Interference Scenarios for an ARM64 Linux System

1. Index

2. Documentation Control

Item	**Description**
Title	Interference Scenarios for an ARM64 Linux System
Author(s)	Igor Stoppa
Reviewers List	Aingara Paramakuru Deepak Nibade Eli Gurvitz Hadi Zibaeenejad Lauri Ora Michal Szczepankiewicz Raghavendra Vishnu Kumar Raman Kishore Sanjay Trivedi Shrife Mohamed Vito Magnanimo
Revision	1.0
Date	Jan 17, 2024

3. Terms and Abbreviations

ASID	Address Space Identifier	Value set by the kernel and used by the MMU for automatically tagging TLB entries belonging to different contexts. The MMU will use only TLB entries that are tagged with the currently active ASID.
ASILn	Automotive Safety Integrity Level n	The qualification of integrity used to define in a standardised way a set of properties of a system, in the Automotive industry. They go from ASIL D, more restrictive, to ASIL A, less stringent.
CFI	Control Flow Integrity	Mechanism used (nowadays in Linux through compiler extensions) to thwart attacks based on Return Oriented Programming or Jump Oriented Programming
Detection		This term has a certain meaning in Fusa Context, however here it represents the ability to take notice of an interference affecting a component with allocated FFI requirements. It applies to interference originating from components at a lower safety integrity level.
ELn	Exception Level n	The execution context at which certain code is executed: n = 0 mean what is traditionally used for user-space n = 1 means what is traditionally used for the kernel n = 2 means what is traditionally used for the hypervisor n = 3 means what is traditionally used for the secure mode (not used in this document)
Exception (ARM definition)		Event which has the potential for diverting the execution flow. In ARM parlance, an exception can be either synchronous or asynchronous. *Synchronous*: an event triggered by the regular execution flow. While it is not always certain that a specific action will result in an exception, it is at least expected that such an event might happen (which is what in Linux is effectively called exception). *Asynchronous:* an event which is either triggered by a software error (still called exception, in Linux) or by an external hardware component, like either an IRQ, an FIQ or an NMI (in Linux called interrupt, fast interrupt and non

		maskable interrupt respectively)
Exception (Linux definition)		Synchronous transition between execution contexts, from lower to higher privilege, driven by the execution flow.
FFI	Freedom From Interference	See definition 3.65 from ISO 26262 Part 1 - Vocabulary
FIQ	Fast Interrupt Request	It's a specialised type of interrupt which, in its hardware implementation, has a more direct path to the CPU, without being routed through as many IP blocks like a regular interrupt, which typically is routed through an interrupt controller. The FIQ is indeed faster, at the cost of occupying one hardware line that could be otherwise used for connecting e.g. an interrupt controller. The associated benefit is reduced latency, for applications where latency is critical.
FuSa	Functional Safety	**Functional safety** is the part of the overall safety of a system or piece of equipment that depends on automatic protection operating correctly in response to its inputs or failure in a predictable manner (fail-safe). The automatic protection system should be designed to properly handle likely human errors, systematic errors, hardware failures and operational/environmental stress. Detailed definition.
Hazard		See definition 3.75 from ISO 26262 Part 1 - Vocabulary
I2C	Inter-Integrated Circuit	Bus interface connection protocol incorporated into devices for serial communication. Typically used for relatively slow peripherals.
Interference		See FFI / Freedom From Interference
IPC	Inter Process Communication	Generic reference to the mechanism (there can be multiple implementations) used by processes to communicate with one another; it can refer to synchronisation primitives, message passing, signalling.
IRQ	Interrupt Request	Asynchronous transition between execution contexts, usually from lower to higher privilege, but also within same privilege, as long as it is sufficient, driven by the hardware

Interrupt		events. It can still be controlled by software, though, if the software has the ability to mask/unmask the fact that a certain interrupt has occurred.
IPA	Intermediate Physical Address	The address outputted by the first stage translation of the MMU and inputted into the second stage translation.
LTS	Long Term Support	Special versions of the Linux kernel which are chosen to be the targets for backporting selected (mostly bugfix/security) patches. They are meant to be used for actual products, which might require sticking to a certain “stable” version for long periods of time, with the intent of preventing such products from becoming targets for unpatched vulnerability and exploits.
MMU	Memory Management Unit	Component inside the SoC that primarily performs translations operations between virtual addresses and intermediate or physical ones, in support of various memory management techniques, like virtual contiguity and on-demand paging.
NMI	Non Maskable Interrupt	Interrupt line that the CPU cannot ignore by disabling it. Depending on the application, different types of sources can be connected. In safety applications it can be exploited for treating exceptional events which cannot be ignored.
ODD	Operational Design Domain	A set of operating conditions for an automated system, often used in the field of autonomous vehicles. These operating conditions include environmental, geographical and time of day constraints, traffic and roadway characteristics. The ODD is used by manufacturers to indicate where their product will operate safely.
OS	Operating System	An operating system (**OS**) is system software that manages computer hardware and software resources, and provides common services for computer programs.
PA	Physical Address	The address output to the second stage translation of the MMU, which is placed on the memory bus.
Prevention		This term has a certain meaning in Fusa Context, however here it represents the ability either to suppress or to prevent from happening, an interference, so that it doesn’t affect a

		component with allocated FFI requirements. It applies to interference originating from components at a lower safety integrity level.
QM	Quality Managed	Refers to the classification of non-ASIL systems, which are still developed according to a set of processes and verification criteria, less restrictive than anything rated ASIL.
Risk		See definition 3.128 from ISO 26262 Part 1 - Vocabulary
SILn	Safety Integrity Level	The qualification of integrity used to define in a standardised way a set of properties of a system, in a wide range of industry fields: aerospace, railways, etc. They go from SIL 4, more restrictive, to SIL 1, less stringent.
SoC	System on a Chip	The shorthand for the entirety of the HW components that constitute the collective of the cores, busses, and the integrated peripherals.
SPI	Serial Peripheral Interface	Bus interface connection protocol incorporated into devices for serial communication. Typically used for relatively fast peripherals.
TEE	Trusted Execution Environment	Optional execution mode of ARM cores that creates a separate context where certain features typically related to trusted computing are enabled.
Toolchain		The set of software tools that support the generation of executable binary artefacts. The actual content varies, depending on the programming language used for the source code. However, in the Linux case, at minimum it consists of: preprocessor, compiler, assembler, linker. But it is common to have additional utilities, like object files manipulation and debugging.
TLB	Translation Lookaside Buffer	Cache of address translations present within the MMU, that avoids incurring in the penalty of generating multiple memory accesses, when translating an address that had been translated recently. It also caches information about access permissions, like the read, write and execute

		permissions.
TZASC	Trust Zone Address Space Controller	ARM ip block which is controllable from safe mode and allows the configurations of memory zones which are exclusively accessible from a cpu core that is in secure mode.
VA	Virtual Address	The address in input to the first stage translation of the MMU.

Notes:

- ARM and Linux attribute different meanings to the term “Exception”, but this document will use the Linux one.

4. References

4.1. ARM64 Memory Management

4.2. Linux Memory Management

4.3. ISO 26262 Part 1 - Road Vehicles FuSa Vocabulary

Note: The Vocabulary refers to Road Vehicles, but the concepts used in the present document utilise terms that are applicable also to other safety contexts.

4.4. CC BY-SA 4.0 Deed | Attribution-ShareAlike 4.0 International | Creative Commons - https://creativecommons.org/licenses/by-sa/4.0/ License

5. Purpose of the document

This document describes some of the most relevant cases of interference that can happen within the Linux kernel and how they are associated with failure modes.

It is common to define “safety integrity levels”, referring to groups/layers of components - both software and hardware - which must meet similar classes of safety requirements.

These requirements are not only functional, but they also influence related processes, such as design, testing, analysis and so on.

The interference between software components can happen at both different and same safety integrity levels; however, it is expected, as part of the definition of said levels, that each level shall also dictate what considerations can be made about same-level interference between components. And even about self interference.

In practice, a higher safety integrity level assigned to a component implies more rigorous qualification processes. Such higher rigour makes it less likely that it will interfere both with itself and with other components belonging to the same safety integrity level, than what can be expected from a different component, with lower safety requirements.

However, these considerations rely on the assumption that inter-level interference is managed.

Management of same level interference can translate to different practical effects, depending on a variety of factors, but it is expected that any sort of interference which is relevant for safety purposes will not be ignored, even if not all interference scenarios might be approached in the same way.

