Proving Memory Separation in a Microkernel by Code Level Verification

Christoph Baumann
Saarland University,
Saarbrücken, Germany
baumann@wjpserver.cs.uni-saarland.de

Thorsten Bormer
Karlsruhe Institute of Technology,
Karlsruhe, Germany
bormer@kit.edu

Holger Blasum and Sergey Tverdyshev
SYSGO AG, Klein-Winternheim, Germany
holger.blasum@sysgo.com
sergey.tverdyshev@sysgo.com

***Abstract*—Often, an integrated mixed-criticality system is built in an environment which provides separation functionality for available on-board resources. In this paper we treat such an environment: the PikeOS separation kernel – a commercial real-time embedded operating system. PikeOS allows applications with different safety and security levels to run on the same hardware. Obviously, a mixed-criticality system built on PikeOS relies on the correct implementation of the separation mechanisms. In the context of the Verisoft XT [1] and TECOM [2] projects we apply deductive formal software verification to the PikeOS separation mechanisms in order to validate this security requirement.**

In this work we consider formal verification of a kernel memory manager which is one of the crucial components of the separation functionality. The verification of the memory manager is carried out on the level of the source code using the VCC tool developed by Microsoft Research. Furthermore, we present the overall correctness arguments needed to prove the intended separation property, describe the necessary functional correctness properties of PikeOS, and explain how to formulate these properties in a modular way to be used by VCC.

In doing so we demonstrate how a proof of a non-functional system requirement can be conducted based on results from formal verification on the lowest possible level of human-written artefacts, that is the source code level.

***Keywords*-deductive verification; microkernel; memory separation**

I. Introduction

Modern electronic systems used in aircrafts, cars, or space vehicles combine applications with different security, safety, and real-time requirements. Systems with such mixed requirements are often referred to as mixed-criticality systems. Usually, integrated mixed-criticality systems are build on top of an environment with separation functionality of on-board resources. For example, a special purpose operating system can provide required separation. Such an architecture allows the system integrator to deploy applications with different level of trust e.g. applications from different vendors, certified and non-certified applications. Obviously, such a mixed-criticality system works correctly only if the underlying operating system does, too.

In this paper we treat such a real-time operating system: the PikeOS [3] separation kernel. PikeOS is an L4-based implementation of a separation kernel. It provides separated partitions where user applications (note that they can be another operating systems, too) run under supervision. PikeOS is used in control systems, health-care, and especially avionics. PikeOS is part of DO-178B [4] certifications for components of the Airbus A350 and A400M [5].

While the separation functionality of PikeOS consists of separation for CPU time, memory, external devices and interfaces, in this paper we treat the separation for memory. PikeOS supports virtualization of memory such that user applications run in their own virtual memory. In addition the kernel memory manager assigns to partitions pieces of memory which are organized in pages. After boot-time initialization, the partitions have been statically assigned memory pages. At run-time these pages dynamically store data structures used by the partitions.

In this treatise we trace a proof of the strict separation of kernel memory partitions at run-time. To this end we prove that the memory manager does not violate the predefined partitioning of memory, i.e. the run-time reallocations of pages between data structures within a partition do not cause any violation of the statical allotment of memory between partitions. Thus, we show that there is no unintended information flow between partitions, due to a misfunction of a partition's memory manager.

We formally verify the latter property on the code level with the help of the verification tool VCC [6]. We employ an "iterative push button" approach that is 1) annotate implementation (create specification; guide proof search), 2) push button (apply VCC), 3) if the verification failed, analyse why and repeat the loop. The desired separation property is easy to describe informally but infeasible to define directly in the specification language supported by code annotation tools like VCC or Frama-C. Therefore, we developed a verification methodology that breaks down the high-level, non-functional requirement into functional memory manager properties, which can be proven by our tool and subsequently used in a paper-and-pencil proof of the desired separation property.

Related Work: A wide body of research in operating system verification in general exists [7]. Although methodologically quite different, in particular the seL4 verification [8] is a very impressive undertaking. In comparison, from a purely quantitative point of view, the proof presented here is more modest in scope. However, unlike the translation approach in seL4 our proof is based on code annotations directly embedded into the code, and thus, it offers a very good traceability to the optimized production code and an easy integration into

existing development/certification processes.

There are different kinds of memory managers. The one we presented here suits well separation kernels. There are also memory managers for user-space allocations, e.g. to implement the C language construct `malloc`. Such a manager has been formally treated in context of virtual memory managers [9], [10] although not from a code annotation point of view.

Our work is comparable to the demonstration of a global memory manager correctness by Tuch [11] in the sense that a part of it is to show the functional correctness of a memory manager. In contrast to Tuch, we apply an automated verification methodology which replaces the interactive proof effort in Isabelle/HOL. As a result, the size of the proof is reduced from 108 pages to about 850 lines of code. Moreover Tuch's work is not concerned with the utilization of the memory manager in a virtualizing environment where several partitions are requesting chunks of memory and the separation of these resources is an issue.

