A maintainable and scalable Kernel qualification approach for Automotive

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Gabriele Paoloni (Intel)
Daniel Bristot de Oliveira (Red Hat)

Disclaimer

Note that this is currently WIP.

No formal results are binding on behalf of ELISA/Linux foundation, nor we make any safety claims based on this preliminary report..

ENABLING LINUX IN SAFETY APPLICATIONS

Agenda

- In-scope and out-of-scope of the presentation
- Possible Functional Safety qualification approaches for Linux
- The Hybrid qualification approach
- Hybrid approach applied: ioctl() example
- Integration Tests through Runtime Verification (RV) Monitors
- Next steps
- Q&A

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

In/out of the scope of this presentation

In Scope:

- Proposal and high level description of a functional safety (FuSa) qualification flow of Kernel code allocated with safety requirements to meet a certain ASIL target according to ISO26262

Out of Scope:

- FuSa qualification of the HW
- FuSa qualification according to safety standards beyond ISO26262
- FFI claim between coexisting Kernel partitions allocated with different ASIL levels

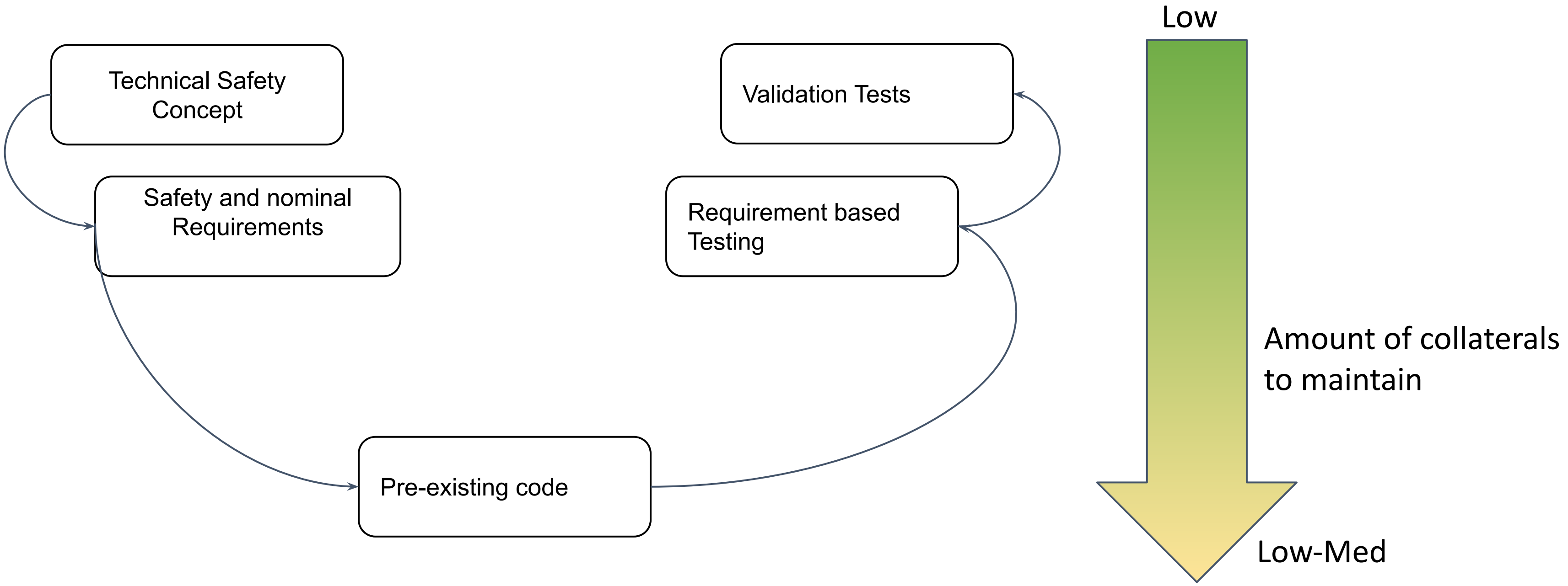
ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

