Pip-MPU: Formal verification of an MPU-based separation kernel for constrained devices

Nicolas Dejon, Chrystel Gaber, Gilles Grimaud

PIP-MPU: FORMAL VERIFICATION OF AN MPU-BASED SEPARATION KERNEL FOR CONSTRAINED DEVICES

Nicolas Dejon[1,2], Chrystel Gaber[1] and Gilles Grimaud[2]

[1]Orange Labs, Châtillon, France
[2]Univ. Lille, CNRS, Centrale Lille, UMR 9189 CRIStAL - Centre de Recherche en Informatique Signal et Automatique de Lille, F-59000 Lille, France

ABSTRACT

Pip-MPU is a minimalist separation kernel for constrained devices (scarce memory and power resources). In this work, we demonstrate high-assurance of Pip-MPU's isolation property through formal verification. Pip-MPU offers user-defined on-demand multiple isolation levels guarded by the Memory Protection Unit (MPU). Pip-MPU derives from the Pip protokernel, with a full code refactoring to adapt to the constrained environment and targets equivalent security properties. The proofs verify that the memory blocks loaded in the MPU adhere to the global partition tree model. We provide the basis of the MPU formalisation and the demonstration of the formal verification strategy on two representative kernel services. The publicly released proofs have been implemented and checked using the Coq Proof Assistant for three kernel services, representing around 10000 lines of proof. To our knowledge, this is the first formal verification of an MPU based separation kernel. The verification process helped discover a critical isolation-related bug.

KEYWORDS

Separation Kernel, Constrained Devices, MPU, Formal Verification, Isolation Property

1. INTRODUCTION

Separation kernels simulate a physically distributed environment on a single machine [1]. Their isolation property is crucial to the security of the system. Greater assurance is given to hardware based solutions supported by formal methods *i.e.* with mathematical foundations. In this paper, we target Pip-MPU, a formally verified separation kernel for constrained devices (scarce memory and power resources) based on the Memory Protection Unit (MPU). The MPU provides protected memory regions configurable at runtime by privileged components. However, a wrongly configured MPU implies breaches that attackers can exploit to steal and corrupt data of expected isolated software entities, which could lead to full control of the device.

The goal of this work is to engage in the formal verification of Pip-MPU's isolation property, since Pip-MPU's high-level design and conceptual foundations have been introduced in earlier works [2, 3]. Indeed, the motivation follows recent increases in cyber-threats on constrained devices (also referred to as low-end devices, of type microcontroller) and especially connected devices. Regulatory incentives add up to cybersecurity standards developed in recent years to increase the global cybersecurity level in computer products [4-6]. To the best of our knowledge, no MPU-based isolation solution has been formally verified before. We detail the formalisation and the intuition of the proofs. The corresponding formal proofs are conducted within the Coq Proof Assistant [7] and available online [8]. Our contributions are a formal basis to reason about MPU-based solutions, the demonstration of the feasibility of Pip-MPU's formal verification by

formally verifying three representative services, and we confirm the success of a proof strategy readapted for a different hardware support conducted directly on a concrete model, instead of using refinement from an abstract model.

The rest of this paper is organised as follows: Section 2 demonstrates the lack of security guarantees in current hardware-based isolation solutions and presents Pip-MPU. Section 3 details the proof workflow and explicates the proof goal of the formal verification of Pip-MPU. The proof goal is then formalised in Section 4, directly followed by Section 5 which presents the formal basis of the hardware model supporting the proofs. In Section 6, we give the sketch of proof to prove the isolation property on the two types of kernel services present in Pip-MPU. The proof development evaluation in terms of proof status and proof effort is exposed in Section 7. We discuss the results, assumptions, and identified perspectives in Section 8 before concluding the document with Section 9.

2. EMBEDDED KERNEL MPU-BASED ISOLATION

2.1. Hardware-based isolation and security guarantees

Many hardware-based isolation solutions exist for low-end embedded systems like TrustedFirmware-M [9], RIOT OS (MPU) [10], Zephyr RTOS (MPU) [11], FreeRTOS-MPU [12], EwoK [13], MINION [14], ACES [15], OPEC [16], MultiZone [17], TrustLite [18], CheriRTOS [19], MbedOS [20], TockOS [21], hardened Java Card Virtual Machine (JCVM) [22]. They show different isolation techniques that exhibit various levels of guarantees, ease of use, and functionalities.

Hybrid approaches combining hardware-based and formally proven isolation expose the strongest means of isolation. However, no solutions for constrained devices reach that level of guarantee. Indeed, the majority does not rely on formal methods to prove isolation, which is necessary to get a mathematical guarantee of the proposed solution. Also, many solutions restrict the number of protected components because of hardware limitations, numerous in constrained devices, which take them away from the ideal isolation solution. Isolation is often seen as a trade-off for performance with negotiable security guarantees. In addition to that, many solutions are tied to a specific architecture, such as Armv7-M [23]. When they do use formal methods, they do not address abstract notions such as isolation at task level, do not analyse the full Trusted Computing Base, are also tied to a specific hardware architecture implementation, and are still very limited in the number of protected components. Many also require manual intervention from the system or application developer. This is error-prone since they are almost all written in an unsafe programming language like C and risk to put the system in a dangerous state. Some take support from properties of the programming languages to increase security by avoiding certain classes of memory vulnerabilities, however, this does not address isolation from the system perspective. Indeed, hardware isolation appears stronger since it treats not the causes of the vulnerabilities but their effects. And lastly, many current systems have no means of isolation and are vulnerable to code injection attacks, privilege escalation, data theft or corruption, and system corruption.

In contrast to previous solutions, formal verification of separation kernels is a long-lasting research discipline in high-end embedded systems. High-end embedded systems have similar memory isolation hardware mechanisms to common general-purpose systems, such as the Memory Management Unit (MMU), which are well-established security mechanisms. Among them, the best known are seL4 [24, 25] (MMU-based process and kernel isolation, MMU-based process isolation, fully formally verified in Isabelle/HOL [26]) which recently announced KataOS [27] to provide full operating system features, CertiKOS-secure [28] (MMU-based process and kernel isolation, MMU-based process isolation, fully formally verified in Coq [7, 29]), ProvenCore [30] (TrustZone [31] or airgap isolation of the trusted core, MMU-based process isolation, fully formally verified in custom language SMART, Common Criteria EAL7

certification) and Pip [32] (MMU-based process and kernel isolation, MMU-based process isolation, fully formally verified in Coq).

However, the listed formally verified systems target high-end devices intimately linked to the MMU component not available for constrained devices. This difficult equation left blank the field of strong security guarantees for constrained devices, as illustrated in Table 1. To the best of our knowledge, ProvenCore-M [30, 33] is the only isolation solution to target the microcontroller sector and to expose strong guarantees of isolation by formal proofs. The major drawback is its proprietary classification, which is why few inner workings have been disclosed.

Table 1: Hardware-based solutions (OSes/kernels, tools, architectures) classified by isolation guarantees for embedded and general-purpose systems.

	Intermediate isolation guarantees	**High** isolation guarantees (supported by formal methods)
Low-end embedded devices	TrustedFirmware-M [9], RIOT OS (MPU) [10], Zephyr RTOS (MPU) [11], FreeRTOS-MPU [12], EwoK [13], MINION [14], ACES [15], OPEC [16], Multi-Zone [17], TrustLite [18], CheriRTOS [19], MbedOS [20], TockOS [21], hardened Java Card Virtual Machine [22]	ProvenCore-M [30] (proprietary)
High-end embedded devices	Windows IoT [34], Linux Embedded [35], Fuchsia [36], TrustedFirmware-A [37]	seL4 [24], mCertiKOS-secure [28], Pip [38-40], ProvenCore (proprietary) [30]

2.2. The Memory Protection Unit (MPU) and flexibility

Applications require memory to store their code and data. However, they are subject to the kernel's security policy. This means memory has to be managed and access to resources must be controlled. To this end, the majority of ARM Cortex-M-powered devices [41] (Cortex M3/4/7/0+/23/33/35P/55), which are constrained devices, have an optional processor unit called the *Memory Protection Unit (MPU)* [23,42]. It restricts memory access by configuring the system's memory layout. The configuration consists of a set of memory regions defined by base and end addresses (or region size), called **MPU regions**, having various permission rights (read, write, execute) and additional attributes (caching, buffering…). The MPU is configured at runtime by privileged software (typically an OS kernel). The number of MPU regions that can be configured and protected at the same time is implementation-defined, generally 8 to 16 MPU regions. All running processes must stick to the configuration. 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-protection in order to set up the protection measures as intended, which is the goal of the demonstrations in this paper. 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 similarly. The MPU provides very limited protected memory areas and no virtual memory.