The document focuses on this kind of inter-level type of interference, which must be managed, as an enabler to making any further safety claims.

While not exhaustive, it provides a core set of benchmarks for evaluating how a certain safety strategy might fare in dealing with hazards and related risk.

The document will attempt to be as generic as possible, however, whenever an architecture or platform specific mechanism needs to be considered, it will refer to an ARM64 architecture with support for EL2.

The safety analysis addresses both various upstream versions of unmodified Linux kernels and derived ones, like the LTS kernel versions; all of them prior to introducing any mitigation that might be deemed necessary for safety purposes.

The notions about safety used here are compatible with various applications(aerospace, automotive, railways, robotics, etc.), even if the terminology and the process-oriented criteria might change from field to field.

This document assumes that the Linux kernel can be either considered as a QM artefact, or that if deemed necessary, further actions can be taken to elevate it at QM state, such as performing code reviews, system analysis and testing.

6. Structure of the document

- The first section lists some basic characteristics of the hardware components involved in the safety analysis, and how Linux uses them. It is not meant to be a full explanation, but merely a reference for the considerations derived in later sections.
    - The first and second subsections focus on the hardware features.
    - The third subsection describes memory management in Linux.
- The second section provides considerations that should guide the analysis.
- The third section describes major sources of interference.
- The fourth section shows how interference can affect the system analysed.

7. Safety-Relevant System Features

For the purpose of this document, the following statements are made with respect to the cores which are treated as part of the primary system. Other cores might be present, each with its own SW stack, that effectively act as smart peripherals, within the SoC or anyways within the package. These are not taken into account in the following chapters, because this very same analysis could be applied iteratively to them.

What follows is a description of the system under analysis, establishing some key facts and implications deriving from them, which will be useful when analysing failure modes, later on.

7.1. SoC, Cores and Exception Levels

7.1.1. A SoC is the hardware component which contains one or more execution units capable of executing programs, and a number of peripheral devices, able to control other external components.

A SoC can thus support multiple concurrent execution flows and control multiple external devices simultaneously.

7.1.2. A core is an execution unit, inside a SoC, that executes programs.

In some architectures, a core has more than one context of execution, and in such cases, when considering the number of parallel streams of execution within a SoC, the number of execution contexts should be considered instead of the number of cores. However, on ARM64, one core equals one hardware thread of execution.

7.1.3. Cores are not necessarily homogeneous, at least from non-functional perspective, with certain cores being optimised for lower consumption, while others have been optimised for higher performance.

7.1.4. Within a SoC, usually all the cores have identical access rights and capability, with regard to shared peripherals.

However, some peripherals might be (more) tightly coupled with certain cores than with others.

7.1.5. At low level, the core is a complex state machine which operates on a set of input, outputs and internal states. States are implemented mostly as registers; which the core can read and/or write.

7.1.5.1. Directly, through the execution of specific instructions

7.1.5.2. Indirectly, as result of other operations performed

7.1.6. On the ARM64 architecture, registers are accessible through 3 different mechanisms:

7.1.6.1. Direct access: reserved for the registers contained within the core; the access is performed through dedicated instructions, which can refer to registers in two ways:

7.1.6.1.1. implicit reference, used for certain special registers (e.g. the link register)

7.1.6.1.2. explicit reference, used for general purpose registers (e.g. the operands of a mathematical operation)

7.1.6.2. Coprocessor access: reserved for tightly coupled peripheral components, like the MMU, which have specific numerical IDs and are accessed through specific coprocessor-oriented instructions.

7.1.6.3. Memory-mapped: certain non-core peripherals, which often are optional or can sometimes be present in multiple instances, are connected to the memory bus, so that they are accessible through the same mechanisms used for reading/writing memory.

7.1.7. Each core supports and can transition, independently from the other cores, between 4 exception levels:

7.1.7.1. EL0, user-space

7.1.7.2. EL1, operating system, typically has higher privileges than EL0

7.1.7.3. EL2, hypervisor, typically has higher privileges than EL1

7.1.7.4. EL3, monitor mode, also known as secure mode, the highest privilege mode. Its presence is discretionary and the decision is left to the hardware designer.

In ARM parlance, the secure mode is called Trust Zone, and it was designed for enabling the execution of a Trusted Execution Environment.

When the security extensions are present, it introduces alternate versions of the previous exception levels, called Secure-ELn or S-ELn. And together they go under the moniker “Secure World”, as opposed to the others, which are treated as Non-Secure.

The presence of an EL3 does not automatically imply the existence of all the S-ELn levels - the ARM specifications define many features as optional.

If present, these secure world exception levels have their own separate set of system registers and they have access privilege over the non secure world. More on this later.

7.1.8. Transitions between exception levels are either exceptions or interrupts.

7.1.9. The instruction set includes means for an exception level to directly transition the flow of execution to a higher level (which will have its own handler, to process the invocation).

7.1.9.1. The invocation of EL1 services is a SVC (system call).

7.1.9.2. The invocation of EL2 service is an HVC (hypervisor call).

7.1.9.3. The invocation of EL3 services is an SMC (secure mode call).

7.1.10. Exceptions are serviced through dedicated stacks, while interrupts are serviced using the stack currently in use on the core receiving the signal at the time the interrupt is handled.

7.1.11. While there may be custom deviations in some specific implementations, the typical (very simplified) boot sequence places the core first in EL3, to guarantee that the system state is not affected by any other software that might be running from a less trusted context.

The EL3 can then proceed to initialise hardware peripherals, load/validate/execute programs at lower level of trust and so on.

7.2. Memory accesses and the TZASC

7.2.1. The TrustZone Address Space Controller acts like a firewall on the memory bus, preventing a core that is not in secure mode from accessing any of the memory zones that have been configured as secure-only.

The configuration is possible only from secure-mode.

This filtering operates on physical addresses and is therefore unambiguous, disregarding any address translation regimen that might be in place, because the filtering happens on the output of whatever address translation that might have happened.

7.3. Memory accesses and the MMU

7.3.1. The MMU plays a central role in ensuring memory isolation between contexts under execution.

7.3.2. During operational state, the MMU is active.

However, there is a short transient, at boot, during which the MMU is not yet enabled

7.3.3. Memory is divided into pages, which are chunks of predefined size and matching alignment and properties. This definition attempts to be generic and therefore omits certain optional features, which can operate on smaller memory granules.

7.3.4. The MMU performs address translations, converting between one memory address space to another, through the use of specialised data structures, called page tables.

7.3.5. The MMU is able to make non-contiguous (sets of) memory pages appear as virtually contiguous, using a scatter-gather approach, through non-linear mappings.

7.3.6. Page tables are a formalised type of sparse tree, with nodes themselves being pages.

7.3.7. The MMU operates translation through (up-to) 2 stages:

7.3.7.1. Stage 1: Virtual Address to Intermediate Physical Address

7.3.7.2. Stage 2:

7.3.7.2.1. For addresses originating from either EL0 or EL1: Intermediate Physical Address to Physical Address

7.3.7.2.2. For addresses originating from EL2:

Virtual Address to Physical Address

7.3.8. VAs are the types of addresses that are regularly handled by the software during execution, while the PA is the address that goes on the memory bus.

In reality complex systems can also have memory management engines that are in charge of orchestrating and optimising memory accesses issued by various cores to achieve optimal memory bus utilisation, however this part can be considered as fully transparent to the MMU.

7.3.9. Each translation stage uses a dedicated page table.

7.3.9.1. The first translation stage operates either from virtual addresses (EL0 or EL1 addresses) to IPA (EL2 addresses).

The Virtual addresses used in EL0 and EL1 are from non-overlapping ranges, and each range uses a separate page table.

7.3.9.2. The second translation stage operates from IPA (EL2 addresses) to PA (MMU Bus addresses).

7.3.10. Different exception levels

7.3.10.1. EL0 can only operate with VAs, using its own page table.

7.3.10.2. EL1 operates mostly with VAs, and it also has its own page table.

However, at boot, prior to initialising its own address translation stage in the MMU, it will use IPAs.

Furthermore, besides using its own page table, it can perform EL0 accesses, through the EL0 page table, although this option is mostly kept disabled.

7.3.10.3. EL2 operates mostly with its own VAs, through its own page table.

However, at boot, prior to initialising the second stage address translation, it will operate on PAs.

But it can also emulate both EL0 and EL1 memory accesses, through their respective page tables.

7.3.10.4. However, if present and active, the second stage translation takes place also for addresses originating from EL0 and EL1; in this case it’s the IPA to be converted to PA.