The remainder of the paper is organized as follows. In Section II we describe our case study: a separation kernel and a memory manager. Section III presents the overall separation argument and a proof based on the artifacts verified on the code level. Section IV introduces the reader to the core concepts of code-level annotation and verification with VCC. We demonstrate the application of these principles to the memory manager in Section V. In Section VI we further discuss our approach and present "lessons learned". We conclude in Section VII.

II. Separation Kernels

The purpose of a separation kernel is to keep apart different applications with either no interference at all or at most a controlled interference. In this work we only examine the first case – security policies for controlled interference are not within the scope of this paper.

To ensure application memory disjointness, on most existing computer systems the hardware memory management unit (MMU) separates virtual memory assigned to processes (address spaces) by maintaining data structures for look-up (page tables) that map virtual addresses within different address spaces to physical addresses. For a running user process, this creates the illusion that the entire address range is available and allows to keep address spaces used by different applications separate.

Separation kernels add another layer: they can group several address spaces together. As the system can be configured so that the address spaces are kept cleanly apart, such sets of address spaces are called *partitions*. Conforming to common separation kernel usage we define a *thread* as an executing unit, and a *task* as an abstraction of the hardware that contains a virtual address space, threads, and (potentially) other allocated resources. In a kernel with strict separation policy a thread is only allowed to access resources belonging to tasks in his own partition. Separation kernels are also referred to as

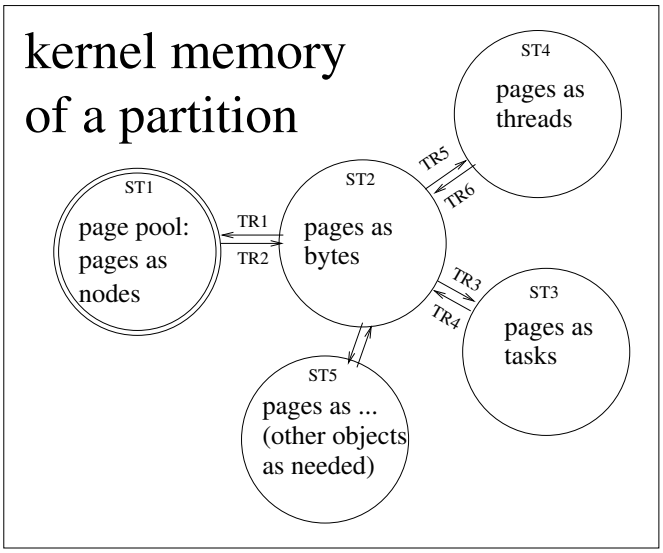

Fig. 1: Kernel objects in a partition of a separation kernel. The enclosing rectangle is a partition. Circles are sets of pages labeled STx for state x. Arrows labeled TRx indicate that one page at a time can be moved between the sets, a *page transition*. The starting point is the page pool marked by a double circle.

MILS (multiple independent levels of security) kernels [12], [13].

A. Kernel Memory Management in Separation Kernels

In order to mirror the strict per-partition separation of user resources also into kernel memory, it is a natural kernel design choice to collect and encapsulate the partition's kernel resources in dedicated partition objects. That is, each partition holds separate kernel data structures facilitating the execution of the partition's threads. These structures are allocated and deallocated dynamically by the kernel memory manager, which is administrating disjoint pools of memory on the heap.

In Figure 1 we show a snapshot of an abstracted kernel memory heap structure of a separation kernel within one partition. It describes the assignment of memory via a memory manager, between an initial "page pool" (ST1, that keeps pages as nodes of a linked list data structure) and runtime production kernel data structures, e.g. thread (ST4), task (ST3) or other objects (ST5). The form of "other objects" (ST5) depends on other implementation-provided features, e.g. interrupt handling, user space memory management structures.

The correctness of the memory manager in this setting is constituted by requirements concerning the correct implementation of its functionality, i.e. that a contiguous (and possibly aligned) region of memory with the desired size is taken out of or returned into the correct page pool. In addition, for separation, one has to show, that the functions of the memory manager in themselves preserve the partitioning of memory. Moreover it must be assured that the allocation or deallocation

functions are never abused by a caller to transfer memory from or to a "foreign" partition.

B. Memory Manager Implementation

In the context of this paper we consider a simplified implementation of a run-time memory manager, abstracted from a more complex PikeOS memory manager, which we also verified applying the methodology described here. The abstracted manager keeps track of free fixed-size memory objects. From a verification perspective the value the size has been fixed at does not really matter, but for clarity of the following exposition let us assume that we deal with blocks of (depending on the hardware's MMU) typically 4096 bytes, called *pages*. Our manager allows to allocate and free memory regions of variable sizes that fit within a page. However, the verification approach does not rely on this fact and would also apply to implementations which allow allocation of objects that are composed of multiple pages. The source code of the memory manager and its functions including all annotations for the formal proofs is available online [14].