2.3. Pip-MPU: embedded kernel for constrained devices with dynamic memory management leveraging MPU virtualisation

Pip-MPU [3] is a separation kernel for constrained devices designed to enhance the dynamism in secure embedded kernels. It is specialised in *memory isolation* and *control-flow management.* More specifically, it ensures strict spatial memory partitioning (confidentiality and data integrity) between **partitions** (executable components). The security mechanism rests on a hierarchical partitioning model guarded by the MPU (illustrated in Figure 1), separation of privilege and atomicity (hardware support to *disable interruptions during service executions*). Pip-MPU is sole privileged in the system while all the partitions run unprivileged in userland. Pip-MPU derives from the MMU-based Pip protokernel, but with complete code refactoring because of the limited features of the MPU compared to the MMU. The implemented Pip-MPU prototype for ARMv7 Cortex-M requires 10 KB of Flash, 550B of RAM and an overhead of 16% on both performance and energy consumption. Pip-MPU does not require any hardware modification on Commercial Off-The-Shelf (COTS) systems and only depends on the MPU, which implies privilege separation (privileged/unprivileged modes).

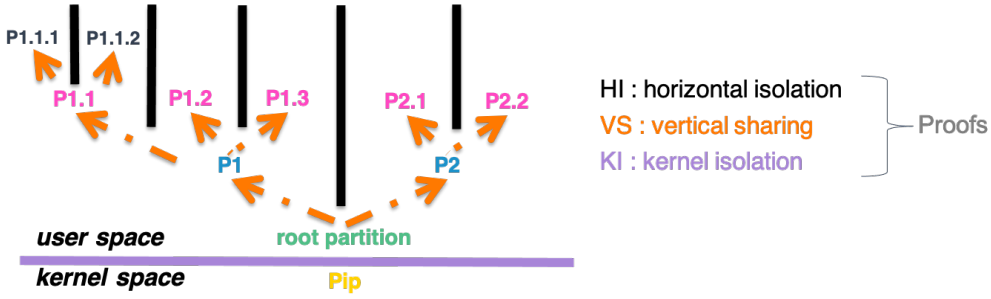

Figure 1: Pip's hierarchical partitioning model. Pip is the sole privileged component in the system.

Partitioning model. Pip-MPU's memory management is based on a **hierarchical partitioning model**. A *partition* would typically reflect a process, a task or any executable piece of code along with its respective dependent data. A partition is thus composed of one or more chunks of memory. The main principle is that a partition can create one or several subpartitions, that in turn can create subpartitions. This creates a **partition tree**, as can be seen in Figure 1, which is a sort of genealogical tree of nested memory spaces (**parent-child** relationships, **descendants**, **ancestors**, **siblings**). All partitions descend from the **root partition** which is the first partition to be created after the boot process. As an example, the root partition could be an operating system creating child processes. The other partitions are dynamically created during the system's lifetime. A parent partition can create child partitions and share memory with them. All the memory attributed to a partition is directly accessible to this partition.

Enhancing the security of RIOT OS with Pip-MPU. Pip-MPU serves as a secure foundation for bare-metal applications and richer OSes like RIOT-OS lacking strong security guarantees. They could leverage Pip-MPU's isolation property to partition their memory space and fine-grain their security policy. Indeed, our teams ported RIOT on Pip-MPU (https://github.com/damaramad/RIOT/tree/2022.04-pip-port) so a full operating system running in the root partition and which is able to interact with Pip-MPU via a dedicated library. We enabled several drivers by manually checking the peripherals do not interfere with Pip-MPU's secure configuration. We report the successful launch of the `hello-world` example.

3. PROOF GOAL: ISOLATION INVARIANT

Pip-MPU claims to provide similar services to the MMU-based Pip and to enforce Pip's security properties (see Figure 1), of which isolation properties, that have been formalised as: the Horizontal Isolation *HI* (strict isolation between sibling partitions), the Vertical Sharing *VS* (memory owned by a child partition is shared from its parent partition), and the Kernel Isolation *KI* (strict isolation between user and kernel memory). However, confidence is higher with formal proofs of the isolation property, as done in the parent project Pip [43]. Similarly, we aim to formally verify these security properties for Pip-MPU.

3.1. The security properties

The security properties are **invariants**, *i.e.* properties that must be verified whatever the system state. They extract information from Pip-MPU's metadata structures to construct the partitioning model on which they can reason about. Consequently, they can only be proven on a sane/correct partitioning model, which implies the metadata structures are correctly formed. Consistency properties ensure the latter. They check the correctness of the configuration of the metadata structures. The consistency properties are also invariants.

Pip-MPU has then two groups of expected invariants, the security properties (*HI*, *VS*, *KI*) and the consistency properties (*C*), gathered in the so-called **isolation invariant**. To be noted, the isolation invariant is an umbrella term that hides that all properties do not participate in the isolation property, strictly speaking.

3.2. Proof workflow

Proofs are always conducted on an abstract view of reality. However, the abstract level depends on the proof's methodology. Previously listed formally verified separation kernels (seL4, CertiKOS-secure, ProvenCore) rely on *refinement*, that is the verification is conducted on high-level specifications (almost 40 abstraction layers in CertiKOS [44]) that are then refined down to the implementation level. Instead, Pip (MMU)'s formal proofs are directly conducted on the source code, which corresponds to the lowest specification levels of the listed kernels, so without refinement. Thus, the abstract model is nevertheless a *concrete* model. Pip-MPU reproduces the same workflow as it facilitates the work by keeping the same formal foundations and leveraging the team's past experience in formal verification.

Overall, we have the following model, reproduced in Appendix A:

- the services update the kernel structures by using low-level primitives (called the hardware abstraction layer, **HAL**) ;
- the primitives interact with the hardware model, among which the MPU, to change the abstract system state ;
- the system state holds properties (including the security properties) verified in the proofs.

3.3. Isolation invariant

The proof goal is to verify the isolation invariant, *i.e.* the security and consistency properties. Indeed, the security of the system is established by the correct configuration of the MPU for the active partition, which effectively sets up the security properties of the real system. The link between the MPU, its correct configuration and the security properties, is made in the formal verification. Hence, the proofs aim to verify that the memory blocks loaded in the MPU adhere to the global partition tree model and that this partition tree itself satisfies Pip's security properties. The glue is the `MPUFromAccessibleBlocks` consistency property, which states that in a given partition, all blocks configured in the MPU are accessible memory blocks belonging to that partition and satisfying the security properties.

Following Pip (MMU)'s proof workflow, Pip-MPU uses Hoare logic [45] to formally verify the kernel services. Hoare logic gives a formal system to conduct the proofs of properties of a program, *i.e.* rigorous deductive reasoning rules of inference. In Hoare logic, the formal specification refers to a **Hoare triple** composed of a program *Q*, a pre-condition *P* and a post-condition *R*. The pre- and post-conditions are general assertions (properties) on the values of the program written in mathematical logic. The Hoare triple is formally expressed as $\{ P \}\ Q\ \{ R \}$. It logically expresses that if the program *Q* executes and terminates, then its post-conditions *R* are true at completion if the pre-condition *P* was true before *Q* was executed. Pre- and post-conditions might hold the same properties: these are *invariants*.