7.3.11. The operation of translating from one address space to another is performed by the MMU, by using the starting address to navigate the associated tree of page tables, starting from the root (Page Global Directory - PGD) and ending with a leaf (PTE - Page Table Entry)

7.3.12. The MMU can be configured to use different page sizes, trading granularity for TLB optimization. Sizes supported are 4kB, 16kB and 64kB, with 4kB being the typical choice.

7.3.13. The page table can be up to 4 levels deep:

7.3.13.1. Page Global Directory (PGD)

While the non secure world does not have any particular constraint, the secure world requires that the PGD be chosen from a memory range that has been configured as safe in the TZASC

7.3.13.2. Page Upper Directory (PUD)

7.3.13.3. Page Middle Directory (PMD)

7.3.13.4. Page Table Entry (PTE)

7.3.14. Each level is composed of pointers to the next level - the pointers are the addresses of the pages of the next level.

Pointers are already translated.

7.3.15. To optimise the use of the TLB, it is possible to turn a branch into a leaf node representing the underlying destination range, provided that it is contiguous and aligned.

7.3.16. The page tables also implement translation attributes, like the ‘executable’ property for code pages and write protection for read-only data.

7.3.17. Attributes of leaf (the memory page being mapped) are set by the last node of the tree (typically a PTE entry).

7.3.18. In practice, of the 64 bits available in the PTE entry, only a certain number is used for the actual address of the page being mapped, and the remaining bits are used for other purposes, including the colouring of the mapping.

7.3.19. Performing a translation is an expensive operation, because the MMU needs to generate various memory accesses, navigating the page table tree.

7.3.20. Translations are not always successful; for example an address might not have a backing memory page, or an operation might be incompatible with the property associated with the memory location involved (ex: writing to a read-only page).

These events trigger exceptions, which are expected to be handled by the operating system.

7.3.21. To mitigate the cost of a memory translation, each stage of the MMU implements a cache (TLB - Translation Lookaside Buffer) which can be implemented in various ways, however it always caches not just the translations, but also their associated properties, like write and execute permissions.

7.3.22. Because of the caching, changes to a page table might not be visible, if a previous, different, translation is already present in the cache, and therefore the cache might need to be invalidated, prior to relying on the updated translation rules.

7.3.23. In EL1, the MMU supports having 2 sets of page tables programmed with different base addresses at the same time, for converting virtual to intermediate physical addresses, TTBR0_EL1 and TTBR1_EL1.

7.3.24. Each core supports having its own set of MMU page tables, as described above, independent from others, with independent TLBs that can also be maintained independently.

7.3.25. The mapping mechanism is such that, at any translation stage, multiple source addresses can land on the same destination address.

In a few cases this is the intended behaviour, and usually it has a transient nature, but in general it is unwanted.

7.3.26. Since the mapping properties are associated with the source address, the same destination address can be accessed with different properties.

7.3.27. The operating system executing in EL1 can manipulate both the core registers and the page tables used for EL0 so that multiple user-space programs can be run in time sharing on that core, without being aware of each other.

7.3.28. Similarly, the hypervisor running in EL2 can manipulate both the core registers and the page tables used for EL1 so that multiple operating systems can run in time sharing on that core, without being aware of each other.

7.3.29. Because performing page tables walks is expensive, and a suspended context (be it either in EL0 or in EL1) will resume in the same state it had when suspended, instead of allowing fully replacing of the TLB entries, it can be more effective to preserve them across context changes, as long as they are temporarily disabled.

For this purpose, it is possible to automatically tag TLB entries of the suspended context, as they are generated, by using the ASID, which is programmed as contexts are activated.

Each context is associated with an unique ASID and the MMU will ignore TLB entries tagged with an ASID that is different from the active one.

7.3.30. Similarly, the EL2 TLB entries support VMIDs for tagging cached translations, obtained from different sets of page tables associated with either different VMs or with the hypervisor itself.

7.3.31. When present, the secure world can limit memory access to selected ranges of physical pages by configuring the TZASC accordingly.

The secure world has its own set of translation tables, that can be used to access memory configured as secure, as described.

However, the Secure world is likely to require access to the non secure world as well. This need can be satisfied by grafting into the S translation tables branches of the NS ones. In such cases, the hardware implementation ensures that any grafted NS portion of translation tables cannot be made to point to pages in the S world.

7.4. Use of the ARM64 MMU in Linux

7.4.1. The Virtual mapping address space is divided into 2 ranges, one used for EL0 mappings and one used for EL1 mappings.

Each range is assigned a separate page table and both can be activated simultaneously (when in EL1), however under normal circumstances, on each core only one of them is active, at any given time.

7.4.2. In Linux it is common to talk about “user-space” as opposite to “kernel-space” and “user-mode” as opposite to “kernel-mode”.

The “xxx-space” terms refer to the split of address ranges associated with user processes and kernel execution.

The “xxx-mode” terms, on the other hand, refer to the exception level active on a certain core at a given time.

7.4.3. The operation of copying/accessing data between kernel and user-space is the exception to the rule of having only one address-space active at any time, on a given core, since it needs both mappings simultaneously active.

Therefore, a core can have simultaneous access to both user-space and kernel-space, but it cannot be both in user-mode and in kernel-mode at the same time.

7.4.4. In Linux, the page tables can have a depth that is different from what is actually supported by the HW, for example by introducing loops on a page level, to simulate 2 levels by having a page pointing back to a different slot of itself.

The use of build-time macros is a more common mechanism for collapsing levels which are formally used by the code, but will be compiled away, if unnecessary on a specific system.

This is done, for example, when the HW has just one level, instead of the pair (PUD, PMD).

7.4.5. The EL0 mappings are specific to each user-space process: each process has its own page table and is unable to interfere with other processes, as long as the underlying writable physical pages are kept assigned to one process, max.

7.4.6. Threads belonging to the same user-space process share the same page table.

7.4.7. Multiple user-space threads belonging to the same process can run in parallel on different cores, simultaneously, but they must employ techniques that ensure concurrent access to shared data will not cause corruption.

7.4.8. Each running user-space thread must have its own stack, though.

7.4.9. Unless configured otherwise, user-space processes are swappable, meaning that least-used memory pages containing read-only code can be dropped, while least-used memory pages containing data can be dumped to a specialised swap file or partition, on disk, and then dropped.

7.4.10. Linux uses on-demand loading for pages which have a valid virtual address but lack backing.

When such an address is accessed, an exception is triggered and the exception handler will schedule the loading of the related content from storage, be it code or data.

7.4.11. The EL1 mappings, instead, are shared among all the cores that run in EL1 mode.

7.4.12. EL1 can also have multiple threads, each with its own stack, however they effectively act as if they belonged to the same “kernel process”.

7.4.13. And therefore, anything running within EL1 context can write into anything else within EL1, provided that it is mapped as writable.

7.4.14. Kernel memory is not swappable: the kernel has no underlying mechanism that would alter EL1 memory allocations, moving them to disk or dropping them (if they are read-only pages).

7.4.14.1. Memory compaction is a partial exception: it works on virtually linear memory allocations, changing the underlying mapping, so that it can carve out larger chunks of contiguous physical memory, which is particularly prized in special use cases (e.g. allocating a large buffer for either a peripheral device or a DMA controller that supports only direct physical memory accesses).

7.4.14.2. An underlying hypervisor could swap out either part or an entire VM, but that would not be controllable by the operating system (unless the hypervisor elected to let it have a say about it).

7.4.15. Almost all the memory the kernel sees as physical, which in reality corresponds to the IPA, is also mapped as VA, and it is referred to as "linear mapping", because it is mapped as contiguous in the VA space as it is in the IPA space.

7.4.16. Linux also supports HIGHMEM, which is physical memory that is not directly accessible by the kernel through the linear map. It is something that mostly impacts 32-bit systems, where physical memory can easily go beyond the size of the address space.

In fact, usually, not even all the physical lines of the address bus are wired, because they would not be necessary.

But certain choices of system design might make HIGHMEM necessary even in real life for ARM64, for example if it was decided to have a flatter page table with fewer levels.

To manage high mem, the kernel is forced to create temporary mapping every time it needs to access a page belonging to high mem, because there would not be any readily available corresponding address. And the mapping would then have to be torn down, once the access is concluded.

7.4.17. The free pages allocator picks free pages straight from the way they are represented in the linear mappings, operating on larger orders, and chopping and dicing large-order chunks to satisfy the requests received.