In Figure 2 we follow the life-cycle of a memory page over typical use period where it is allocated as an object of type `task`. Phase 1 depicts the memory layout at boot-time. Then the partition objects which hold the partitions kernel memory resources at run-time are created. In the next step the boot memory manager distributes kernel memory pages among the partitions `P1` and `P2`. Phase 4 shows the state after allocation of a `task` object via the kernel manager. A kernel thread of `P2` has called the allocation function of the memory manager, requesting a page of memory from its partition's page pool. Then the memory is casted into an object of type `task`. Note that only a part of the page is used by the allocated object and the remaining chunk is depicted as `rest`. These rests are implicitly split off the page by the type cast and must be considered in the separation proof as well. In the final step the task is deallocated again using the memory managers `free` function. The memory it occupied is merged with `rest`, resulting in a page that is returned in to the page pool of `P2`.

The manager functions `alloc` and `free` essentially manipulate the organisation of memory chunks in the pages pools. In our example, the pages are stored using a standard doubly-linked list implementation. The aforementioned type casts were a tricky part in the verification of the manager function use cases because of the implicit splitting of the pages and the reinterpretation of plain memory chunks as typed objects. However, the primary challenge was not proving the functional correctness of the memory manager, but translating the desired separation properties into the annotation language.

III. Memory Separation Theorem and Proof

There is a multitude of concepts addressing the formulation of security properties. A classic approach in restricting information flow is to aim for a process non-interference property [15], which basically states that the execution of a low sensitivity thread is not influenced by the actions of other threads affecting high sensitivity data. The Bell-LaPadula

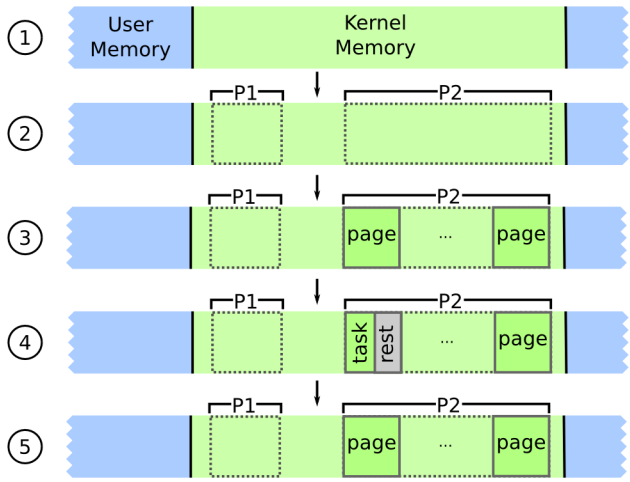

Fig. 2: Life-cycle of a page. Initialization (steps 1 and 2) is only done once at boot-time.

model [16] on the other hand focuses on a hierarchy of security levels and the adherence to an access policy in every state of the system. For separation kernels Greve, Wilding and Vanfleet introduced a formal security policy [17].

However, considering our strict separation scenario, these models appear to be too fine-grained. We follow a more straight-forward approach in formulating the security property. We present a paper & pencil proof to derive a global separation property from the formally proven property that separation is preserved by the memory manager.

Separation property: G: All memory accesses in the kernel preserve an initial disjoint partitioning of memory, and obey a security policy where a thread is only allowed to access memory from its assigned partition.

Correctness of the memory manager:

- M_1: The memory manager has the desired functionality, i.e. it provides pages from the page pool and puts the pages back when it is requested.
- M_2: A thread may only allocate or free memory belonging to its partition.
- M_3: A call to the memory manager does not alter the partitioning of memory locations.

M_1, M_2 and M_3 are proven with the help of VCC and are the outcome of the verification engineer's work: to have made explicit the memory each function is allowed to operate on and to have selected annotations that are appropriate for the code under verification. Our choices will be given in Section V.

Assumptions for the proof:

- A_1: Encapsulation – Only the kernel functions operating on the memory manager access memory manager data structures. The kernel data structures are not accessible to user-space applications.
- A_2: Rely-guarantee methodology – The preconditions of the functions operating on the memory manager hold

when they are called, e.g. relevant kernel data structures are in a well-defined state. These functions are executed atomically.

- A_3: Remaining proof effort – Kernel thread memory accesses outside the slice of functions part of memory manager functionality are restricted to locations assigned to the thread's partition.
- A_4: Low level assumptions – The kernel code is compiled by a correct compiler and runs on hardware that implements its specification correctly. We assume the absence of errors due to physical hardware faults or bit glitches.
- A_5: Soundness of the tool's methodology – The verification approach of VCC, its axiomatization of C as well as its memory and concurrency model are sound, so that any property holding in the framework of VCC holds also for the running C program.
- A_6: Soundness of annotations – The formal specifications we devise are adequate, encoding the properties we have in mind correctly. Moreover our annotations are consistent, thus they neither contradict themselves nor introduce unsoundness into the proof VCC conducts. Further, we only add specification constructs and do not alter the original implementation by our annotations.
- A_7: Tool correctness – Our toolchain (VCC/Boogie/Z3) implements its code verification methodology correctly. The verification conditions that VCC generates imply the semantics of corresponding annotation constructs.