Consequently, we express the Pip-MPU's isolation invariant as independent Hoare triples for each service:

$\{VS\ \&\ HI\ \&\ KI\ \&\ C\}$
$Pip - MPU\ service\ (parameters)$
$\{VS\ \&\ HI\ \&\ KI\ \&\ C\}$

Thus, if the invariants are satisfied at the start of the execution of a service, we must prove they are still satisfied at the end of the execution. Each service has a corresponding Hoare triple that can be independently proven. Indeed, *services cannot be preempted* (exceptions are disabled) and userland operations have no influence on the kernel structures. The latter is assured by the Kernel Isolation invariant which implies the kernel structures, and thus the partitioning model, can only be updated by privileged code. And in Pip-MPU, only the kernel services execute in privileged mode. As a consequence, the kernel structures (on which are based the security properties) do not change between system calls and so the properties satisfied at the end of the execution of a service are the same when calling the next service later on, whatever happened during the execution of userland partitions. Thus, services are called one at a time, and Hoare triples are chained one after the other. The initial state satisfies the invariants because Pip-MPU prepares the initial partition correctly (assumption).

Note that with constrained single-core embedded systems, which are the targets of this study, and because the service calls are cooperative atomic operations as said previously, we avoid race condition issues faced by multi-core systems. Thus, invariants are only required to be true at the start and end of the service but are allowed to have temporary states within the execution where the invariants are not satisfied, for example during a metadata structure update.

4. FORMAL EXPRESSION OF THE SECURITY PROPERTIES

While the isolation invariant in Pip (MMU) and Pip-MPU is similar, the formal expressions differ because of structural differences relating to the memory unit. For example, the Horizontal Isolation property expressed in Pip (MMU) states that sibling partitions cannot hold the same physical pages, so their base address should be strictly different. This works because Pip is based on physical memory pages of fixed size, which exist from the system start. No subpages are referenced in the system. Instead, Pip-MPU allows cutting memory blocks arbitrarily, and so a subset of a memory block in a parent partition could be an entire memory block in a child partition. Hence, a Pip-MPU memory block identifier based on its start address, equivalent to the page identifier in Pip, would not work any more to uniquely identify a memory block. Instead of memory blocks, Pip-MPU considers a finer-grained memory unit, *the memory address*.

4.1. Note on the Kernel Isolation property

The full Kernel Isolation *KI* security property is in fact split in two: 1) the kernel code and data isolation from userland partitions and 2) the protection of the metadata structures (here named *Kernel Data Isolation* by inheritance to the parent project Pip). Only the Kernel Data Isolation property must be proven because we argue that the partitions can never access kernel code and data. Indeed, partition memory is only given by ancestor partitions originating from the root

partition. Hence, partitions only own strictly less memory than their ancestors. When Pip-MPU starts, the root partition is loaded with the system's memory space *pulled out* from kernel code and data memory regions. Thus, the only property that remains to be proven for Kernel Isolation is Kernel Data Isolation (hereafter named *KI* instead of full Kernel Isolation).

4.2. Formal expressions

We first formally define some design elements. We refer to the global partition tree *PartTree*. The memory blocks are owned by a partition *part* from the set *MappedBlocks[part]* stored in the *Blocks* structure. The subset of memory blocks that are user accessible is called *AccessibleMappedBlocks[part]*. The blocks holding metadata structures (configuration blocks, not user accessible) are referred to as *ConfigBlocks[part]*. The children are referred to as *children[part]*. *AllPaddr[blocks]* lists all addresses contained in the *blocks* list.

The proofs rely on 29 consistency properties. While they are necessary to conduct the formal verification of the security properties, their full description is not relevant to this presentation and is thus excluded. Nonetheless, they are described when required in the following proofs, while the full set can be found online. The security properties are formally described in the following.

Definition 4.1. (Vertical Sharing *VS*).

$\forall parent \in PartTree,$
$\forall child \in children[parent],$
$\forall blockAddrChild \in AllPaddr[Blocks[child]],$
$blockAddrChild \in AllPaddr[Blocks[parent]].$

Definition 4.2. (Horizontal Isolation *HI*)

$\forall parent \in PartTree,$
$\forall child1, child2 \in children[parent],$
$\forall blockAddr \in AllPaddr[MappedBlocks[child1]],$
$blockAddr \notin AllPaddr[MappedBlocks[child2]].$

Definition 4.3. (Kernel Data Isolation *KI*)

$\forall partition1, partition2 \in PartTree,$
$\forall blockAddr \in AllPaddr[AccessibleMappedBlocks[partition1]],$
$blockAddr \notin AllPaddr[ConfigBlocks[partition2]].$

Note that *KI* is not defined as a parent-child relationship because the isolation of metadata structures from accessible memory blocks must also be satisfied within the partitions that hold them. To be noted as well, proofs are conducted considering the full set of blocks and not only the active blocks configured in the MPU, because all blocks are eligible to be mapped in the MPU.

5. MPU-BASED HARDWARE MODEL

Pip-MPU's security properties must be verified on a hardware platform composed of a CPU (Central Processing Unit), MPU, memory, peripherals, and any other hardware components. However, the properties only depend on the kernel's effects, *i.e.* how Pip-MPU interacts with the hardware platform. This means the abstracted view does not need to include all the hardware platform's components because Pip-MPU just manipulates kernel structures and the MPU. Thus, the **abstracted hardware model** is just composed of a **memory model** and an **MPU model**, as shown in Figure 2. In particular, it does not include registers nor the current privilege mode as in other works [46] because we only consider Pip-MPU services that always execute in privileged mode. We propose in the following a formalisation of the MPU-based hardware model, as expressed in the Coq model.

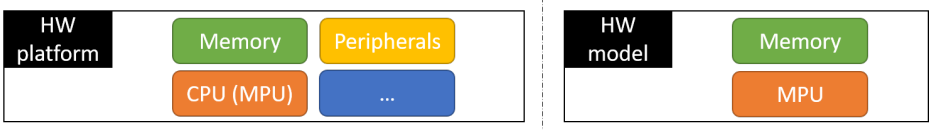

Figure 2: On the left, the real hardware composed of many elements. On the right, the abstract hardware model used in the proofs composed of the memory model and the MPU model.

Formal model of the memory. The memory is modelled as an association list (list of key-value pairs). It associates a physical address with a value, just like Pip (MMU)'s hardware model. The value is either a kernel structure entry or another physical address (representing a pointer). Indeed, the memory model is only filled with kernel-related elements because the services execute in privileged mode, reserved for the kernel, and as such never include userland operations. Furthermore, Pip's security properties only concern kernel structures and never the contents of the memory blocks of the userland partitions. Modelling these contents would then be useless to prove the properties.

Formal model of the Memory Protection Unit. The MPU configuration is modelled as a list of block identifiers (the active memory blocks of the current partition). Our Coq model includes a coercion on the length of this list that should be lower than the actual number of MPU regions.

```
MPU : list paddr ;
HMPU : length(MPU) <= MPURegionsNb
```

Each element of this list is mapped as an active MPU region and refers to a `BlockEntry` record containing the corresponding block information.

```
Record block := {
startAddr : paddr;
endAddr : paddr ;
Haddr : startAddr < endAddr ;
Hsize : endAddr - startAddr <
maxIdx(* the size of a memory
block cannot exceed the memory *)
}
```

```
Record BlockEntry : Type:=
{
read : bool;
write : bool ;
exec : bool;
present : bool;
accessible : bool;
blockindex : index;
blockrange : block ;
Hidx : blockindex <
kernelStructureEntriesNb (* limited
number of entries per kernel structure *)
}.
```

Note that this is a very simplified view of an MPU. It does not reflect the specific MPU version hardware constraints (size and alignments in ARMv7 for instance). The rationale is that Pip-MPU should be portable on different MPU versions and architectures that might have different constraints. The constraints are considered in the C implementation of the HAL.

Formal model of the system state. The system state captures the evolution of the system at a certain instant. Our system state is the same as Pip's. It considers a **system state** composed of the **memory state** and the **current partition state**. The current partition state is the identifier of a partition. This is the partition that calls the service, *i.e.* the parent partition for services operating on a child. The memory model has been introduced above. The state is updated each time a write operation modifies the memory model or the current partition identifier.