7.4.18. The slub allocator obtains memory from the get_free_pages one, and then uses various mechanisms to further dice the pages, providing sub-page granularity, if needed. Furthermore, it also supports additional optimizations in the reuse of previous allocations, like the ability to support locality in a NUMA system.

7.4.19. The virtual memory allocator is capable of providing large amounts of virtually contiguous memory, provided that there are sufficient (even non-contiguous) pages available. The allocator will create alternate contiguous mappings, to make them all appear as if they were contiguous.

7.4.20. Contiguous virtual memory allocations for both EL0 and EL1 are fundamentally identical, in the way they are performed, differing only in the chosen address range, which needs to be compatible for the receiving exception level.

EL0 mappings are also subject to active manipulation, due to on-demand paging and page eviction, driven by a need to provide addressable memory to other requestors.

More in details:

7.4.20.1. On-demand loading: not all the pages are loaded from disk right away, some are loaded only upon access attempt.

Till that moment, only the address range has been reserved, and even the backing physical page might be missing.

7.4.20.2. Active dropping of physical pages: under memory pressure from other entities (other processes, the kernel itself), the kernel might drop the backing physical page for constant content.

In case it is needed, it will be re-loaded. Heuristic decides which pages to target.

7.4.20.3. Active swapping out of non constant data: pages which cannot be dropped are written to disk, to a block device that the kernel uses for swapping out content that is present only in memory.

The reloading mechanism is similar to what described above, only in this case it happens from the swap device instead of from specific files in the file system.

Here too heuristics decide which pages will be the victim.

7.4.20.4. Page sharing: primarily during the forking of a process, most of the memory pages that do not require to be immediately copied and/or altered (like the stack) are dealt with in a lazy way, using COW: the page is shared, but mapped as read only by the process “borrowing it”, so that any attempt to alter such page will trigger an exception.

When the exception is handled, it creates a local, writable, replica of the page, and from this point onward the new process can write to it, because it’s not accessing a shared page anymore.

7.4.20.5. Page tables will evolve accordingly to the other phenomena described above, which means that they can also grow.

7.4.20.6. Out of memory killer: this is a sort of extreme case, where an entire process is terminated, to recover the physical pages that it had been allocated.

7.4.21. The activity of creating new mappings has an effect on the page tables themselves, which usually need to be expanded, to create new branches in the tree, to support the new nodes, unless branches (partially) already existed, due to previous allocations.

7.4.22. Pure kernel threads are executed in EL1 context, while the user processes are primarily executed in EL0 context. However, sometimes user processes need to transition to executing in kernel/EL1 mode, when the operations they require are limited to be executed in EL1 mode.

This is implemented through syscalls, which are a way for EL0 to invoke an handler in EL1.

The execution in EL0 relies on a call stack which is mapped in EL0, however a separate call stack is used when running in EL1 mode, due to the different page table in use.

The syscall will execute a specific service, as requested by EL0, and then return the execution to EL0 mode, once there is no further need of EL1 privileges.

8. Guidance on Safety Analysis and Mitigations

8.1. Code generation

The safety analysis must include the parameters used to generate additional code that might be introduced by the tool chain (which must be qualified too).

For example, enabling CFI mitigations has an effect on function calls that must neither be forgotten nor ignored.

It is not sufficient to use a tool chain that claims to be qualified; it is also necessary to confirm that it is being used within the limits of the qualification, in case only certain portions have been approved.

This is particularly important when the toolchain might produce binary executables which are not an exact product of the source code, because they might introduce a grey area of functionality that has not been validated, for example through source code analysers / linters.
In such a case, believing that one can rely on analysis of the source code alone is mostly wishful thinking.

8.2. Limitations of both the “Tested” and the “Proven in use” argumentations

This section is not about testing as it is typically intended during any development and integration process. It refers to approaches that might be proposed as a replacement for rigorous analysis and implementation of countermeasures / mitigations:

- “Proven in use”: the software has been already deployed over a large fleet of devices, for a very long time, with sufficiently similar use cases, both in terms of the actual hardware used and the way the software is exerted, to support the claim that the space of all plausible inputs has been exhausted, a very high number of times, with sufficient evidence to document that operations have happened within the expected operational parameters.

The documenting aspect is particularly important in building the argumentation.
Provided that all the changes are documented and proven to not introduce relevant alterations, the “proven in use” argumentation can provide exemption from more rigorous work.

- “Tested”: a testing campaign is created, to generate sufficient evidence that the software operates as desired, empirically.
  It can be seen as an alternative to “proven in use”, when lacking sufficient historical evidence.

The following considerations are related to these empirical argumentation and the associated limitations.

Even if “proven in use” and “tested” are different approaches, they are both exposed to variations in the conditions under which data is gathered, which tend to lead to comparable considerations.

8.2.1. Empirical data collected from extensive utilisation must prove to be relevant to the case at hand.
From this point of view, extensive utilisation in the field can be seen as equivalent to execution of a campaign of particularly well focused testing.

8.2.2. In order to leverage the results of empirical data, it must be proven that it is representative of the actual operating conditions that will be found in real life, during the utilisation of the product in the field.

8.2.3. In the case of historical data, it is necessary for the use case(s) that were leveraged to collect said data, to be also compatible with the intended new use, having similar fields of application and use cases.

8.2.4. To be credible, any empirical argumentation must survive this non-exhaustive list of invalidating objections:

8.2.4.1. Timing:

8.2.4.1.1. Exerting the code in the field, under a certain set of old conditions, can be significantly different from doing it still in the field, but under a new set of conditions. Or from doing it in an artificial setup.

This applies to both microprocessor-level operating conditions and macro level operating conditions, described as ODD.

8.2.4.1.2. Changes to the way the code is exerted can expose behaviour (and defects) that are significantly different from what emerged during either previous appraisal or testing campaigns.

Some causes:

8.2.4.1.2.1. Variation in the build parameters,
for example enabling/disabling debug options

8.2.4.1.2.2. Variations in HW builds, where different HW revisions have different timing

8.2.4.1.2.3. Variations in the code itself, for example a device driver changing from polling to interrupt-driven.
This applies both to changes that might take place over a long period of time, for the “proven in use” argumentation, and to changes that might take place during development, for a testing campaign that takes place in parallel to the development of the product.

8.2.4.1.2.4. Variations in the user-space payload, especially if it comes with RealTime constraints, since they can - and often will - preempt the operating systems in ways that are also affected by the payloads themselves.

8.2.4.1.2.5. Variations in the memory pressure, coming both from kernel and user space.
For example, a new version of a product might have additional applications, or the applications

might require more memory, or change the pattern at which they allocate memory (ex: going from individual allocations to bursts).

Or the Linux system might be running as one of the partitions managed by an hypervisor, and the other partitions would alter their behaviour, without the hypervisor enforcing any form of capping.

8.2.4.2. Memory layout:

Changes to the order that data and code appear in memory can expose different components to never-detected-before defects.

For example:

8.2.4.2.1. Changes to the layout used by the linker

8.2.4.2.2. Changes to the sizes of buffers from old to new builds

8.2.4.2.3. Changes to the set of device drivers in use, with consequent alteration of the memory occupation

8.2.4.2.4. Changes to the order of the initialization sequence of SW components, including device drivers.

For example:

8.2.4.2.4.1. Adding a new driver that registers an init function of whatever type (late, early, etc) will perturbate the sequence associated to the init functions of that specific init type (late, early, etc.)

8.2.4.2.4.2. Changing the init priority of a device driver (ex: from late to early init) will affect the timing of all the types involved, and possibly also of the types in between them.

Here is a very simple example.

A defect in the use of a statically allocated buffer causes an occasional overflow, which trashes the adjacent memory. But the memory happens to belong to a variable that is never used after the overflow happens.

Or maybe it is padding for optimised aligned access.
Then something changes; for example the variable being overwritten is turned into constant, and moved to a different segment.

Then, the same memory will be overwritten, but it might now be assigned to some data which is still referred to, after the trashing.

Even very effective testing or extensive use cannot rule out the presence of defects; it can only prove that there are no observable defects.

Perturbing the structure of the system voids the validation.

8.2.4.3. Equivalence of binary code:
changes to the build parameters and toolchains are likely to affect the actual executable code, with effects comparable to what discussed above. This refers to “live” binary, once loaded in memory and executed, rather than binary executables as they are in the file storage, where they might contain debug symbols and other data which does not affect what is loaded in ram and run.

