memory blocks than available MPU regions and as such we propose a virtualisation of the MPU (but still no virtual memory as in the MMU). As such, we created a blocks selection algorithm to pick up which blocks should be configured in the MPU at a given moment. The memory space definitions are also isolated from the userland components to protect their integrity, as well as the system call procedures and data.

The design choices are evaluated in terms of complexity and desired behaviour. Sharing and cutting memory should be as fast as possible so all system calls are tailored to allow this.

B. Memory space definition with `create` and `delete`

It merely consists in setting up the structures to hold a memory space definition.

1) MPU protection of the memory blocks: A software component has a set of memory blocks which are protected by the MPU. Each block is configured in a corresponding MPU region which holds the block's start and end addresses as well as its permission access rights. The MPU architecture dictates some hardware constraints (like size and alignment) that are discussed later.

2) Protection of the memory space definition: The memory space definition should not be directly accessible by the software components but only by the privileged entity via the system calls. Otherwise, the userland components could change theirs or others' memory space definition like changing the access rights or expanding their memory.

The MPU can achieve this requirement by several means. First, these regions could stay enabled in the MPU but with modified permissions for privileged accesses only. Userland components lose access to these regions. Second, the MPU can be configured with the background region enabled. The background region roughly gives a privileged access to all memory not configured in enabled MPU regions. Hence two means: either by disabling the regions containing the definitions or by pulling out these regions from the MPU. Since the MPU regions are limited, we chose the last option and configure the MPU with only enabled regions so that no MPU regions are wasted with disabled regions. It can be noted that the same means can be obtained by disabling the MPU and without enabling the default memory map when the kernel code executes. However, keeping the MPU always enable frees us from the risks of not enabling it when it should be.

In any case, we don't know a priori which memory blocks are memory space definitions since the API is generic and can transform any memory block into a definition when creating a subspace. In fact, they are first enabled in the MPU before becoming definitions and set inaccessible. We must then create a selection algorithm that picks up the accessible memory blocks and reconfigures the MPU accordingly when this happens. The algorithm is presented in subsection IV-C.

3) More memory blocks than available MPU regions: A software component's memory space will mostly be composed of code, data and peripherals memory blocks. A typical eight-region MPU is sufficient to hold these enabled memory blocks (since we pull out from the MPU the disabled regions).

However, code and data could be fragmented over the whole memory and require several memory blocks for the memory space to be consistent. Also, the memory space holds some blocks that are part of a subspace. These may be temporary blocks while performing a subspace memory addition or permanent blocks like shared memory with subspaces. Thus, the number of memory blocks for a memory space depends on how well fragmentation is reduced, what amount of shared blocks is necessary and temporary situations. For these reasons, an eight-region MPU may not be enough. As we don't want to reject any system with only eight MPU regions, we chose to virtualise the MPU regions, i.e. a memory space definition might hold more memory blocks than available MPU regions. We enhance the previously mentioned selection algorithm with a larger set of memory blocks to choose from.

4) List of memory blocks: The memory blocks are packed together (enabled and disabled blocks) in a common list. There is one list per memory space. Enabled and disabled blocks are not split in different lists as only enabled blocks are really configured in the MPU so there are no security consequences. In addition to that, having split lists implies a block copy between the lists, as they become enabled or inaccessible, which we are glad to avoid. Finally, we could also have chosen to forbid block sharing by default, i.e. when a memory space shares a memory block with a subspace, it would loose instead the access to this block. Thus, the memory space could satisfy itself with its own blocks, there is no need for more enabled blocks than MPU regions. However, there would still subsist the temporary state when the current memory space forges blocks to subspaces out of its own with the `cut` system call. So if the memory space needs all MPU regions for itself, it would still need additional blocks for the subspace management, getting back to the problematic of dealing with extra blocks than at disposal by the MPU. And we consider this feature essential for performances, so blocks are still enabled by default when sharing.

We considered first to link the blocks in a sorted tree structure in order to speed up the retrieval of blocks in the system calls. But this would encapsulate each block in a wrapped structure containing the pointers to chain the tree together. This requires memory taken from the block which needs to be protected (usually a minimum MPU region size of 32 bytes) and a heavier processing to organise the structure.