In Coq, we model the system as follows:

```
Record state : Type := {
currentPartition : paddr;
memory : list (paddr * value)
}.
```

Note that the MPU model is not present in the system state. Indeed, the MPU configuration depends on the partition and is stored in the corresponding kernel structure. As such, the MPU model is stored in the memory model and the current partition identifier is enough to find the kernel structure where to retrieve the MPU state.

5. SKETCH OF PROOF OF THE ISOLATION PROPERTY

Formal proofs of Pip-MPU's security properties, including isolation, are conducted in the Coq Proof Assistant [7] on the concrete model following Pip's proof workflow. The objective is to construct valid proof terms in the Gallina language and tactics that satisfy the proof goal, *i.e.* the isolation invariant. In this paper, we expose the logical arguments to verify the kernel services, while the full and verbose Coq proofs that mirror the detailed intuition are accessible online [8]. *The proof intuition and sketches of proof detailed below are then demonstrated by interactive, machine-checked proofs in Coq*.

From a functional point of view, Pip-MPU's services are distributed in two categories: pure informational services and modification services. We give the sketch of proof for the Pip-MPU services `findBlock` from the first category and `addMemoryBlock` from the second category. Because the proof target is Pip's security properties, we assume the consistency properties hold. Obviously, the consistency properties must be proven in the full proof.

5.1. Proof of pure informational services: example with `findBlock`

The `findBlock` service looks for a block in the memory space of a partition and returns all block attributes. It takes as parameters a partition descriptor and any address contained in the searched block. `findBlock` has a first phase of checks to verify the user parameters are in their expected range and correspond to meaningful values: 1) the partition descriptor where to search for the block exists and is either the current partition or one of its children, and 2) the searched address is contained in one of the mapped blocks. Then, it reads information in memory and returns them to the user: it returns the block's attributes.

Problem 1. Proof goal: the isolation invariant on `findBlock`.
$\{VS \ \& \ HI \ \& \ KI \ \& \ C\}$
`findBlock` *part addressInBlock*
$\{VS \ \& \ HI \ \& \ KI \ \& \ C\}$

Assumption. *The consistency properties C hold*.

Proof. The service does not modify the system state; thus, the system state is the same as before the execution of the service. We know the invariants were satisfied at the start, so they are still satisfied at the end of the service. □

We can observe that the sketch of proof is simple because of the unmodified system state.

5.1. Proof of modification services: example with `addMemoryBlock`

The `addMemoryBlock` service is interesting because it deals with a parent-child relationship, manipulates almost all memory types and modifies a single partition, the child. `addMemoryBlock` adds (shares) a memory block to a child partition. It also has a check phase of the user parameters before modifying the state. The order of modification instructions has no importance because we prove the security properties in the last step, when all modifications have passed. *s0* refers to the initial state at the service start, while s' refers to a final state, whatever the path of execution. We write *list* ++ *[element*] to represent a list *list* extended by the element *element*.

Problem 2. Proof goal: the isolation invariant on `addMemoryBlock`.
$\{VS \ \& \ HI \ \& \ KI \ \& \ C\}$
`addMemoryBlock` *child blockToShare*
$\{VS \ \& \ HI \ \& \ KI \ \& \ C\}$

Assumption. *The consistency properties C hold*.

Proof. The proof has two sorts of exits: either the service is fed with incorrect user parameters (values not in expected range, block not existing or not accessible and present…) and stops before modifying the state; or it passes through the whole modification phase. This shows the importance of not modifying the state before being sure the execution will go through the whole code to prevent exits that leave the state inconsistent with invariants that will fail.

Exit 1: incorrect user parameters. Incorrect user parameters precipitate the execution path in returning the NULL address before any updates have been done. Because of no state modifications, the same sketch of proof as for pure informational services can be applied. □

Exit 2: Correct user parameters. With correct user parameters, the service executes to the last line. To be noted, the *VS* and *HI* properties consider combinations of several partitions with parent-child relationships that could be the parent-child relationship in `addMemoryBlock`. The *NoDupPartitionTree* invariant guards against impossible cases implying cycles like the parent is also its own child. Furthermore, to assign the roles in the relationships, the *isChild* and *isParent* invariants support the proof and check whether the pair of studied partitions is a correct parent-child relationship or not.

Vertical Sharing. The service does not modify the parent's mapped blocks, so mappedBlocks[parent]$_{s'} = mappedBlocks[parent]_{s0}$. On the contrary, the child's mapped blocks do change: $mappedBlocks[child]_{s'} = mappedBlocks[child]_{s0} + +blockToShare$.

We need to show that:
$\forall blockAddr \in AllPaddr\left[mappedBlocks[child]\right]_{s'}, block \in AllPaddr\left[mappedBlocks[parent]\right]_{s0}$.

So given an address *addr*, there are two cases:

1) $addr \in AllPaddr\left[mappedBlocks[child]\right]_{s0}$ or 2) $addr \in AllPaddr[blockToShare]$.

Case 1: $addr \in AllPaddr\left[mappedBlocks[child]\right]_{s0}$. We know that the Vertical Sharing property is true at *s0*. □

Case 2: $addr \in AllPaddr[blockToShare]$. We know $blockToShare \in mappedBlocks[parent]_{s0}$ because we retrieve the block from that list. This means the *addr* already was in one of the parent's blocks when the property was true at *s0*. □

Horizontal Isolation. For this property, the focus is set on two children of a certain parent partition. The children's memory spaces must not intersect (neither their mapped blocks nor their kernel structures). Two cases must be covered for this property: 1) one of the children is the *child* modified by the service `addMemoryBlock` (whether it is the first or the second child does not matter as the property is symmetric) and 2) the children are not the considered *child*.

Case 1: The check phase states that the block is not shared at *s0*, which means we know that $blockToShare \notin mappedBlocks[child1]_{s0} \wedge blockToShare \notin mappedBlocks[child2]_{s0}$. The service just modifies one of the children as explained in Vertical Sharing, leaving the other one untouched.

We take here the example of *child1*. We get $mappedBlocks[child1]_{s'} = mappedBlocks[child1]_{s0} + +blockToShare \wedge mappedBlocks[child2]_{s'} = mappedBlocks[child2]_{s0}$. For *child1*, there are then two options: either we consider a block in the initial set of memory blocks $mappedBlocks[child1]_{s0}$ or it is *blockToShare*.

Assuming the property is true at initial state *s0* implies that the first option is trivial. . □

Thus, remains the second option. Memory spaces are disjoint only if $blockToShare \notin mappedBlocks[child2]_{s'}$, which is equivalent to $blockToShare \notin mappedBlocks[child2]_{s0}$ which follows from the check phase as stated above. □

Case 2 : for any partition not being the one modified in the service, their mapped blocks remain untouched in the modified state.

Hence, $mappedBlocks[child1]_{s'} = mappedBlocks[child1]_{s0} \wedge mappedBlocks[child2]_{s'} = mappedBlocks[child2]_{s0}$. The property is then equivalent to the property which was satisfied at *s0*. □

Kernel Data Isolation. The current partition is not modified, which means the focus is set on the child partition *child*.

Because the service calls only consider parent-child relationships, three cases must be covered here: 1) *partition1* and *partition2* are the same partition, 2) *partition1* is *partition2*'s parent, and 3) *partition2* is *partition1*'s parent.

All cases are split in two possibilities: a) *partition1* is the child in `addMemoryBlock` b) *partition2* is the child in `addMemoryBlock`. All in all, six cases must be considered. However, the configuration blocks are not modified for any partition, not even the child in `addMemoryBlock`, and so equal at *s0* and *s'*, therefore we need to prove:
$\forall partition1, partition2 \in PartTree,$
$\forall blockAddr \in AllPaddr\left[AccessibleMappedBlocks[partition1]\right]_{s'},$
$blockAddr \notin AllPaddr\left[ConfigBlocks[partition2]\right]_{s0}$.
This means the proof path is only crucial for changes in *partition1*, so possibility a).