8.2.4.4. Changes of toolchain are likely to invalidate the “proven in use” argumentation.

Evidence (like regression testing performed by the toolchain vendor) that the new toolchain produces similar binary artefacts can help in supporting the continued validity of the argumentation, however they should be supported also by additional testing of the code base under qualification.

8.2.4.5. **Generic effects of the observation on the system observed:** Unless the observation is proven to be completely non-invasive, it is expected that it will bring changes to the system, which will alter its behaviour.

Examples:

8.2.4.5.1. Need to allocate more memory for local storage of measurements performed during the testing.

8.2.4.5.2. Network bandwidth required or telemetry, where applicable.

8.2.4.5.3. Cpu used for performing the measurements (especially if polling-based) and transmitting them.

While it might be tempting to refer to unit-test, this should not be taken as a reference, because it is very unlikely to be able to reliably reproduce the full spectrum of events that can be encountered in a field-testing situation. And it would be anyways yet another equivalence that would have to be proven, while the whole point of the testing is to ensure that the right real-life conditions are met.

8.2.5. To make any reasonable claim of "empirically validated", regardless if it is "proven in use" or "tested", it is necessary to:

8.2.5.1. Identify all the system-level use cases

8.2.5.2. For each system-level use case to be ignored, provide evidence that the use case cannot cause interference, under any circumstances that are expected to be met during intended use. This requirement means that **it is not acceptable to omit a use case without having analysed it, and proven that it is acceptable to omit it**.

8.2.5.3. For each system-level use case to be considered, document if anything has been omitted from the testing plan in any capacity, and prove that they are acceptable from the perspective of safety analysis.

This refers to, for example, testing only for sub-ranges of certain parameters, or ruling out that one phenomenon might affect another, thus avoiding the test of combinations/permutations of parameters belonging to different subsystems (“equivalence classes”, in ISO26262 parlance), for the sake of reducing time/cost associated with testing.

8.2.5.4. For the remaining scenarios, prove that all the permutations of the relevant operating parameters have been exerted sufficiently (and justify what is deemed to be sufficient).
This is called, in the world of Functional Safety “Input triggers Space”.

Since Linux is a very complex system, it might not be feasible to achieve such a level of analysis that it would cover all the possible scenarios.
This would, obviously, invalidate any “proven in use” claim and it would require, instead, a qualification campaign.

8.2.5.5. Empirical evidence must be intended as proof of the inability to trigger errors, rather than as proof of their absence.

This is a fundamental concept, because it means that a change in the operating conditions - with consequent adjustment of the utilisation scenarios, can expose latent defects that have always been present.
It shows how “proven in use” argumentation really hinges on proving not only that previous use was not exposing problems, but also that the future use will happen in a very controlled environment, that is proven to be equivalent to the previous one. And, similarly, that “tested” was so exhaustive to cover all the plausible scenarios.

8.2.6. To make any reasonable claim of either “proven in use” or “tested”, it is necessary to:

8.2.6.1. Identify all the differences against the system configuration(s) that generated the empirical data.

8.2.6.2. For each of them, assess the impact on the argumentation.

8.2.6.3. For each difference that is negligible, provide evidence that indeed it can be ignored.

8.2.6.4. For each difference that is <u>not</u> negligible, provide analysis and, if needed, new, additional evidence that the future utilisation is safe.

8.3. Limitations of the self-protection argumentation

8.3.1. Since the self protection actuated by the kernel is primarily implemented through the page tables, which are exposed to interference, it is questionable how well the kernel can be expected to protect itself, without relying on an external component with a sufficiently high safety integrity level, and typically shielded from interference that might happen either within the kernel or as manifestation of them.

8.3.2. Similarly, in case specific features/mechanisms (for example redundancy of certain data parameters) are introduced and used as part of the safety argumentation, they first need to be demonstrated to have the required safety integrity level.

The challenge with these additional mechanisms is to prove that they are sufficiently simple to not have dependency on the very same context they are trying to protect (similar to the concept of Technical Independence).

It needs also to be clarified what it is really meant with “protection”, because, strictly speaking, protection requires either the ability to dodge interference or the ability to recover from it both sufficiently and timely.

For example, using canary values on the call stack can probably be sufficiently good for detection of certain types of interference, like stack smashing, however, once an interference is detected, it is unlikely that it can be corrected through the use of the canary values.

8.3.3. Even by allowing the definition of protections to be broadened to include what is in fact detection, it needs to be rooted into a simple, easy to both prove and verify, mechanism.

For example, a self-test diagnostic capable of detecting spatial interference in a certain component, by running periodically and/or in event-driven mode, is still exposed to interference and it might not be easy to prove that such interference can at the very least be detected.

However, by pairing it with a simpler mechanism, like a watchdog, then it can become easier to make claims about detecting the interference, because the detector can be designed in a way to fail to ping the watchdog, in case of its own corruption.

8.4. Mitigation Strategies

It is a fact that there is almost no defects-free code, and the Linux kernel is certainly not an exception, therefore any analysis/mitigation that relies on proving that the Linux kernel is safe, should also come with a self evaluation of its own vulnerability to unforeseen failures, based on how exposed it might be to some assumptions turning out to be false, including assumptions about testing coverage, for example.

In other words, there are 2 paths, when assessing the effectiveness of countermeasures for interference on a target:

8.4.1. The typical path: identify the type(s) of interference and introduce countermeasures which are independent from the possible source.

This approach ensures that, even in the presence of an incomplete analysis of the sources of interference, the countermeasures will still be effective, provided that all the types of interference have adequate countermeasures.

8.4.2. The alternative path: identify all the sources of interference, for all the types of interference that are meaningful, and introduce a countermeasure for each of these sources.

This approach is theoretically equivalent to the previous one, but it relies on having an absolutely exhaustive list of sources of interference, each of them paired with an acceptable countermeasure, or proof that a countermeasure is not necessary.

The burden of proof about completeness and effectiveness is on whoever might choose this path.

8.5. Statistical considerations

8.5.1. In the light of previous observations, about hard barriers vs deductive argumentation and defect density, one should also consider the chances that a certain component might generate interference (which depend also on its size and complexity) vs the frequency said component is exerted (assuming a periodic or quasi-periodic invocation).

8.5.2. This leads to a qualitative evaluation of which components are more likely to cause interference and therefore deserve additional analysis, from multiple perspectives: complexity, frequency of execution, types of operations performed, detectability of interference it might generate, delays in the detection, etc.

This is not a small task, but it is critical in understanding the price to pay for utilising the inductive method, and **failing to do so will introduce the risk of having a system that both lacks physical barriers and has not been properly analysed.**

There is also a feasibility problem: linux is ever evolving and there is no official bug tracking system. At most some 3rd party might keep track of defects related to security and vulnerability, however that is far from being the full picture.

One might be tempted to use mathematical models that attempt to model the software in terms of bug density, severity, etc. and use said model to predict the probability of interference from a certain component.

While this approach might work with components that are fully owned by a single organisation / entity and tracked appropriately to support said approach, it is very easy to see how the concept would fall apart, when applied to Linux.

One might be tempted to fork Linux, to make it fall back in the fully-owned software scenario, however this approach would be unfeasible, because it would rapidly become unmanageable, with need of backporting an increasingly number of patches such as security fixes, and the local fork diverging from upstream, effectively losing most of the advantages brought by the use of Linux.

8.5.3. Such consideration points to the fact that certain mechanisms which can be seen as supportive of integrity (and thus safety) in user-space are a double-edged sword, since they can get invoked quite frequently and can be fairly large and complex, in terms of size of code. In a way that cannot be reliably handled.

Some examples:

8.5.3.1. cpu/memory/network cgroups

8.5.3.2. security frameworks, like the Linux Security Module and SELinux.

They are particularly relevant because their behaviour is strongly affected by entities considered hostile or at the very least not friendly, like non-safety relevant processes attempting to generate large amounts of cpu load / memory pressure / network traffic or file accesses.

8.5.4. Even system activity such as unrestrained patterns of memory allocation / release cycles is more likely to trigger problems.

These allocations can happen at high frequency, while the underlying system changes its state rapidly, producing a large variety of permutations in the code paths being followed for achieving the very same results; for example the previous allocation patterns, triggered also by unrelated processes, can lead to different ways of procuring the pages required.

In some cases it might be necessary to stall and invoke the buddy allocator, in others it might be possible to just pluck the memory from the queue implemented by the slub allocator for the local core.