Theorem: $M_1 \wedge M_2 \wedge M_3 \wedge A_1 \wedge ... \wedge A_7 \rightarrow G$.

Proof: We prove the theorem by induction on steps of a kernel execution sequence.

1) Induction base: The partitioning is established by the boot memory allocator before the first induction step. Current work considers a (simplified) initialization example, showing that the separation property G not vacuously holds.
2) Induction step: Assume G holds for the previous state. In principle there are many ways of splitting the execution sequence into induction steps. Here, without limitation of generality, an induction step consists either (a) of a run through an atomic function operating on the memory manager function, or (b) a run through something else.
   a) The induction step consists of a function that operates on the memory manager. By A_1 this function must be a memory manager function. Also, we can rely on the preconditions for executing this function to hold (A_2). Finally, VCC has proved the function's implementation to respect its specification, thus it is guaranteed that the properties M_1, M_2 and M_3 we encoded in our annotations hold within VCC's framework. Assumptions A_4 to A_7 give that these properties also hold for the running C program. From M_2 and M_3 our desired property G follows directly.
   b) The induction step consists of a run exclusively consisting of a slice through functions that are not part of the memory manager. Then the mapping of memory is preserved and memory accesses obey the security policy, regardless whether we speak of user space (A_1) or invocations of other kernel functions (A_3). So after this step, G still holds.

Therefore, G holds for all kernel execution sequences.

Justifications for the assumptions: In principle, A_1 to A_3 given enough verification engineer resources, also could be proved from *within* VCC. Assumptions A_5 to A_7 are about VCC, its methodology and our annotations.

- A_1: The absence of interference from userspace can be shown by checking that privilege separation between kernel mode and user based on hardware address space separation (MMU) and the usage of hardware supervisor mode is implemented correctly. It is a strong assumption that only the memory manager operates on memory manager data structures. Practically, the operations of the memory manager are the most likely cause for a misfunction of the memory manager. In the system under verification we have formally verified non-memory manager functions of the core security architecture and shown that they do not operate on the partition data structures. For other functions in the system, where due to lack of time resources a formal verification has not been done yet, we can justify this property from the design (e.g. interprocess communication is not intended to access the memory manager data structures), and verify it by code inspection. In this respect, formal verification is not a "magic bullet" but gives us incremental evidence for the functions formally verified.
- A_2: Rely-guarantee methodology – This is also a lemma that can be proved in VCC. However, function requirements may be propagated to the caller functions, so that one might need to verify an increasing number of functions in order to discharge the memory manager's preconditions. As we consider a single processor setting, guaranteeing atomic execution boils down to preventing any interruption of the program, e.g. by disabling interrupts before entering these functions.
- A_3: Remaining proof effort – The separation properties for other kernel functionality can be proved using the our verification methodology instantiated here for the memory manager.
- A_4: Low level assumptions – The correctness of compilers and hardware is usually ensured via extensive testing and often part of the trusted environment. Nevertheless the pervasive formal verification of realistic systems is a feasible task [18].
- A_5: Soundness of the tool's methodology – Concerns like these are easily risen and in general difficult to assuage. Trust into the verification methodology of VCC can be gained from the papers released by Microsoft Research (e.g. [19]) describing the underlying approach. In addition the axiomatization of VCC's specification constructs is openly available and can be checked for unsoundness using formal methods. Mathematically justi-

fying the computational model and C semantics suggested by VCC is ongoing work at Saarland University.

- A_6: Soundness of annotations – Firstly, we only add specification constructs and do not alter the original implementation by our annotations. Moreover, this is partly a matter of experience in working with the tool. Having excessively used VCC in close cooperation with the developers, we have high confidence that we chose the right annotations for our specification. Concerning possible unsoundness VCC offers checks to test for inconsistencies. Moreover VCC has built-in checks that rule out specification code to affect the implementation.
- A_7: Tool correctness – Here classical software engineering techniques are applied to ensure tool correctness. For this, the software testing methods have to be adapted towards testing formal verification tools, amongst others, by defining appropriate coverage criteria [20]. Proving the verification condition generation to be correct would also be possible, albeit costly, given formal VCC semantics. Another approach would be to employ (and formally verify) proof checkers for the output generated by the Z3 theorem prover.

IV. Deductive Code-Level Verification with VCC

Before going into the details of the actual specification and verification effort, a brief introduction of our code-level verification tool VCC is in order. For more details, see [21].