Indeed, in any other combination with possibility b), *partition1* is different from the child in `addMemoryBlock` so its memory blocks are unchanged and so it leads to the initial state *s0* ($AllPaddr\left[AccessibleMappedBlocks[partition1]\right]_{s'} = AllPaddr\left[AccessibleMappedBlocks[partition1]\right]_{s0}$) when the property is assumed true.□

Case 1 a) : *partition1* and *partition2* are the same partition, so the child in `addMemoryBlock`.

An accessible memory block from the child is either one of the initial blocks or the newly copied block from the parent. Because the property is satisfied at the initial state, if it is one of the initial blocks, the proof is trivial. □

If the block is the newly added one, we know it is also part of the parent's memory space, especially at the initial state *s0*. Because there are no restrictions on relationships between *partition1* and *partition2* in the expression of the security property, the property is true at the initial state, especially for *partition1* being the parent partition and *partition2* being the child. The parent partition's accessible blocks did not change after service execution, so still the same set as at the initial state where the copied block belonged to this set. The property was satisfied at the initial state. □

Case 2 a) : with the child in `addMemoryBlock` being *partition1*, we know $AccessibleMappedBlocks[child]_{s'} = AccessibleMappedBlocks[child1]_{s0} + + blockToShare$ because the new added block *blockToShare* becomes accessible. Hence, two sub-cases: either we consider an address *addr* in the initial accessible blocks or in *blockToShare*. Furthermore, the service does not affect any partition except the current partition and the child in `addMemoryBlock`, so especially does not modify the eventual descendants of this child. Thus, the memory spaces of the children, notably for *partition2*, are unchanged compared to the initial state *s0*, and especially their configuration blocks: $ConfigBlocks[partition2]_{s'} = ConfigBlocks[partition2]_{s0}$ The first sub-case is then trivially verified because the security property is assumed true at *s0* for the same combination of *partition1* and *partition2*. □

For the second sub-case, we need to consider another combination for the security property at *s0*, taking the parent (current) partition of `addMemoryBlock` and *partition2*, *i.e.:*

$addr \in AllPaddr\left[AccessibleMappedBlocks[current]\right]_{s0}$,
$addr \notin AllPaddr\left[ConfigBlocks[partition2]\right]_{s0}$.

Indeed, $blockToShare \in AccessibleMappedBlocks[current]_{s0}$ because of the Vertical Sharing property and the fact that it is this block that is copied into the child *partition1* during `addMemoryBlock` so it must be accessible as verified in the check phase of the service. This means *addr* also lies in one of the initial accessible blocks of the current partition when the property is assumed true. □

Case 3 a) : *partition2* is the parent (current) partition of the child in `addMemoryBlock`.

Similar to discussed previously, the configuration blocks of the child and the accessible mapped blocks of the parent are untouched after executing the service, so $ConfigBlocks[partition2]_{s'} = ConfigBlocks[partition2]_{s0} \wedge AllPaddr\left[AccessibleMappedBlocks[current]\right]_{s'} = AllPaddr\left[AccessibleMappedBlocks[current]\right]_{s0}$. This is equivalent to proving the property at *s0* which is assumed to be true. □

7. EVALUATION OF PIP-MPU'S PROOF DEVELOPMENT

We evaluate the proof development with common metrics linked to formal verification.

7.1. Proof setup

The proofs are developed using the CoqIDE 8.13.1 on an ArchLinux 5.18.1 distribution. The tools run on an HP ZenBook G4 17 provided with a 64-bit Intel i7-7820HQ 4-core CPU, 16 GB RAM, 16 GB swap space, 155 GB SSD usable space. The security properties are proven apart from the consistency properties for the modification service.

7.2. Results

Three services have been completely proven : `readMPU`, `findBlock` and `addMemoryBlock`.

Table 3 provides proof metrics on the proof development process (formalisation and the verification stages) and summarizes the current status of the proof development.

The metrics on the invariants provide information on the number of consistency properties and security properties. The invariants are also written (specified) in Gallina and the number of SLOC reports the total size of the properties, trimmed from comments. The formalisation stage considers three phases: the HW specification understanding, the design phase, and the Coq model implementation. The first phase consisted in collecting, reading, and combining various sources on the target HW platform [23, 31, 47, 48, 49]. The design phase includes any conversations, thinking time, methodological reflections, paper trials that finally converged to services in pseudo-code which were eventually translated in Python scripts for simulation purposes. The last phase required a full model implementation by forking the Pip (MMU) project [40] and by adapting the model to a hardware without virtual memory and modifying the kernel structures to match Pip-MPU's design, given the Python implementation. Duration is estimated based on status advancement reports and code versioning history.

We do include in our time estimation the time spent to prove the proof levels of the low-level HAL primitives because they are proved along the way, when used by a service. However, because the primitives are very similar, the individual proofs are basically small adaptations of the proofs on other primitives, so the duration is negligible compared to the development of the services. The proof development metrics gather the number of Gallina lines of the services and corresponding Coq proof lines to prove them (without internal functions and lemmas). The number of lines is computed after removing all comments. However, this method does not really

capture the number of necessary steps in the proof development process because a great number of variables are split over several lines and because there could be several tactics on the same line. So we also estimate the proof effort regarding the number of used *tactics* which are terms of the language used by the prover and the way the proof developer interacts with it. They are instructions capturing the logical reasoning and intuition of the proof developer in understandable language to the prover. The number of tactics is computed by removing all comments and proof specifications and counting the number of instructions ending with a dot or semi-colon.

Table 2: Proof status and effort.

Proof status				
Services				
# services	15			
# fully verified services	3			
Invariants				
# consistency properties	29			
# security properties	3			
Total invariant properties	32			
Consistency properties SLOC	129			
Security properties SLOC	14			
Total properties SLOC	143			
Proof development				
Formalisation				
Duration of HW specification understanding	2 months			
Duration of design phase	9 months (including 2 months Python simulation)			
Duration of model implementation in Coq	2 months			
Duration of service development in Coq	2 months			
	readMPU	**findBlock**	**addMemoryBlock**	
# SLOC (w/o HAL)	11	14	25	
Proofs	*All properties*		*Consistency properties*	*Security properties*
# LOP (Lines of Proof)	76	81	8572	1607
# tactics	90	93	8883	842
Ratio #LOP/SLOC	6.9	5.8	342.8	64.3
Ratio #tactics/SLOC	8.18	6.6	355.3	33.7
Duration of proof development	2 days	4 hours	9 months	4 days
person-month (estimate)	0.09	0.02	6	0.18
Ratio person-month/LOC	0.008	0.0014	0.22	0.0072
Duration of proof compilation	0.5s	0.4s	939s (15.65 min)	127s (2.1 min)
Memory footprint during proof compilation	400 MB	400 MB	5000 MB	900 MB
CPU usage during proof compilation	100%	110%	115 %	105%
Proof coverage (estimate)	100%	100%	100%	100%

We report the human time spent during the verification of each service estimated via the code versioning history. Note that `findBlock` has about the same number of SLOC (Source Lines of Code) and LOP (Lines of Proof) as `readMPU`, but is proved in some hours instead of days. Indeed, they share a lot of common functions and lemmas, and so the *findBlock* proof recycled a lot from the previous proof, drastically diminishing the proof effort.

The compilation of all proofs on the host machine in charge of the proof development is expressed in time, CPU usage and memory footprint. The measurements are recorded by using the `psrecord` tool (https://github.com/astrofrog/psrecord), as illustrated with `addMemoryBlock` in Figure 3. They do not include the compilation time to prove their dependency lemmas (that could be common between different services), just the main lemma.

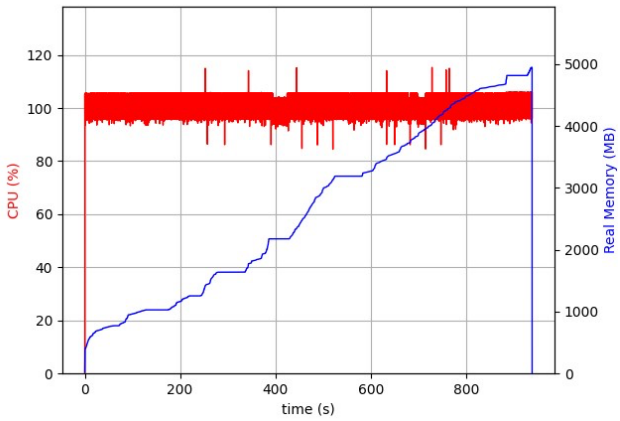

Figure 3: CPU and memory usage during the proof compilation of `addMemoryBlock`

The proof coverage estimation gives the completion rates of each service.