5) Performance considerations of `create` and `delete`: The creation consists in transforming a block into a memory space definition, i.e. hosting the default memory space definition. But as it should be protected, the block must be discarded from all memory spaces in the system. The overall complexity is then in `O(k)`, with `k` the number of memory spaces.

The deletion procedure consists in setting back the state before the creation, and thus is also in `O(k)`.

The subspace creation and respective deletion don't need to be fast operations and the complexities are then reasonable.

C. Blocks selection algorithm

The blocks selection algorithm configures the MPU according to the list of memory blocks. It only picks enabled blocks. The MPU configuration holds in the MPU registers and not in RAM as with the MMU. When changing memory space and thus the current context, the MPU should be reconfigured entirely as in all dynamic systems of Section II.

1) MPU virtualisation: At memory space initialisation, the MPU is populated with some of the enabled blocks chosen by the initiator. Indeed, only a subset of all memory blocks can be configured in the MPU because the number of memory blocks exceeds the number of available MPU regions. Thus, a memory fault can occur with a legitimate access if the targeted and enabled memory block is not configured in the MPU at that moment. In such case, the algorithm should look for the memory block, verify its legitimacy, and replace an MPU region with the faulted memory block. Hence, we introduce virtualisation of the MPU and the MPU acts like a cache. Memory faults here are direct consequences of the fragmentation that can be limited by the developer. To a certain extent, this feature is similar to a copy-on-write operation in Linux.

2) Complexity of the selection algorithm: When a memory fault occurs, as the access can be legitimate and since the list of memory blocks is not sorted in any ways, the algorithm must go through the whole list to find and verify the block. For a list of `n` elements, the complexity is `O(n) < N`, with `N` being an arbitrary upper bound set at compile-time.

Static permission model systems algorithms are usually in `O(1)` complexity because the whole MPU configuration is often saved in an identical structure that just needs to be loaded. When no operations changed the memory space between two context switches, it is desirable to achieve the same complexity for our algorithm. For this purpose, we store the MPU configuration (the selected subset) in a special structure in the memory space when a context switch occurs and restores the exact configuration when the current context is activated again. As only the subset is taken as reference to configure the MPU, its structure should resemble as much as possible as the MPU registers to achieve the fastest reconfiguration. Blocks in the list may then be of any shape because they will be copied in the subset structure before being configured in the MPU.

3) MPU constraints: Memory faults can also occur when the MPU constraints are not satisfied with all memory blocks. Indeed, the MPU regions usually have a minimum size of 32 bytes that can be enforced by the developer, but other constraints could be more difficult to set up like alignment. For example, in ARMv7, the MPU region must be a power of two aligned on a multiple of its size. The developer can align a memory space's blocks properly, but must always deal with blocks intended to a subspace that are first part of the current memory space. This means the current memory space may have blocks that later will be cut and shared to a subspace, fulfilling the MPU region constraints of the memory space and its subspace only after some time. In the meantime, the memory blocks are not aligned and result in memory faults. As such, in a non-aligned case, the selection algorithm extracts a subregion of a legitimate block that fulfils all constraints of the platform. If there is a memory fault for a memory block that has only been partially configured, the algorithm selects the subregion that respects all constraints and covers the faulted legitimate memory access. The MPU can then be configured with partial memory blocks and reconfigured at memory fault. Thus, there is a penalty for temporary memory blocks or badly configured ones, while there are no penalties for final blocks that have been properly configured.

4) Algorithm triggers: The selection algorithm is triggered by several operations in addition to the MPU virtualisation feature and the MPU hardware constraints.

The algorithm is called when a block is consumed to hold metadata and so becomes inaccessible due to a `create` or `prepare`. Any other operation adds new enabled blocks or makes blocks accessible again which do not invalidate the enabled blocks currently configured in the MPU. For example, adding a block to a subspace does not invalidate the current memory space MPU configuration as it is shared by default, nor the subspace's as the current selected subset stays valid. However, when the subspace would like to access the new shared block, it will end in a memory fault and trigger the selection algorithm.

D. Sharing and sharing tagging with `add`, `remove`, `prepare` and `collect`

It merely consists in tagging a memory block of the memory space's list as shared with one or more subspaces, e.g. enhancing the block's attributes with pointers to these subspaces, and copy the block in the subspace. Since we target single-core platforms, block copies in different memory spaces are never accessed simultaneously.