Specifying functional properties of programs is possible with VCC by using contracts on methods of the implementation as well as invariants on the state of the program.

Function contracts may contain pre- and postconditions (given by `requires` and `ensures`) as well as `writes`-clauses. A precondition of a method defines properties of the state of the program that have to hold before executing the method (in the *pre-state*). If these conditions are met, the postcondition captures the effect of the method by describing properties of the state after execution (the *post-state*) of said method. Postconditions may refer to the pre-state of a method by enclosing access to data by the keyword `old`.

Methods are only permitted to modify global state that is mentioned in a writes-clause. This allows the prover to deduce that all other global data structures are left unchanged when executing a method. To express that certain data mentioned in the writes clause do not change between pre- and poststate, they can be specified to be `unchanged`.

Compared to function contracts, invariants capture properties of the state of the program throughout its execution and are defined in the context of objects. For this, the VCC methodology superimposes the notion of objects on top of the C memory model. For example, C structs are considered to be objects, as well as user defined groups within such structs. Invariants on such objects may define valid or consistent states of an object or describe relationships between data of different objects.

Within specifications a separate specification state (or *ghost state*) is available that is not observable from within the executing C program. This ghost state can for example be used to keep track of intermediate values of variables or to record abstractions of the program's state, e.g. abstracting from details of data structure implementations. Such abstractions can be attached to data structures via coupling invariants, that state the relation between the abstraction and its implementation.

Another feature of VCC that we use extensively for the specification of our separation property is the notion of *ownership*. Each object in the program (directly or transitively through other objects) belongs to exactly one owner. Only the owner of an object is allowed to access its contents and VCC checks this condition automatically for all variable accesses.

With the help of ownership we are thus able to structure the memory of the program in a way to enforce the intended separation properties.

V. Memory Manager Correctness in VCC

To be able to formulate and verify the intended separation properties with VCC, three major components of the specification are necessary:

(A) Abstract representation of the implementation's data structures.
(B) Specification of memory separation between partitions with the help of VCC's ownership model.
(C) Function contracts of the memory management functions of the kernel stating that global memory distribution between partitions is left unchanged.

In the following, we will describe how each of these components are implemented in our example with the help of the VCC methodology.

A. Abstraction of Implementation Data Structures

Our implementation uses doubly-linked lists which are a common operating system design pattern. To be able to reason efficiently about the contents of a doubly-linked list, the list is abstracted to a set in ghost state containing all elements of the list. Consistency between the list contents and its ghost state counterpart is ensured by several invariants on both representations. Our work illustrates how to use list annotations provided with the VCC distribution [22], which we had to adapt for our use case.

B. Representing Memory Partitioning via Ownership

In order to keep apart and restrict access to the memory chunks of each partition, we need to keep an account of the set of memory pages as allocations and deallocations are performed on the partition's memory pool. Using VCC, a natural candidate to encode the set of memory blocks belonging to each partition is ownership.

For each data type A that can be allocated and casted from the memory pool of a partition, we introduce a ghost-state manager object that owns all allocated objects of type A. As the size of these data types is often smaller than the size of the smallest allocatable memory chunk (a page), an unused memory block remains (cf. Fig. 2, named *rest*).

This rest is not explicitly taken care of in the implementation. For verification purposes, we have to introduce a pointer to this memory block to be able to maintain the ownership property between the block and the associated partition – ensuring that no thread belonging to another partition may access these memory locations.

In our example, we used a straight-forward and concise way to specify and reason about the partition's memory state based on the ownership relations: by encoding ownership into a map from pointers to partition identifiers. Invariants on the partition structure ensure consistency between this map and the ownership relations between the partition and associated memory blocks.

Moreover it is a system invariant, established at kernel entry and thread switch, that the currently running kernel thread owns its partition data structure. Hence it transitively owns and can access the memory assigned to its own partition. Conversely, a thread never owns a partition that it does not belong to and therefore all accesses to corresponding memory locations will be detected by VCC as ownership violations.

C. Function Contracts

We have the following code-level verification goals:

- C_1 Functional correctness. The correctness is expressed as function-specific annotations. For the run-time memory manager functional correctness ensures that an object that has been created after successful casting is owned by the appropriate manager and that proper accounting of the manager's size (i.e., the number of owned elements) is done.
- C_2 Objects and local memory map. Each partition can own different objects (that are in different containers). It needs to ensure that these objects do not overlap, this is realized by a map embedded into the partition data structure itself (partition-local map). The coupling invariant may be relaxed for the object under memory reinterpretation while running through the function but it is guaranteed to hold always at function entry and exit. As we require the partition object to hold and because only threads can only own their associated partition, we ensure also that threads can use the memory manager allocation functions to access memory from their partitions only.
- C_3 Conservation of the global partitioning map after boot. For the memory manager verification, the most important property is that the initial assignment of memory to partitions remains unchanged. We do this by a defining a global memory map and requiring that there exists a coupling between the global memory map and each partition-local memory map. Moreover, we specify that after booting the global memory map stays unchanged.