7.3. Bug discovery during the verification

The verification process unveiled a bug in the `addMemoryBlock` service. Indeed, the check phase omitted to verify if the block to be shared with the child partition was not already shared with another partition. This would have had the undesired effect that a block would be present in several children, which would break the isolation property. As a matter of fact, the bug was discovered during the proofs of the security properties, where it was impossible to conclude. Indeed, it was impossible to demonstrate the Horizontal Isolation property with evidence from the proof context that no other partition, except its ancestors, already contained the block to be shared with the child partition. A quick analysis of the code showed this faulty state was possible to reach. A bug affecting the security properties would never pass because of this impossibility to conclude and is not due because of a lack of skills of the proof developer or functionality of the theorem prover. Except for the case where information is missing from the proof context, a latter where hypotheses would be inconsistent with oneanother would not let us finalise the proofs as well. The lack of hypotheses or inconsistent ones stopping the demonstration of a proof goal is a stringent point for the soundness of the approach and is assured by the theorem prover. Furthermore, the instant in the proofs where the demonstration stops clearly points out the inconsistency, and gives a sense of how to correct the code to not fall into this unexpected situation. This demonstrates the very essence of formal verification to detect such situations. It also exhibits that severe bugs can be found even in a small code base which focuses on security. Bug discovery is documented in other formal verification projects [25, 39, 46]. This bug also stresses what is and what is not claimed in this formal verification process. The only properties that are proved are the isolation invariant and no proofs concern functional properties. This remark also refers to other security policies than Pip's, for example, the isolation invariant is not concerned about the read, write, and execution rights of the memory blocks.

8. DISCUSSION AND PERSPECTIVES

Our results demonstrate higher assurance in Pip-MPU's isolation property by formal verification. We discuss in this section related assumptions and consequences.

8.1. Proof status

While only some kernel services of the full set have been fully formally verified, we are confident in the possibility to finish the proof because we addressed and completed proofs from the two proof categories (*i.e.* with and without state modifications). They lay the foundations for the other services. A future challenge is to show the proof framework is also correctly working for the type of service that destructs kernel elements, such as `removeMemoryBlock`, instead of services that create kernel elements like `addMemoryBlock`. Nevertheless, the approach already showed an important spin-off with the bug discovery.

8.2. Proof development

Many parts of the proof script were eluded in this paper for the sake of clarity by focusing on the crucial moments of the proof. Additional constraints on kernel structures, for example bounded values for a certain type, are not discussed here but are present in the full proof script. In addition to that, proof development is still in its infancy with research efforts trying to rationalise the process from scratch to proof maintenance [51, 52, 54]. Future work aims to find new proof techniques to decrease the overall proof effort, equally desired by proof experts than newcomers.

8.3. Proof metrics

Presented proof metrics give an idea of the underlying proof effort, however, must not be taken as raw metrics without further discussion.

On the Coq model implementation, the consistency properties are introduced as the proofs of the services evolve, *i.e.* proof scripts are developed when needed and not by anticipation. Meanwhile, it means the formalisation is still ongoing and the duration reflects the time spent to cover the three services fully addressed until now, but could increase in the future due to new properties.

While the SLOC shows a quantitative measure of the proof effort, it does not indicate a transferable metric to other systems or other developers that would develop the proofs differently according to their style (for example in terms of structure, proof context reorganisation, or use of lemmas to generalise situations). Duplicates are clearly indicated in the proof script for future maintenance. The goal of this work was to prove the security properties, however, the means to achieve it are infinite and not optimised. In other words, many parts are copies of other proofs, wordy, and might only exist for human clarity while being useless for the prover, which does not reflect neither the time spent on the proof nor the difficulty of the task. This is also emphasised in other formal verification projects [46].

In addition to that, the person-month/SLOC ratio gives us an idea of the average effort to prove a single line of code. In Pip-MPU, we can take the overall person-month estimate (6.29) and the current number of verified instructions (50) to compute the estimate of 0.13 person-month by line of code. We can compare the time spent just to prove the security properties, as it is done separately in seL4 [55] on the abstract model and directly on the concrete model in Pip-MPU. For seL4, this leads to a ratio of 0.0066 person-month per line of code, whereas in Pip-MPU it is at worst 0.0072 for `addMemoryBlock`, so remarkably close, and because there is no proof effort in `readMPU` and `findBlock` it leads to an average of 0.0036 for the time being, so twice better.

8.4. Proof assumptions

Incorrect assumptions imply a false context, which means everything can be proven out of it. Our assumptions are mostly similar to Pip (MMU) [39] and other formally verified kernels [24, 28, 30] in a broader perspective.

Indeed, the trustworthiness of the proofs depends on:

- the used tools: the Coq kernel, gcc, which could be replaced by CompCert (a formally proven C compiler), and Digger (a custom tool that provides a literal translation from Gallina to C) which could be replaced by its verified equivalent δx to increase confidence;
- the correctness of the model: we considered a simplified hardware model (memory layout, MPU) which captures the basic functionalities of the hardware platform because their formal specification does not exist yet. One way to increase confidence would be to play the proofs on a formal model of the hardware. It could be directly handed over by processor manufacturers, like recent efforts on the ARMv8-A architecture [55] (not available for Cortex-M devices), on which we would directly "plug" our proofs. Our hardware state assumes the kernel manipulates a writable non-volatile memory and we assume no external influence can affect memory without bypassing the MPU control. In particular, we consider either peripherals like the DMA (Direct Memory Access) not existing or disabled, like other verified kernels [46, 56]. Recent research showed the DMA could be used even in strict secure environments [13, 44]. Pip (MMU) already does this by controlling that the configuration registers of the DMA are valid to the partition accessing them, and is currently being adapted to Pip-MPU by our development team. Another concern is that the low-level primitives of the HAL might not reflect the real C implementation that is manually written and so not checked. In other words, wrongly implemented C primitives could do something else than what was expected in the Coq model. The probability of such an event is considered low because the primitives are composed of a small amount of C lines (around 5 lines on average) and directly correspond, literally, to their Coq model counterparts;
- the hardware implementation and deltas with its specification: we must trust a correct implementation of the hardware platform, and that the implementation itself is consistent. We assume that the MPU is working correctly (physical failures or attacks) and that the memory controller correctly writes and reads the values that are passed by the HAL primitives. ;
- the specification of the isolation invariant: the security properties and all consistency properties, could be incorrect. However, many similarities exist between Pip (MMU) and Pip-MPU which seeks the same proof goal. Pip (MMU) proved an equivalence with the isolation formalised by Rushby [1, 57] and so we are confident about our specification. ;
- the bootstrapping phase : we assume a secure state at each service execution. The code base is limited (< 10 KB [37]) and we take extra care to bootstrap the system to correctly prepare and launch the root partition in a state which satisfies the isolation invariant.

9. CONCLUSION

This paper considers the minimalist separation kernel Pip-MPU and presents the formal verification of its isolation property. Pip-MPU brings high isolation flexibility for constrained devices and is derived from the formally verified Pip protokernel. Pip-MPU replicates Pip's spatial memory partitioning model and so its security properties, including isolation, must also be satisfied. Even though functionally equivalent, Pip-MPU's code is completely refactored because of the MMU to MPU transposition causing major structural differences. Pip's formal proofs are thus not applicable any more for Pip-MPU. In this paper, we retain Pip's formal verification approach on the concrete model and propose equivalent security properties by introducing an MPU model and similar low-level invariants. We demonstrate the formal verification approach

on two read-only services and one service that modifies the state with writing instructions. The publicly released proofs have been implemented and checked using the Coq Proof Assistant, thereby validating a formal approach directly on a concrete model and helping the discovery of a critical bug related to isolation.