Taking as example a non-safety relevant application which generates many events such as allocating/releasing memory / opening and closing sockets and files / spawning and starting and stopping threads, it will trigger associated high frequency execution of code paths within SELinux and cgroups, to validate the legitimacy of said actions.

This is perfectly fine from the perspective of containing user-space, however, since neither SELinux nor cgroups are by default safety qualified, it also means that within the kernel there will be a high frequency execution of a large amount of code which can cause either direct or indirect interference.

And such interference is not always detectable, depending on which component it might affect.

9. Sources of Interference

It is useful to model the most probable causes for spatial interference, even if not exhaustively.

Note: for the purpose of this document, the hardware is considered qualified up to the highest safety integrity level required by any use case. The qualification is to be intended at unit level, meaning that no hardware component will exhibit aberrant behaviour, when exposed to the supported stimuli.

In other cases, should the previous assumption not be true, each HW component must be individually checked for safety qualification and those not matching safety requirements must be included in the list of sources of interference.

9.1. DMA-capable entities

Those components which sidestep the MMU-enforced memory protection, by generating write operations on the memory bus through other bus-master devices than the MMU.

The catch here is that, even if such access is performed by a separate DMA controller, the programming and triggering of the write operation is performed by a device driver that is exposed to interference and can therefore cascade it (through either mis-configuration or mis-operation of the DMA controller).

Causes: The interference is possible because it originates from a component that is architecturally capable of generating it, but said component was not assigned sufficiently high safety requirements, that would account for such possibility. Nor a mechanism is in place to manage the interference.

Effects: Unpredictable corruption of the state of safety-relevant context. The extent of the corruption is highly dependent on local conditions that are not necessarily repeatable.

Detectability/Mitigation: This sort of interference is practically impossible to detect, without HW means, like an IO/MMU, a memory firewall, or some form of redundancy.

Lacking any of that, it is only possible to try to detect side-effects, without guarantees on the timing.

9.2. Components present in EL1, with lower safety integrity level

Any code running in EL1 has unbridled write access to any non-write-protected memory pages in EL1.

Any SW component which runs in EL1 with any safety integrity level, is to be considered as a source of interference to any other EL1 component with higher safety integrity level.

Typically, this would include the entirety of the Linux code base, as it is obtained from upstream, unless it has been adequately reviewed and fixed, or it is used in a way that mitigates its intrinsic low safety integrity level.

Causes: The lower safety code must be assumed to contain defects that will generate interference which cannot be claimed to be mitigated purely by the rigorous development process.

They can be either low level defects, or conceptual defects.
Typical defects:

- Functional defects

  Can be anything, literally, however, even assuming a reasonable amount of unit test, extensive integration testing is what can make the difference between qualification at different safety integrity levels.

- Races

  Possibly a specialised case of the previous point, it is a type of fault which can emerge from missing to consider all the possible execution paths, especially when factoring-in unrelated asynchronous and synchronous events, caused by memory pressure, I/O, task migration, underlying presence of other partitions managed by an hypervisor, etc.

  All of this can contribute to diverging from the expected (and intended) execution flow, if concurrence was not taken into account properly.

- Use-after-free

  This is a source of interference that can be hard to detect in a subset of cases.

Primarily, it affects memory pages which are accessed through the linear mapping, provided by kmalloc/get_free_pages, then get released and eventually re-allocated, possibly also through different allocation mechanisms.

And then the previous reference is used again, after being released.

A write operation will cause an interference that can affect any other component that received the memory allocation, in an unpredictable way.

Effects: *Any data within the EL1 context is automatically downgraded to the lowest safety integrity level present within EL1. Any memory used for EL0 is mapped in EL1, so also anything user-space is equally exposed, both code and data, constant or not.*

Detectability: Here HW-based detection through an IO/MMU or a memory firewall is not an option, because the source of interference is a core itself, going through the MMU.

Mitigation: Detection needs to rely on indirect effects that are not guaranteed to be noticed. A micro kernel would deal with this problem through MMU-enforced isolation, but Linux is a monolithic kernel and cannot do that.

9.3. Partitioning of hardware components between different safety integrity levels: constraints and limitations

This type of interference could be seen as a design flaw, but in practice one might not be able to implement as much hardware partitioning as the ideal case would require.

The typical example is a shared hardware resource, for example a bus controller like I2C or SPI, where multiple external peripherals might be connected, and only some of them would belong to a safety scenario at a certain safety integrity level, while others would be at a lower safety integrity level.

Another common example could be an interrupt controller with shared lines.

Causes: Even assuming spatial FFI between the lower safety kernel components and the driver for the shared resource, the lower safety components might mis-configure the shared resource (e.g. by interfering with a channel assigned to a different peripheral) or hog it to the point of affecting the operations of the higher safety ones .

Effects: Higher safety components depending on the shared resource might be unable to use it as intended, being starved, or their use might be disrupted in other ways, either corrupting the state of the shared device or of other components that are proxied by the shared device.

Detectability/Mitigation: As long as it is possible to set expectations about the temporal evolution of the systems whose safety is being analysed, it might be possible to rely on a timeout-based detection system, however, purely asynchronous events, like a safety-relevant peripheral attempting to request servicing through an interrupt, could go completely unnoticed.

9.4. System libraries

The Linux kernel provides a large number of libraries implementing basic functions, both specific to an operating system and others that replace what would be part of the compiler libraries. In Linux the compiler is used in free-standing mode and therefore even basic I/O like the support for printing to console is not available by default.

These libraries can be invoked from both highest integrity and lower integrity contexts, which means that they need to be validated and managed according to the integrity level depending on them.

Examples: list management, locking.

Causes: The library is used against safety-relevant data, and it needs to be qualified for doing so.

Effects: Lack of adequate safety integrity level can cause undetectable interference in the data that the library operates on. Or, even worse, on data which happens to be writable by the library, even if it is unrelated.

Detectability/Mitigation: This sort of interference is practically impossible to detect, and the only option left is to qualify the libraries to the required safety integrity level.

10. Exposure to Interference

10.1. Criteria for evaluating interference

The Linux kernel is a very complex SW component, which has tight integration with the HW components of the system it happens to run on.

Even limiting the analysis to spatial interference, there is a broad range of elements which can suffer from it and cause various degrees of failure in the system.

Depending on the use cases/expectations, certain failures might be relevant from a safety point of view, or not. It is therefore impossible to assert in absolute terms what constitutes a safety-relevant interference, without referring to a specific system.

However, it is possible to produce a set of high level considerations, to be used when analysing a specific system.

Even the outcome of the evaluation is subordinate to the requirements set ad-hoc for a specific application: the very same interference can be seen as acceptable, under certain requirements, and unacceptable under stricter requirements.

10.2. Fundamental Considerations

Nevertheless, it is possible to conjure some considerations that will apply to any analysis of a system based on Linux, even if they will lead to conclusions which are specific to certain use-cases.

10.2.1. No matter how a system might be partitioned for facilitating its analysis, the only true boundaries to interference are those enforced by either the MMU or some other, equivalent, HW component (e.g. a HW Memory Firewall), defining a memory context.

Other methods might give the illusion of providing partitioning, but it rapidly becomes even harder to prove their correctness.

Formal verification might be tempting, but it would not be practical, when applied to a complex Os that was not designed from the ground up for it, not to mention the fact that - lacking any control whatsoever on the OS release process, and the content of said releases, it becomes hopelessly unpredictable to anticipate the amount of work required for refreshing the verification on new OS releases.

Chosen a target for interference, there are several ways the interference might happen:

10.2.1.1. Self interference, where the component does not behave according to expectations and compromises its own safety, due to defects in one or more of design, implementation, integration.

10.2.1.2. Interference from other components which are indeed expected to interact with it, but for some reason are not acting according to expectations.

10.2.1.3. Interference from other components that are not expected to interact at all with it, yet they do it.

The latter is particularly troublesome, because, lacking hard boundaries, anything can interfere with anything else.

As long as the target for interference is exposed to other components which have the same or higher safety integrity level, the exposure is acceptable, even if not desirable.

However it is normally the case that different components have different qualifications.

10.2.2. Attempting to rely on induction for introducing a “soft” partitioning doesn’t work too well either, because it would require:

10.2.2.1. Hard evidence that the reasoning is completely exhaustive of all possible interactions (omitting something due to ignorance is not acceptable, because it doesn’t prove that it was a valid simplification).

10.2.2.2. Re-assessment of the induction every time the system is updated with new SW that can potentially interfere, till it is proven otherwise.

This approach either rapidly becomes unsustainable, or it imposes very harsh limitations on the frequency for introducing updates.