1) Performance considerations of `add` and `remove`: Sharing may need fast execution when conveying messages or setting up a shared memory space. The `add` and `remove` procedures are thus thought to be as fast as possible with a `O(1)` complexity.

For sharing, the block needs to be copied in the subspace, and there should be a free slot where to make the copy. Instead of going through the list of memory blocks and find a free slot, we keep a reference to this free slot in the memory space definition. Retrieving the free slot this way and making the copy perform with the desired `O(1)` complexity.

For unsharing, the block needs to be retrieved, but as it was copied in the first free slot the user has no clue where it lies in the list and so cannot just use an index to point to it. We added a reference to the index of the copied block into the original memory block. Thus, retrieving the block copy is immediate when knowing the original block. The freed slot is inserted at the head of the free slot list where it was taken from the beginning. All operations are then in `O(1)` complexity.

2) Performance considerations of `prepare` and `collect`: In order to have fast sharing and unsharing, the management of the references is separated in the `prepare` and `collect` procedures. These can be called whenever required and especially in advance so to not impact the fast sharing and unsharing procedures.

In order to always have a fresh free slot, all free slots in the memory blocks list are chained at creation of the list with `prepare`. When a free slot is used for an operation, it is pulled out of the list of free slots and the next free slot is pointed in the memory space for a future operation. The `prepare` operation requires to go through the whole list of memory blocks and hence a `O(n)` complexity, with `n` being the size of the list of memory blocks.

The reverse operation with the `collect` procedure scans the memory blocks list (so again in `O(n)`) to verify its emptiness and is responsible to set back the memory space to the state before the previous `prepare`.

E. Cutting and cutting tagging with `cut` and `merge`

Cutting a block splits it in two subblocks of size determined by the location of the cut. Tagging cut subblocks merely consists in enhancing the subblocks' definition with a reference to the original block and links the two created sibbling subblocks.

1) Performance considerations of `cut` and `merge`: As `cut` and `merge` are operations on the current memory space only, there are expected to be fast.

They can be seen as `add` and `remove` in the same memory space, without consequences in the subspaces, and so also have a complexity of `O(1)`. However, as a cut subblock is expected to be used as memory space definitions for subspaces, as soon as they are cut, the blocks are set inaccessible in the ancestor memory spaces. The overall complexity is then `O(p)`, with `p` being the number of levels of subspaces until the current one. But the number of subspace levels is not expected to be higher than 3 or 4 which sets the upper bound.

F. Protection of the privileged MPU responsible entity

The privileged entity implementing the API should never be userland accessible which could harm its integrity.

The involved protection mechanisms of this entity can then be similar to the protection of the memory space definition. For the same reasons as exposed earlier, the entity's code and data sections are only accessible as part of the MPU background region and are never configured in the MPU registers.

Whenever there is a system call, the system shifts to the privileged mode and thus all system call procedures are available for the privileged entity. At the end of the system call, the system shifts back to the user mode. Hence, the privileged entity's memory space is never configured in the MPU nor visible for the software components at any time.

Of course, the privileged entity is exposed to the other entities in the kernel space, one of which might be the OS. As for any security critical features, it depends on the trusted components.

V. Discussion and limitations of the approach

We presented an implementation of the API to show the MPU can be used to create MPU protected nested subspaces.

Current protection mechanisms are efficient but not flexible to deal with use cases requiring more protected domains with runtime flexibility requirements. Even if compatible, this generic approach may not be relevant for many of the current use cases as they just define a global security policy. However, we argue that the possibility to have a dynamic local permission model could enhance their features or will leverage new use cases. Furthermore, the framework does not imply any hardware modifications. In a broader perspective, the nested compartmentalisation goes beyond other systems' architectures based on flat or limited nested memory protections, e.g. with Intel SGX enclaves [21] or with TrustZone.

In addition to that, memory spaces can hold more blocks than available MPU regions. This enhanced flexibility never engages security compared to the trade-off present in some other MPU-based systems. Compatible security policies may include the least privilege principle, kernel self-protection, and temporary isolated memory spaces, which we will illustrate in future works.

The system calls' complexities match expectations for nominal operations in embedded systems. However, all operations depend on the user inputs that could be invalid. An additional checking phase adds up some processing and penalizes the overall complexity for both sharing and unsharing to `O(n)`, where `n` is the number of memory blocks in a memory space, upper bounded with `N` the maximum allowed number of blocks. This is not an issue for our target scenario, like many others primarily interested in better security, but we acknowledge it could be critical for some use cases. Benchmarks and detailed expected performances are intended in future works.