ACKNOWLEDGEMENTS

The research leading to these results were funded by ANRT Convention Cifre n°2020/0380 and contributed to the TinyPART project funded by the MESRI-BMBF German-French cybersecurity program under grant agreements n°ANR-20-CYAL-0005 and 16KIS1395K. This paper reflects only the authors' views. ANRT, MESRI and BMBF are not responsible for any use that may be made of the information it contains. The authors would like to thank Damien Amara, member of the 2XS team (CRIStAL laboratory, University of Lille), and Samuel Legouix (SASE team, Orange Innovation Caen), for their contributions to port RIOT OS on Pip-MPU.

REFERENCES

[1] J. M. Rushby (1981). Design and verification of secure systems. *Proceedings of the 8th ACMSymposium on Operating Systems Principles, SOSP 1981* 15 (1981), 12–21. https://doi.org/10.1145/800216.806586

[2] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud (2021). Nested compartmentalisation for constrained devices. In *2021 8th International Conference on Future Internet of Things and Cloud (FiCloud)*. IEEE Press, 334–341. https://doi.org/10.1109/FiCloud49777.2021.00055

[3] Nicolas Dejon, Chrystel Gaber, and Gilles Grimaud (2022). From MMU to MPU: adaptation of the Pip kernel to constrained devices. In *3rd International Conference on Internet of Things & Embedded Systems (IoTE 2022)*. AIRCC Publishing Corporation, Sydney, Australia. https://doi.org/10.5121/csit. 2022.122309

[4] European Telecommunications Standards Institute (ETSI) (2022). ETSI EN 303 645: Cyber Security for Consumer Internet of Things: Baseline Requirements. https://www.etsi.org/deliver/etsi_en/303600_303699/303645/02.01.01_60/en_303645v020101p.pdf. [Online; accessed October 10, 2022].

[5] European Union (2022). EU Cyber Resilience Act. https://digitalstrategy. ec.europa.eu/en/policies/cyber-resilience-act. [Online; accessed October 10, 2022].

[6] National Institute of Standards and Technology (NIST) (2022). IoT Cybersecurity Improvement Act of 2020. https://www.congress.gov/bill/116thcongress/house-bill/1668. [Online; accessed October 10, 2022].

[7] INRIA (2023). *Website of: The Coq Proof Assistant*. Retrieved February 20, 2023 from https://coq.inria.fr

[8] Pip Development Team (2022). Website of: Proofs of Pip (MPU) Gitlab. https://github.com/2xs/pipcore-mpu/tree/master/proof/invariants. [Online; accessed October 10, 2022].

[9] Linaro (2020). Website of: TrustedFirmware-M. https://www.trustedfirmware.org/projects/tf-m/. [Online; accessed November 20, 2020].

[10] Emmanuel Baccelli, Oliver Hahm, Mesut Günes, Matthias Wählisch, and Thomas C. Schmidt (2013). RIOT OS: Towards an OS for the Internet of Things. In *2013 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE Press, Turin, Italy, 79–80. https://doi.org/10.1109/INFCOMW.2013.6970748

[11] Zephyr Project (2020). Website of: The Zephyr Project. https://www.zephyrproject.org/. [Online; accessed November 20, 2020].

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

[13] Ryad Benadjila, Arnauld Michelizza, Mathieu Renard, Philippe Thierry, and Philippe Trebuchet (2019). WooKey: Designing a Trusted and Efficient USB Device. In *Proceedings of the 35th Annual Computer Security Applications Conference* (San Juan, Puerto Rico, USA) *(ACSAC '19)*. Association for Computing Machinery, New York, NY, USA, 673–686. https://doi.org/10.1145/3359789.3359802

[14] Chung Hwan Kim, Taegyu Kim, Hongjun Choi, Zhongshu Gu, Byoungyoung Lee, Xiangyu Zhang, and Dongyan Xu (2018). Securing Real-Time Microcontroller Systems through Customized Memory View Switching. *NDSS* (2018). https://doi.org/10.14722/ndss.2018.23107

[15] Abraham A Clements, Naif Saleh Almakhdhub, Saurabh Bagchi, and Mathias Payer (2018). ACES: Automatic Compartments for Embedded Systems. In *27th USENIX Security Symposium (USENIX Security 18)*. USENIX Association, Baltimore, MD, 65–82. https://www.usenix.org/conference/usenixsecurity18/presentation/clements

[16] Xia Zhou, Jiaqi Li, Wenlong Zhang, Yajin Zhou, Wenbo Shen, and Kui Ren (2022). OPEC: Operation-Based Security Isolation for Bare-Metal Embedded Systems. In *Proceedings of the Seventeenth European Conference on Computer Systems* (Rennes, France) *(EuroSys '22)*. Association for Computing Machinery, New York, NY, USA, 317–333. https://doi.org/10.1145/3492321.3519573

[17] Sandro Pinto and Cesare Garlati (2020). Multi Zone Security for Arm Cortex-M Devices. Nuremberg.

[18] Patrick Koeberl, Steffen Schulz, Ahmad-Reza Sadeghi, and Vijay Varadharajan (2014). TrustLite: A Security Architecture for Tiny Embedded Devices. In *Proceedings of the Ninth European Conference on Computer Systems* (Amsterdam, The Netherlands) *(EuroSys '14)*. Association for Computing Machinery, New York, NY, USA, Article 10, 14 pages. https://doi.org/10.1145/2592798.2592824

[19] Hongyan Xia, Jonathan Woodruff, Hadrien Barral, Lawrence Esswood, Alexandre Joannou, Robert Kovacsics, David Chisnall, Michael Roe, Brooks Davis, Edward Napierala, John Baldwin, Khilan Gudka, Peter G. Neumann, Alexander Richardson, Simon W. Moore, and Robert N. M. Watson (2018). CheriRTOS: A Capability Model for Embedded Devices. In *2018 IEEE 36th International Conference on Computer Design (ICCD)*. IEEE Press, 92–99. https://doi.org/10.1109/ICCD.2018.00023

[20] Arm mbed (2020). Website of: Mbed OS. https://os.mbed.com/mbed-os/. [Online; accessed November 20, 2020].

[21] Tock development team (2020). Website of: Tock. https://www.tockos.org/. [Online; accesse November 20, 2020].

[22] Guillaume Bouffard and Léo Gaspard (2018). Hardening a Java Card Virtual Machine Implementation with the MPU. In *Symposium sur la sécurité des technologies de l'information et des communications (SSTIC)* (Rennes) https://www.sstic.org/2018/presentation/hardening_a_java_card_virtualmachine_implementation_with_the_mpu/

[23] ARM (2022). Website of: ARMv7-M Architecture Reference Manual. https://developer.arm.com/documentation/ddi0403/ee/. [Online; accessed October 10, 2022].

[24] Gernot Heiser (2005). Secure embedded systems need microkernels. *USENIX* 30, 6 (2005), 9– 13.

[25] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood (2009). SeL4: Formal Verification of an OS Kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles* (Big Sky, Montana, USA) *(SOSP '09)*. Association for Computing Machinery, New York, NY, USA, 207–220. https://doi.org/10.1145/1629575.1629596

[26] University of Cambridge and Technische Universität München (2022). Isabelle. https://isabelle.in.tum.de/index.html. [Online; accessed October 10, 2022].

[27] Google (2022). Website of: Project Sparrow: KataOS (Github). https://github.com/AmbiML/sparrow-kata. [Online; accessed October 31, 2022].

[28] David Costanzo, Zhong Shao, and Ronghui Gu (2016). End-to-End Verification of Information-Flow Security for C and Assembly Programs. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Santa Barbara, CA, USA) *(PLDI '16)*. Association for Computing Machinery, New York, NY, USA, 648–664. https://doi.org/10.1145/2908080.2908100

[29] Yves Bertot and Pierre Castéran (2013). *Interactive Theorem Proving and Program Development*. Springer Berlin, Heidelberg, Berlin.

[30] Stéphane Lescuyer (2015). ProvenCore: Towards a Verified Isolation Micro-Kernel. In *MILS@HiPEAC*.

[31] Arm Limited (2017). Introduction to the Armv8-M architecture. Version 2.0.