Frequency that might not be acceptable, if it doesn't meet the minimum requirements of delivering updates to a product.

Security updates are a perfect example of a situation where, even in presence of a release plan, critical vulnerability can require out-of-cycle releases. And the security fixes have the potential of voiding precedent safety assessments.

10.2.3. Risking to state the obvious, there is one exception to having HW-enforced partitioning: **time-enforced partitioning**.

If it can be proven that a certain component will **cease operations past a well established watershed moment**, then it is possible to consider that time boundary as an effective isolation.

However, it is still necessary to prove that, after the aforementioned watershed, no interference has been found, which makes this argumentation far less trivial to implement than it might appear.

The Linux kernel init phase is a major opportunity for this sort of argumentation, because Linux supports even wiping and reusing the memory initially assigned to "__init" data and code segments.

Other operations might equally benefit from a similar argumentation, provided that it can be proven that:

- They are employed only during init.
- Their effect can be verified right after init has completed.

10.2.4. Every subsystem relies on memory, allocated in various ways, to manage its internal states.

The internal states of any subsystem are exposed to potential interference from any other code that happens to be executed within the same memory map .

10.2.5. When considering means for mitigations to interference, it must be considered how they operate:

10.2.5.1. *"Passive" means:* they are based on redundancy or similar principles.

They can be equally exposed to interference, but this is acceptable, as long as it is proven that they are sufficiently resilient (ex: use of self-correcting encoding, like CRCs).

These means are less invasive, because they only need to be hooked into the execution flow, but they do not alter it substantially.

When subject to interference, they - most likely - do not affect the component that they are trying to protect.

This approach doesn't introduce correlation between the target for the mitigation and the mitigation itself, therefore it can be considered sufficiently trustworthy to detect a single interference.

However, it is expensive to handle any form of redundancy on a large scale.

For this approach, it must be proven that, given a certain target for instrumentation, the specific implementation of the redundancy is sufficient.

10.2.5.2. *"Active" means:* they introduce restrictions or anyways alter the way that the system behaves.

For example, they can introduce runtime restrictions, dynamically limiting the space of possible actions to those which are allowed, by design, at any given time.

In practice, anything which is not explicitly allowed becomes forbidden and can raise some form of exception, to notify that an illegal operation was attempted and possibly performed.

This approach aims at restricting what can happen silently when a component goes out of its expected operational zone, but without compromising what happens within the legitimate operating parameters values.

Therefore, because of the tighter integration, "active" means have a potential for becoming a source of interference themselves, causing cascaded interference, if not adequately treated.

It must be proven that they are themselves protected from interference (technical independence), or else the argument is moot.
The advantage they have over the passive type is that they can scale up better, because the underlying mechanism doesn't have to be applied to each individual parameter to be protected, but

rather rely, for example, on establishing memory regions and boundaries.

10.2.6. Depending on the subsystem analysed, the very same interference can require different types of mitigations.

For example, assuming that it's acceptable to perform polling and validation on a system where the amount of data to monitor is small, the same approach might not be suitable for another system where the amount of critical data to monitor would be so large that the polling and validation activity would generate an excessive overhead, where the meaning of "excessive" is subject to the specific system and requirements..

Similarly, on another system, there might be a relatively small amount of data to poll and monitor, but it might be changing so rapidly that the associated polling would, again, cause excessive overhead, because it would have to happen with comparable rapidity.

10.2.7. The requirements will affect as well what sort of mitigation might be necessary.

For example, minimal or no mitigations might be required, if the only goal is to detect interference in selected subsystems and prevent effects from spreading in an uncontrolled fashion.

However, if it is required to ensure a set level of availability, pure detection might not be an option, and prevention would become necessary, with all the associated implications.

10.2.8. When dealing with interference, it boils down to two options:
(**Note:** FuSA and FMEA jargon assign very specific meanings to the words below, when referring to failure modes. However, in this document, they are used with regard to the interference that might introduce a failure mode, and therefore these words are to be intended exclusively with their plain meaning from the English vocabulary. See also their definitions in the section Terms and Abbreviations.)

10.2.8.1. **Prevention (of an interference)**
The act of denying a potential interference the possibility of actually manifesting itself.

Prevention is harder to implement, but it ensures that the relevant context will not be compromised, and thus doesn't come with a timing constraint, **enabling higher levels of availability**.

10.2.8.2. **Detection (of an interference)**
The act of identifying an interference that has already happened, either directly or indirectly.

Detection, on the other hand, doesn't require the creation of any barrier, provided that there are mechanisms in place that can sense that an interference has happened.

However, it does put a strain on the sensing, because it needs to happen sufficiently fast to satisfy the timing specified by the related safety requirements.

10.2.9. Obviously, there would be a third option: *removing any source of interference from components at lower safety integrity levels*.

In this case, <u>everything</u> which is within a memory boundary needs to be qualified at sufficient safety integrity level, to avoid also indirect interference of secondary order or higher, through cascading.

<u>This level of blanketing, very broad qualification is very unlikely to achieve and even more fleeting, given how it would rely on every single component to remain qualified.</u>

10.2.10. When it comes to interference, as long as detection is a viable option, it's not so much about how catastrophic the interference might be, but how likely it is that it might go undetected and take the system out of its operational parameters in a way that is not mitigated (or even considered), and what the effects would be.

10.3. Components susceptible to spatial interference

The following is a minimal, non-exhaustive, set that should be considered in a safety checklist.

10.3.1. **ASIL components at system initialization**

While freedom from interference is meant to be a “live”, persistent property, with any violation detected timely, some of the mechanisms employed rely on detecting unexpected evolutions in their state.

But first it is necessary to assess whether the initial state is safe, or if it has already been compromised during its initialisation phase.

Exposure: Such an assessment could be avoided, if it was possible to prove that all the mechanisms employed for the initialisation are safe, and that unrelated unsafe components cannot interfere.

But in case that was not possible, then there is a risk of interference, for any component of a certain level of safety, coming from lower safety integrity level ones.

Effects: The effects are specific to each individual component, however a lack of confidence that the system is within safe operating margins even before it begins active operations would void any further assumption about its continued safety.

Detectability: This is the base-line level of detection, that a component with an allocation of safety requirements is still safe according to its intended safety integrity level, at the very beginning of operations. It should always be possible to do so.

Were it not possible, then it becomes questionable how it can be proven, later on, that it is still operating according to its safety requirements.

10.3.2. EL1 system registers

Not all system registers are equal, some have special purposes that can deeply affect the flow of execution.

Exposure: Within a certain exception level, most registers are writable directly, without any form of protection.

Depending on the register type, they might be more or less exposed to interference:

- Registers encoded within instructions are less likely to be accidentally accessed

- Registers that are memory mapped in a parametric way are more exposed to risks of interference (this is an ARM-specific problem: as a counter example, x86 uses special instructions for these sorts of registers and therefore they do not belong to the same space as regular memory).

Effects: This is not possible to be generalised, because each type of register can lead to different effects, if exposed to interference.

Obviously, those related to control flow have a bigger potential to lead to unsafe behaviour.

Detectability: The only way to perform direct verification would be to have some sort of twin system evolving in parallel. In certain SoCs there is indeed a lock-step mode where a shadow twin is connected, and it mirrors every action, to validate it, assuming that the interference is not happening systematically on both.

Up to a certain point, the shadow twin could be emulated with a hypervisor and a "shadow VM", but that would create a large overhead, and also pose the questions of which synch points to use and how often to perform the comparison of the registers.

Entry / exit points of function calls might be good candidates, but since some of the Linux operational parameters are intentionally randomised, it should be proven - for each individual use - that it's either reproduced or irrelevant for safety purposes.

A simpler approach would instead define checkpoints based on design and expected behaviour, and monitor that such checkpoints are reached timely.

But the caveat is in ensuring that checkpoints are defined correctly and that they are sufficient. Which then needs to be proven.

Prevention is an easier argument to support, if it can be substantiated.

10.3.3. **EL1 Safety-relevant Call Stacks** used for:

10.3.3.1. Safety-relevant kernel threads

10.3.3.2. Safety-relevant processes

10.3.3.3. Exception handlers

Same stack is used for both safety-relevant and non safety-relevant exceptions.

Exposure: They are mapped in regular EL1 memory and they are writable

Effects: The registers saved on the call stack can be corrupted in ways that are not easy to detect, and cause an uncontrolled drift in the operating parameters outside of their admissible values.