The MPU virtualisation implies memory faults on legitimate memory blocks that ideally should be avoided. As a matter of fact, the *MPU cache* strategy follows a random replacement and may not be called at efficient moments. If this strategy shows considerable downsides, we aim to make the selection algorithm callable and customizable by the developer. Indeed, the developer knows the best which MPU configuration is the most efficient at some point in the system's lifetime and would match as closely as possible the *perfect* cache policy.

One rigid limitation is of course the availability of the MPU, which is an optional unit. Many reasons let developers put aside the MPU even when there is one, should it be because of the hardware constraints, the power it drains or the time-to-market pressure and lack of time to set it up [22]–[24]. However, we argue the protection it provides is sufficient to

reach a high level of security when correctly configured and our proposition aims to ease its adoption. For current systems, like the ones studied in Section II, some non-trivial changes are required to pass over to the proposed generic approach. Future works encompass the modifications needed to adapt an OS and assess the complexity of the task.

Our framework and proposed implementation do not depend on the MPU version. Indeed, the framework is designed for and encapsulates both ARMv7 and ARMv8 MPU programmer models, but is extendable to similar hardware components like RISC-V Physical Memory Protection (PMP). Many of the studied systems did not have at development time the most recent ARMv8 version enhanced with the TrustZone and more MPU regions, and which releases some of the hardware constraints of the previous MPU version. However, the number of subspaces that can be established will still be limited with a raw usage while our framework only requires memory blocks protection with constraints absorbed in the implementation. Furthermore, ARM-powered systems and their MPUs are broadly used for embedded systems and the trend shows an increase of IoT devices in the next years [25]. As vulnerabilities will certainly continue to strike the ecosystem in the near future, our proposition is in line with an immediate hardening of deployed constrained devices.

VI. Conclusion

New use cases upset the small embedded devices, requiring more complexity and dynamism than ever before, carried along with increasing IoT devices, interconnectivity and installed verticals. Creation of dynamically defined protected subspaces is one of these use cases but are not compatible with current systems without deeply adapting them. We propose a framework which permits any number of nested compartments. These could be seen as implicit privilege levels. In order to address constrained devices, the framework is based on the memory protection provided by the MPU without requiring hardware modifications. The security harness is built by centralising the MPU configuration role to a sole privileged entity. To the best of our knowledge, the proposed framework enables a degree of flexibility not yet reached by any existing systems.

We then present a possible technical implementation of the framework. We explain how to leverage the MPU features to create the protected nested compartmentalisation. We provide an MPU virtualisation in order to manipulate more memory blocks than MPU regions. We also discuss the design choices that conducted the implementation and guided the performances of each of the framework's system calls. This demonstrates the feasibility of providing nested compartmentalisation with the MPU. Any software component can then locally control and isolate its subspaces as it intends, while conforming to a global security policy integrated in the implementation and in the chosen software architecture.

References

[1] K. M. Giannoutakis, M. Spanopoulos-Karalexidis, C. K. F. Papadopoulos, and D. Tzovaras, "Next generation cloud architectures," *The Cloud-to-Thing Continuum*, p. 23, 2020.

[2] K. Ogawa, H. Sekine, K. Kanai, K. Nakamura, H. Kanemitsu, J. Katto, and H. Nakazato, "Performance evaluations of iot device virtualization for efficient resource utilization," *Global IoT Summit, GIoTS 2019 - Proceedings*, pp. 2019–2022, 2019.

[3] IETF , "Website of : Terminology for constrained-node networks (ietf)," https://tools.ietf.org/rfc/rfc7228.txt, 2020, [Online; accessed November 20, 2020].

[4] Kellen Beck , "Website of: Hackers exploit casino's smart thermometer to steal database info (mashable)," https://mashable.com/2018/04/15/casino-smart-thermometer-hacked/?europe=true, 2018, [Online; accessed December 10, 2020].

[5] Krebs on Security, "Website of: Ddos on dyn impacts twitter, spotify, reddit," https://krebsonsecurity.com/2016/10/ddos-on-dyn-impacts-twitter-spotify-reddit/, 2016, [Online; accessed March 27, 2020].