[32] Narjes Jomaa, David Nowak, and Paolo Torrini (2018). Formal Development of the Pip Protokernel. In *ENTROPY 2018 Workshop* (Villeneuve d'Ascq, France). ACM, New York, NY, USA.

[33] D Bolignano (2016). Formally Proven and Certified Off-The-Shelf Software Components: the Critical Links for Securing the Internet of Things. *CESAR* (2016). https://www.cesarconference. org/wp-content/uploads/2017/06/Actes_Cesar_2016.pdfpage=109

[34] Microsoft (2022). Website of: An overview of Windows 10 IoT. https://learn.microsoft.com/enus/ windows/iot-core/windows-iot. [Online; accessed October 10, 2022].

[35] eLinux (2022). Website of: Embedded Linux Distributions (wiki). https://elinux.org/Embedded_Linux_Distributions. [Online; accessed October 10, 2022].

[36] Google (2022). Website of: Fuchsia. https://fuchsia.dev. [Online; accessed October 10, 2022].

[37] Linaro (2022). Website of: TrustedFirmware-A. https://www.trustedfirmware.org/projects/tf-a/. [Online; accessed October 10, 2022].

[38] Quentin Bergougnoux (2019). *Co-design et implémentation d'un noyau minimal orienté par sa preuve, et évolution vers les architectures multi-coeur*. Ph. D. Dissertation. Université de Lille, Villeneuve d'Ascq, France. Advisor(s) Grimaud, Gilles and Cartigny, Julien.

[39] Narjes Jomaa (2018). *Le co-design d'un noyau de système d'exploitation et de sa prevue formelle d'isolation*. Ph. D. Dissertation. Université de Lille, Villeneuve d'Ascq, France. Advisor(s) Grimaud, Gilles and Nowak, David.

[40] Pip Development Team (2022). Website of: Pip (MMU) Gitlab. https://gitlab.univlille. fr/2xs/pip/pipcore/-/tree/main/. [Online; accessed October 10, 2022].

[41] ARM (2022). Website of: Arm Cortex-M Processor Comparison Table. https://developer.arm.com/-/media/Arm%20Developer%20Community/PDF/ Cortex-A%20R%20M%20datasheets/Arm%20Cortex-M%20Comparison%20Table_v3.pdf. [Online; accessed October 10, 2022].

[42] ARM (2017). 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/. [Online; accessed March 09, 2021].

[43] Narjes Jomaa, Paolo Torrini, David Nowak, Gilles Grimaud, and Samuel Hym (2018). Proof-Oriented Design of a Separation Kernel with Minimal Trusted Computing Base. *Automated Verification of Critical Systems 2018 (AVoCS 2018)* 76 (2018).

[44] Ronghui Gu, Zhong Shao, Hao Chen, Xiongnan Wu, Jieung Kim, Vilhelm Sjöberg, and David Costanzo (2016). CertiKOS: An Extensible Architecture for Building Certified Concurrent OS Kernels. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation* (Savannah, GA, USA) *(OSDI'16)*. USENIX Association, USA, 653–669.

[45] C. A. R. Hoare (1969). An Axiomatic Basis for Computer Programming. *Commun. ACM* 12, 10 (oct 1969), 576–580. https://doi.org/10.1145/363235. 363259

[46] Pauline Bolignano (2017). *Formal Models and Verification of Memory Management in a Hypervisor*. Ph. D. Dissertation. Universite de Rennes 1, Rennes, France.

[47] Nordic Semiconductor (2022). Website of: nRF52832 Product Specification v1.8. https://infocenter.nordicsemi.com/pdf/nRF52832_PS_v1.8.pdf. [Online; accessed October 10, 2022].

[48] Nordic Semiconductor (2022). Website of: nRF52840 DK (Nordic Semiconductor). https://www.nordicsemi.com/Products/Development-hardware/ nrf52840-dk/. [Online; accessed March 18, 2022].

[49] TockOS (2022). Website of: Tracking: Support RISC-V (Github). https://github.com/tock/tock/issues/1135. [Online; accessed October 10, 2022].

[50] June Andronick, Ross Jeffery, Gerwin Klein, Rafal Kolanski, Mark Staples, He Zhang, and Liming Zhu (2012). Large-scale formal verification in practice: A process perspective. In *2012 34th International Conference on Software Engineering (ICSE)*, Vol. 2012 34th International Conference on Software Engineering (ICSE). IEEE Press, Zurich, Switzerland, 1002–1011. https://doi.org/10.1109/ICSE.2012.6227120

[51] David Aspinall and Cezary Kaliszyk (2016). Towards Formal Proof Metrics. In *Proceedings o the 19th International Conference on Fundamental Approaches to Software Engineering – Volume 9633*. Springer-Verlag, Berlin, Heidelberg, 325–341. https://doi.org/10.1007/978-3-662-49665-7_19

[52] Jonas Haglund and Roberto Guanciale (2019). Trustworthy Isolation of DMA Enabled Devices. In *Information Systems Security: 15th International Conference, ICISS 2019, Hyderabad, India, December 16–20, 2019, Proceedings* (Hyderabad, India). Springer-Verlag, Berlin, Heidelberg, 35–55. https: //doi.org/10.1007/978-3-030-36945-3_3

[53] Talia Ringer (2021). *Proof Repair*. Ph. D. Dissertation. University of Washington, Seattle, Washington.

[54] Alastair Reid (2017). Who Guards the Guards? Formal Validation of the Arm v8-m Architecture Specification. *Proc. ACM Program. Lang.* 1, OOPSLA, Article 88 (oct 2017), 24 pages. https://doi.org/10.1145/3133912

[55] Gerwin Klein, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser (2014). Comprehensive Formal Verification of an OS Microkernel. ACM Trans. Comput. Syst. 32, 1, Article 2 (feb 2014), 70 pages. https://doi.org/10.1145/2560537

[56] seL4 (2022). Website of: FAQ seL4 including DMA. https://docs.sel4.systems/projects/sel4/frequently-asked-questions.html. [Online; accessed October 10, 2022].

[57] John M. Rushby (1982). Proof of Separability: A Verification Technique for a Class of a Security Kernels. In *Proceedings of the 5th Colloquium on International Symposium on Programming*. Springer-Verlag, Berlin, Heidelberg, 352–367.

APPENDIX

A. PIP/PIP-MPU Executable and Proof Workflow

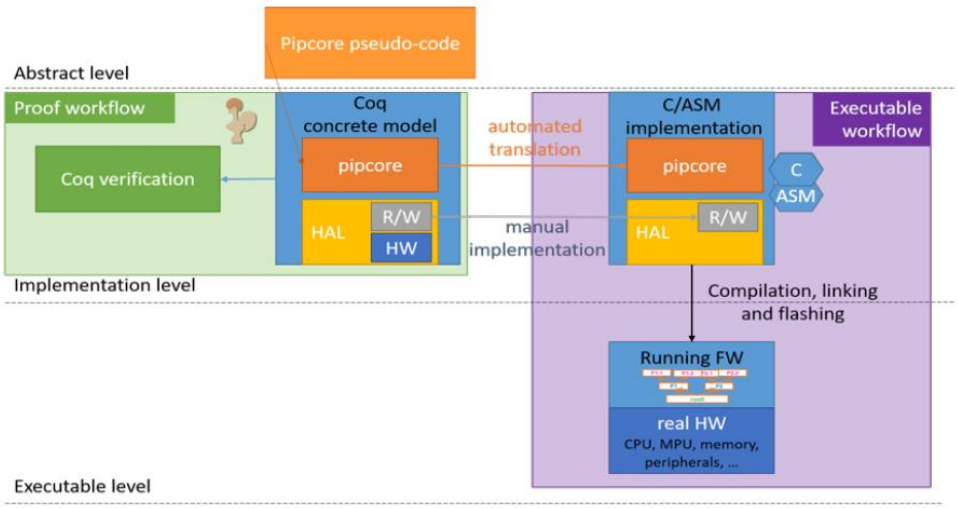

Figure 4: Pip/Pip-MPU workflow: proofs and executable.