The goals C_1, C_2 and C_3 correspond to M_1, M_2 and M_3 from Section III. The goal C_2 allows us efficient usage of VCC's object-oriented approach. Figure 3 presents the verification goals (C_1, C_2, and C_3) as ascertained in the function contract annotations of function `task_alloc` in a simplified syntax. We require the partition-local pointer map to equal the invariable global memory map before and after the function call. Also the ownership invariants of the partition must hold and the current thread must own the partition object it is accessing. From these properties follows the preservation of memory partitioning. Functional correctness is expressed by the claim that either a page of memory was transformed into a task object owned by the partitions task manager or the allocation was unsuccessful and the partition is unchanged.

```
1  task_t *task_alloc(partition_t *p, unsigned int id)
2  requires
3    ∀i. i > 0 → p->memmap[i] = memmap_g.memmap[i] //C3
4  ∧ p->relax == 0 //C3
5  ∧ p∈owns(current_thread)∧invariant(p) //C2
6
7  ensures
8   unchanged(memmap_g.memmap) //C3
9  ∧ unchanged(p->memmap) //C3
10 ∧ ∀i. i > 0 → p->memmap[i] == memmap_g.memmap[i] //C3
11 ∧ result →
12       ( result∈owns(p->man_task) //C1
13         ∧ p->man_node.Manager->size ==
14             old(p->man_node.Manager->size) - 1 //C1
15       )
16 ∧ ¬result → p->man_node.Manager->size ==
17             old(p->man_node.Manager->size) //C1
18 ∧ result∉owns(p->man_node.Manager) //C1
19 ∧ unchanged(p->id) //C2
20 ∧ p->relax == 0 //C2
21 ∧ p∈owns(current_thread)∧invariant(p) //C2
22
23 writes(p) //C3
```

Fig. 3: The verification goals C_1, C_2, and C_3 are represented in the assertions for a function contract (explanation of technical details can be found in the `README.txt` of [14]).

VI. Discussion

A. Security Property Assurance Gained

What has been shown: We have a proof that it never happens to the memory manager to misallocate memory from one partition to another. In addition we ensure for an exemplary set of use cases that via the memory manager threads can only access memory locations belonging to their own partition.

The separation property we have shown is non-functional: we have shown that during run-time of the separation kernel, there is no unintended run-time information flow between partitions, due to misfunction or misuse of the memory manager. As we are talking about the *absence* and not the presence of a behavior in this form such a non-functional property is generally hard to prove [23].

Limitations: It is important to understand the limits of what has been verified and what not. In our case, other functions than the allocation functions could impair the use of the memory manager or there may be memory manager function invocations in the kernel which do not fulfill the ownership requirements. These issues either can be ruled out by traditional human source code inspection or in future by also having all other functions and use cases verified by VCC.

In our separation property we cover one source of errors, i.e. the run-time memory manager correctness. In this demonstration we have *not* covered all possible scenarios which could breach memory separation in the kernel. However we have addressed the assumptions of our approach and pointed out the remaining challenges that must be met to complete the separation proof.

Another possible drawback of our approach is the focus on a strict separation property. Currently systems allowing controlled communication are not supported. Nevertheless from our experience we see no major obstacle to extend the approach and find ownership structures incorporating the permitted shared data accesses.

For such systems that allow to share data between partitions, however, focussing on information flow may be an alternative approach. Note that one can trace information flow by labeling all pieces of information appropriately (and VCC is also being extended to support this [24]).

B. How the Approach Scales

The meaning of "scaling" in this context is twofold. On the one hand it can refer to the scalability of our code-level annotations to a bigger set of use cases or more complex code. In general all functions of our example verify in a relatively short time (about 3 minutes on average on a 2 GHz single CPU 1 GB RAM machine) which still allows for an interactive verification process. When increasing the complexity of the example, e.g. by increasing the number of object types that could be allocated and held by a partition, no real performance loss was observed, although a quadratically growing number of disjointness invariants, stating that elements in one object manager structure do not occur in another, had to be proved.

From a proof development point of view we believe the approach to scale as well, though there is no empirical data available supporting this claim. Devising the ownership structure and memory map for the example and completing the formal proofs took an effort of approximately 2 person months. Given that the overall ownership specification is now already fixed, transferring the experience from the memory manager to another kernel component in order to prove the separation property appears to be be an easy task.

C. Verification Engineering Aspects

A general challenge that we discovered while dealing with a complex system like the PikeOS kernel, was the modelling of the global state of all kernel data structures. This is usually not necessary in projects that aim at a verification of "local" algorithms.