Detectability: Direct detection is possible only through introduction of mitigations, like canary values, and even that is not 100% reliable. Indirect detection is even more difficult to implement, because of the wide range of possible effects.

10.3.4. EL0 Safety-relevant Processes Memory

These processes represent the typical high-level use-cases encountered in a system with safety requirements.

Their overall safety depends on multiple factors, which can be monitored with different levels of success, however this specific point refers to their freedom from spatial interference.

Exposure:

- **The physical memory (IPAs, if considering also EL2) mapped in EL0 for processes to use is also part of the linear mapping in EL1 and fully addressable and writable.**

- The memory used for the page table of the EL0 process is equally exposed to interference originating within EL1

Effects: Depending on the chance, either data, code, or both, including stacks, belonging to the process can be corrupted in any unpredictable way.

This path of interference can also affect constant data and code belonging to the process, because their write protection exists only in the EL0 mapping, not in the EL1 mapping, where they are completely vulnerable.

One can assume that alterations to the page tables would be less likely to pass unnoticed,since it’s extremely unlikely that any corruption would point to another memory page with content that would not cause the MMU to throw an exception about a malformed page table entry.

But that would be an educated guess, rather than an objective fact.

Detectability: Here too, the interference can assume a very wide range of effects, only the most blatant of them easily detectable.

For example, corruption of a table of constant data would require a periodic verification through either checksums or similar measures.

Corruption of code would be far more easily detectable.

Some types of interference can be spotted far more easily than others.

10.3.5. EL0 Shared Memory / Pipes for Safety-relevant Processes

Conceptually similar to the previous point, however possibly subject to a different mitigation strategy, it is listed separately to ensure visibility.

Exposure: Like any other memory mapped in EL0, it is exposed to interference from EL1 through the linear map.

Effects: Data exchange between user processes can be corrupted.

Detectability: Processes can implement some form of checksumming for detecting corruption. This can become burdensome for them.

10.3.6. EL1 Memory Managers - Buddy Allocator - get_free_pages()

The Buddy allocator is at the heart of almost any runtime allocation performed in a Linux system, including both the creation and whole lifetime support of memory allocations for user space processes.

Exposure: The allocator relies on its own internal data structures, which are supportive of keeping track of which memory pages are available, which are in use and the exact way they have been partitioned.

These data structures have their own life cycle and are sourced from the very same physical pages that are used also for satisfying the allocation requests from various memory users.

This includes both memory used for safety-relevant use cases and for non safety-relevant use cases.

The data is fully writable by any code running in EL1

Effects: Any corruption in this set of internal data structures can cause cascaded interference on safety-related data, for example by handing over, as usable, the memory pages already reserved for a safety-relevant process.

This would cause the pages to be overwritten, with a wide spectrum of possible outcomes.

Detectability: Anyone's guess. Depends on the specific case, but it must be assumed that it would not be detectable in more cases than it would be acceptable.

Or else, evidence must be produced, to claim otherwise.

Certainly, the interference cannot be detected directly, but only through 2nd or 3rd order side effects that might not always be easily detected.

One should either assume the worst, and detect/prevent it, or provide evidence about why the worst would be unlikely to happen, in case the mitigations are not extensive.

Considerations: Unless proven that the memory manager is:

- Of adequate safety integrity level (the highest required between EL0 / EL1)

- Free From Interference for what concerns its own metadata

It is not sufficient to prove at runtime that safety-relevant allocations have happened in a successful way (for example doing them at init and verifying post-init that they were correct), because:

- If the memory manager is QM, it can still cause interference to the **existing** allocations in use by components with safety requirements, for example by lending a memory page that is already and still in use by a safe component.

- if the metadata of the memory manager is still exposed to kernel QM components, it can still be corrupted and lead to the same type of problems mentioned in the previous point.

10.3.7. EL1 Memory Managers - Slub allocator - kmalloc()

The slub allocator is the go-to allocator for typical runtime needs of allocating memory at runtime, both because it is more efficient, especially when dealing with per_cpu allocations, and because it is capable of dishing out sub-page allocations. It is widely used within EL1, but it doesn't have direct effects on EL0 processes.

It specialises in optimising finer grained allocations than the buddy allocator, including their lifecycle and caching.

Exposure: The slub allocator has a compounded exposure, since it uses the buddy allocator as backend, but it also has its own set of internal data structures, which is writable by any code with EL1 write capability.

Effects: The effects are comparable to those caused by a corruption of the buddy allocator, minus the effect on EL0 processes.

Detectability: Similar to the detectability of the buddy allocator.

Considerations: Similar to those made for the buddy allocator.

10.3.8. EL1 Memory Managers - VMALLOC - vmalloc()

Vmalloc too has its own set of internal data structures (for keeping track of a large variety of runtime parameters), and it is involved with both EL1 allocations and with the creation and backing of EL0 processes.

Compared to the previous allocators, vmalloc adds one more layer of complexity due to the fact that it also needs to keep track of address ranges and physical backing of reserved addresses.

Exposure: The exposure is even more compounded by the fact that vmalloc relies on both the buddy allocator and the slub allocator for its internal data structures.

Furthermore it relies even on itself for allocating housekeeping memory of a certain type, in case the amount of memory required is so large that it cannot be satisfied by kmalloc.

Effects: Very unpredictable, but they can certainly lead to the corruption of safety-relevant context.

Detectability: Similar to the detectability of the buddy allocator.

Considerations: Similar to those made for the buddy allocator.

10.3.9. EL1 Memory Managers - others

While they might not be as broadly known and used as the ones previously listed, the Linux kernel does provide a host of other allocators which are meant to support the management of special memory.

Examples: genalloc, memblock, cma_alloc.

Exposure: also these allocators rely on metadata they need for housekeeping, typically obtained from kmalloc/vmalloc, therefore they are equally exposed to interference coming from anything else with lower safety integrity level.

Effects: In case the device drivers using these allocators must meet safety requirements, the effects of interference can be even worse than on other allocators, because in some cases these allocators are used for keeping track of free/used blocks in storage devices, like I2C/SPI permanent memories.

A corruption of the bitmap tracking free/used blocks can easily lead to permanent obliteration of some/all of the data stored.

Detectability: This is a sort of problem that might not be trivial to detect, without ad-hoc mitigations.

10.3.10. EL1 PageTables Integrity

The EL1 page tables are relevant not only to safety contexts; far from it.

However, their integrity is a necessary condition for the integrity of the safety contexts.

At the very least, one must consider the portion of the page tables which supports safety-relevant mappings.

Indirectly, though, also the rest of the mappings is relevant, to ensure that a safety-relevant page is not mapped also elsewhere.

Exposure: The memory pages comprising the page tables are writable from within EL1 context.

Effects: In the best case, corruption won't cause noticeable problems, however it can cause anything from crashes to subtle corruptions, depending on what might cause the interference.

In the next-best case, the effects will be so massive that they can be detected immediately.

Detectability: Also in this case, direct detection is unlikely, and indirect detection is based on the ability of modelling the most probable side effects.

10.3.11. EL1 PageTables Consistency - Double Mapping Prevention

Albeit it could be seen as a sub-case of integrity of the page table, consistency should be considered separately, because it is not a problem of the page tables themselves, but of what information is stored in them, in the first place.

It consists of the same page being mapped at two or more different addresses, and for different purposes. It can be caused either by a defect in the management of free pages, similar to what described earlier, or by the corruption of the metadata that a page-based memory manager maintains for housekeeping purposes.

Exposure: A memory page used for safe content - even if mapped as read-only at a certain address - can be mapped as writable (and written!) at another address, completely unrelated.

Effects: In the best case, corruption won't cause noticeable problems, however it can cause anything from crashes to subtle corruptions, depending on what might cause the interference.

In the next-best case, the effects will be so massive that they can be detected immediately.

Detectability: Also in this case, direct detection is unlikely, and indirect detection is based on the ability of modelling the most probable side effects.

10.3.12. EL1 Task Execution

This represents a host of features that are in charge of juggling tasks; for example:

- Management of related data structures (tasks and cred structures, stacks, etc).
- Management of threads; creation, destruction.
- Work queues.
- Timers / scheduling.

Exposure: Any of the features mentioned can be affected by interference, in some form.

Effects: Not all the features are equally affected, from a safety perspective.
For example, the credentials structure is less likely to cause direct problems to safety.

Detectability: Provided that the timing constraints for periodic events is known, external monitors can be deployed, to confirm that the task is being executed accordingly to the expected timing constraints.

11. License: CC BY-SA 4.0