[6] Ministry of Electronics and Information Technology of India , "Website of: Mozi iot botnet," https://www.cyberswachhtakendra.gov.in/alerts/MoziIoTBotnet.html/, 2020, [Online; accessed March 09, 2021].

[7] J. M. Rushby, "Design and verification of secure systems," *Proceedings of the 8th ACM Symposium on Operating Systems Principles, SOSP 1981*, vol. 15, no. 5, pp. 12–21, 1981.

[8] ARM , "Website of: Armv8-m memory protection unit version 2.0," https://static.docs.arm.com/100699/0200/armv8m_memory_protection_unit_100699_0200_en.pdf/, 2017, [Online; accessed March 09, 2021].

[9] A. Sensaoui, O.-E.-K. Aktouf, D. Hely, and S. Di Vito, "An In-depth Study of MPU-Based Isolation Techniques," *Journal of Hardware and Systems Security*, vol. 3, no. 4, pp. 365–381, 2019.

[10] G. Bouffard and L. Gaspard, "Hardening a Java Card Virtual Machine Implementation with the MPU," 2018.

[11] A. A. Clements, N. S. Almakhdhub, S. Bagchi, and M. Payer, "ACES: Automatic compartments for embedded systems," *Proceedings of the 27th USENIX Security Symposium*, pp. 65–82, 2018.

[12] Arm Limited, "Introduction to the armv8-m architecture," 2017, version 2.0.

[13] P. Koeberl, S. Schulz, A. R. Sadeghi, and V. Varadharajan, "TrustLite: A security architecture for tiny embedded devices," *Proceedings of the 9th European Conference on Computer Systems, EuroSys 2014*, 2014.

[14] ARMmbed , "Website of: The arm mbed uvisor," https://github.com/ARMmbed/uvisor/, 2018, [Online; accessed March 09, 2021].

[15] C. H. Kim, T. Kim, H. Choi, Z. Gu, B. Lee, X. Zhang, and D. Xu, "Securing Real-Time Microcontroller Systems through Customized Memory View Switching," *NDSS*, 2018.

[16] R. Benadjila, A. Michelizza, M. Renard, P. Thierry, and P. Trebuchet, "Wookey: Designing a trusted and efficient USB device," *ACM International Conference Proceeding Series*, pp. 673–686, 2019.

[17] Guillaume Bouffard and Léo Gaspard , "Website of: Choupi os (github)," https://github.com/gavz/choupi-os/, 2020, [Online; accessed November 20, 2020].

[18] Tock development team , "Website of: Tock," https://www.tockos.org/, 2020, [Online; accessed November 20, 2020].

[19] FreeRTOS , "Website of: Freertos-mpu (freertos)," https://www.freertos.org/FreeRTOS-MPU-memory-protection-unit.htmll, 2020, [Online; accessed November 20, 2020].

[20] Zephyr Project , "Website of: The zephyr project," https://www.zephyrproject.org/, 2020, [Online; accessed November 20, 2020].

[21] Intel , "Website of: Intel® software guard extensions (intel® sgx)," https://www.intel.co.uk/content/www/uk/en/architecture-and-technology/software-guard-extensions.html/, 2021, [Online; accessed June 14, 2021].

[22] H. Xia, J. Woodruff, H. Barral, L. Esswood, A. Joannou, R. Kovacsics, D. Chisnall, M. Roe, B. Davis, E. Napierala, J. Baldwin, K. Gudka, P. G. Neumann, A. Richardson, S. W. Moore, and R. N. Watson, "CheriRTOS: A Capability Model for Embedded Devices," *Proceedings - 2018 IEEE 36th International Conference on Computer Design, ICCD 2018*, pp. 92–99, 2019.

[23] EETimes, "2019 Embedded Markets Study," 2019.

[24] E. Foundation, "IoT Developer Survey 2019 Results," no. April, 2019. [Online]. Available: https://iot.eclipse.org/resources/iot-developer-survey/iot-developer-survey-2019.pdf?sc_icampaign=pac_iot-developer-report&sc_ichannel=ha&sc_icontent=awssm-2530&sc_iplace=banner&trk=ha_awssm-2530

[25] P. Sparks, "The route to a trillion devices The outlook for IoT investment to 2035," *ARM Whitepaper*, pp. 1–14, 2017.