Partitioning via ownership: We have found the ownership feature of VCC very appropriate to model the desired strict separation of memory in our annotations. Since ownership of an object is exclusive and VCC checks all memory accesses for compliance with its ownership policy, illegal memory accesses are easily detected.

Finding the right reification into specification objects: The innocuously simple Figure 1 we had started out with in this form actually is not taken from any design document but the result of working out the proof in VCC (e.g. the reification of "pages as bytes"). We think that this representation adds maximal clarity for thinking about the footprint scope of the memory management functions. Further reifications were the specification state rest fields, and the different containers ("managers") assigned to a partition, and of course the `memmap` representation of the otherwise implicit memory state.

Also finding the right place to put invariants and assertions needed some experience, an example was the decision to bind some properties of the `rest` field of the `node_t` to the partition and not the node, because the size of the rest depends on where the list is used (e.g. it is different in a scheduler queue) and is thus better kept at the partition level.

Recurring annotations: In the verification of similar use case of the memory manager functions, e.g. allocation functions for different types, it was discovered that certain blocks of annotations repeated themselves and could be made generic using parametrized macros.

Testing the specification: The proof meta-argument and the code annotations given here have evolved after a good amount of experimentation with the VCC tool. In case the verification engineer distrusts his/her specifications that might have overlooked something, he/she has evidence at hand that the transformation was successful and the annotations are adequate, when she can use the result, e.g. for the low-level page allocation function TR2 in Fig. 1, we gain assurance because it is used by the high-level task allocation function `task_alloc` (TR3 in Fig. 1; Fig. 3). Again, in the example, the `task_alloc` function is checked against by a `lifecycle` function that repeats the idea of the life-cycle example from Figure 2. Moreover, we have enabled a VCC heuristic that searches for unsound specifications such as cycles in an ownership graph or inconsistent statements ("smoke-checking"). While this is not needed for running the examples, in general we also recommend to enable testing for inconsistencies by the "`/b:/smoke`" switch.

Harnessing the Proof State: In the current VCC (or Frama-C) approach the verification engineer only interacts with the proof state via choosing invariants and asserts. At least from our experience, it would be desirable to have more fine-grained control over it. For example, being able to flush (a part of) the proof state would be desirable to avoid cluttering the prover with unnecessary information.

Certification aspects: DO-178B/C: The use of formal methods in avionics is encouraged in the DO-178C formal methods supplement [25]. In particular, the supplement requires the use of a *formal model* which in our case is encoded as annotations for properties to be shown. Moreover, in the accompanying discussion paper [26], there is an example where code annotations are matched to low-level requirements (Airbus "unit proofs", the annotations in the tool Caveat [27] look very similar to those in VCC) and another example where

low-level requirements are matched against high-level requirements (Rockwell Collins). In this setting, the challenge we have answered here, (although only for exploitation of faulty implementation or use of a memory manager) is to span from code annotations all the way up to a high-level requirement, for example the partition separation evidence gained here for an integrated modular avionics (IMA) setting [28].

VII. Wrapping Up

We have shown that the memory manager is well-behaved, and that no interference between partitions can appear. Based on Hoare-style code annotations we have derived a global non-functional security requirement. We have presented how to prove this separation property conceptually in Section III and formally proven the correctness of the memory manager with the VCC tool in Section V, using the ownership mechanism of VCC. We also successfully realized part of our concept in another verification tool Frama-C which is one more fact for generality of our approach. Future work includes the completion of the overall separation proof as well as the application of code level verifcation to validate other requirements on the PikeOS microkernel.

Acknowledgment

The authors acknowledge joint work with Jérôme Creci, Dilyana Dimova as well as fruitful discussions on the VCC codeplex forum [6], in particular with Mark Hillebrand. We acknowledge discussions with Eyad Alkassar, Frank Dordowsky, Sabine Schmaltz and Virgile Prevosto. This work was partially funded by the German Federal Ministry of Education and Research (BMBF) in the framework of the Verisoft XT [1] project under grant 01 IS 07 008 as well as the TECOM Consortium [2] (Grant Nr. 216888).

References

[1] "The verisoft xt project, bmbf grant nr. 01 is 07 008," http://verisoftxt.de/.

[2] "Tecom consortium, grant nr. 216888," http://tecom-project.eu/.

[3] R. Kaiser and S. Wagner, "Evolution of the PikeOS microkernel," in *MIKES: 1st International Workshop on Microkernels for Embedded Systems*, I. Kuz and S. M. Petters, Eds., 2007.

[4] RTCA SC-167 / EUROCAE WG-12, *DO-178B: Software Considerations in Airborne Systems and Equipment Certification*. Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L St. NW., Suite 805, Washington, D.C. 20036, December 1992.

[5] SYSGO AG, "PikeOS RTOS Technology," http://www.pikeos.com.

[6] Microsoft Research, "VCC homepage," http://vcc.codeplex.com.

[7] G. Klein, "Operating system verification — an overview," NICTA, Sydney, Australia, Tech. Rep. NRL-955, June 2008.

[8] G. Klein, J. Andronick, K. Elphinstone, G. Heiser, D. Cock, P. Derrin, D. Elkaduwe, K. Engelhardt, R. Kolanski, M. Norrish, T. Sewell, H. Tuch, and S. Winwood, "seL4: Formal verification of an OS kernel," *Communications of the ACM*, vol. 53, no. 6, pp. 107–115, Jun 2010.

[9] E. Alkassar, M. Hillebrand, W. Paul, and E. Petrova, "Automated verification of a small hypervisor," in *Third International Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'10)*, ser. LNCS. Springer, 2010.

[10] W. D. Young and W. R. Bevier, "Mathematical modeling and analysis of an external memory manager," in *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, ser. LNCS, vol. 1313. London, UK: Springer-Verlag, 1997, pp. 237–257.

[11] H. Tuch, G. Klein, and M. Norrish, "Types, bytes, and separation logic," in *Proc. 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'07)*, M. Hofmann and M. Felleisen, Eds., Nice, France, Jan 2007, pp. 97–108.

[12] Information Assurance Directorate, "U.S. government protection profile for separation kernels in environments requiring high robustness. Version 1.03," June 2007.

[13] W. M. Vanfleet, J. A. Luke, R. W. Beckwith, C. Taylor, B. Calloni, and G. Uchenick, "MILS: Architecture for high-assurance embedded computing," *Crosstalk*, August 2005.

[14] "Sources of the memory manager example," http://www.verisoftxt.de/d/src/memorySeparation.tgz.

[15] J. A. Goguen and J. Meseguer, "Security policies and security models," *Security and Privacy, IEEE Symposium on*, vol. 0, p. 11, 1982.

[16] D. E. Bell and L. J. LaPadula, "Secure computer systems: Mathematical foundations and model," *MITRE CORP BEDFORD MA*, vol. 1, no. M74-244, p. 42, 1973.

[17] D. Greve, M. Wilding, and W. Vanfleet, "A separation kernel formal security policy," in *Proc. Fourth Int'l Workshop ACL2 Prover and Its Applications*, Jul. 2003.

[18] E. Alkassar, M. A. Hillebrand, D. Leinenbach, N. W. Schirmer, and A. Starostin, "The verisoft approach to systems verification," in *2nd IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE'08)*, ser. LNCS, N. Shankar and J. Woodcock, Eds., vol. 5295. Springer, 2008, pp. 209–224.

[19] E. Cohen, M. Moskal, S. Tobies, and W. Schulte, "A precise yet efficient memory model for C," *Electron. Notes Theor. Comput. Sci.*, vol. 254, pp. 85–103, 2009.

[20] T. Bormer and M. Wagner, "Towards testing a verifying compiler," in *International Conference on Formal Verification of Object-Oriented Software (FoVeOOS)*, B. Beckert and C. Marché, Eds., vol. KIT-INFO-TR 2010-13. Karlsruhe Institute of Technology, Technical Report, Jun. 2010.

[21] E. Cohen, M. Dahlweid, M. Hillebrand, D. Leinenbach, M. Moskal, T. Santen, W. Schulte, and S. Tobies, "Vcc: A practical system for verifying concurrent c," in *Theorem Proving in Higher Order Logics*, ser. Lecture Notes in Computer Science, S. Berghofer, T. Nipkow, C. Urban, and M. Wenzel, Eds. Springer Berlin / Heidelberg, 2009, vol. 5674, pp. 23–42.

[22] Microsoft Research, "List.c: "This file provides a sample implementation of doubly-linked lists."," https://vcc.svn.codeplex.com/svn/vcc/Test/testsuite/examples/List.c(accessed19July2010).

[23] M. Glinz, "On non-functional requirements," in *Requirements Engineering, IEEE International Conference on*. IEEE Computer Society, 2007, pp. 21–26.

[24] F. Dupressoir, "Development: wrapping pointer comparisons in casts?" http://vcc.codeplex.com/Thread/View.aspx?ThreadId=214955.

[25] D. Brown, H. Delseny, K. Hayhurst, and V. Wiels, "Guidance for using formal methods in a certification context," in *ERTS^2 2010: Embedded Real Time Software and Systems*, May 2010, pp. 1–7.

[26] D. Brown, K. Hayhurst, and SC-205/WG-71 SG-6, "IP0602 Revision E: Formal methods discussion paper," June 2010.

[27] J. Souyris, V. Wiels, D. Delmas, and H. Delseny, "Formal verification of avionics software products," in *FM*, 2009, pp. 532–546.

[28] RTCA SC-200 / EUROCAE WG-60, *DO-297: Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. Radio Technical Commission for Aeronautics (RTCA), Inc., 1828 L St. NW., Suite 805, Washington, D.C. 20036, November 2005.