ISO26262 Introduction

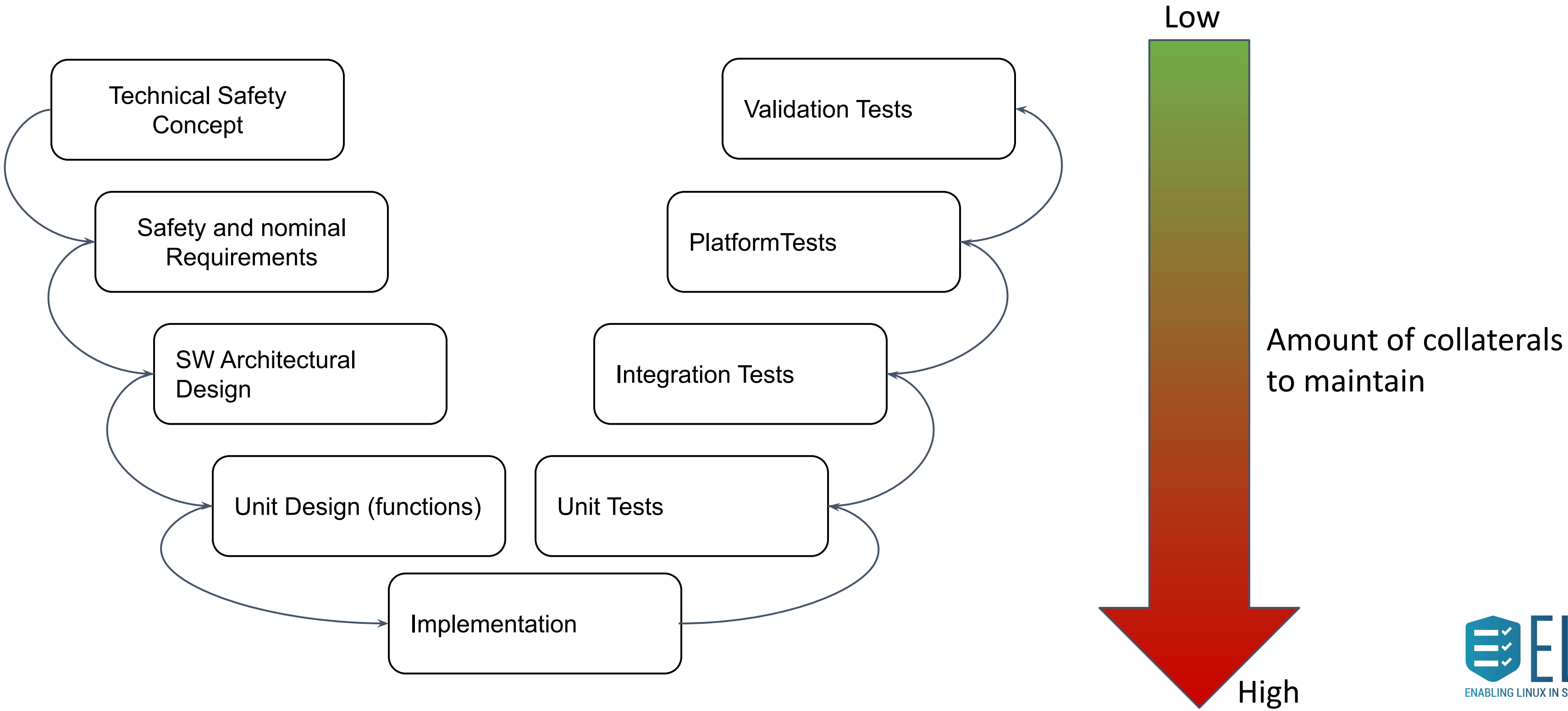
- ISO26262 provides **three options** to qualify pre-existing SW components
  - Part 8.12:
    - It is a **black box approach**
    - Based on verifying the SW component to meet the allocated top level nominal and safety requirements.
    - Although there are not explicit statements about complexity, it is **commonly accepted only for simple SW** components whose behavior can be comprehensively described by the top level specifications;
  - Part 6:
    - It is a **modular and hierarchical white box approach**:
    - It is suitable to develop and assess SW components of **any complexity.**
  - Part 8.14
    - It is a qualification based on the proven in use of the SW component
    - Enough statistical data about failures in time of the SW component must be available
    - The component configuration and its usage conditions must be identical or have a high degree of commonality with those used to collect the statistical failure data
    - The approach is **harder to scale to different HW, Configurations, Use Cases**
  - Part 10.9
    - It is a qualification or development approach based on assumptions (assumed safety, nominal requirements and conditions of use). Practically speaking it redirects to any acceptable development or qualification approach already defined in other parts of the ISO26262 standard
    - **It doesn't provide an additional approach in practice**

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Part 8 Standard Approach



Part 6 Standard Approach



ENABLING LINUX IN SAFETY APPLICATIONS

ISO26262's possible approaches for Linux

- Given the current state of ISO26262:
  - Linux is too complex to be qualified by ISO26262 **Part 8.12** alone
  - The SEooC approach only covers the requirements definition, the rest of the component is still to be qualified according to a possible ISO26262 method.
  - It could be assessed according to **Part 6;** however, the application of the **ISO26262 Part 6** in Linux is challenging, especially with respect to the amount of work required to meet the clauses of unit design, implementation and testing
  - It could be qualified according to part 8.14, but only if statistical data is available for the specific HW, Configuration and Usage conditions of the target system where Linux is deployed.

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

ISO26262-6 pain points in Linux

- Specific notations for the unit design:
  - Informal notation for ASIL up to B
  - Semi-formal or formal notation for ASIL C and beyond
- Specific design/implementation principles for SW units
  - One entry/exit point in each function
  - No dynamic objects/variables
  - No multiple use of variable names
  - No implicit type conversion
  - No unconditional jumps
  - (and so on…)
- Unit tests verification:
  - 100% code coverage and requirements coverage of SW units

Linux accounts for:

- **> 80 thousands functions**
- **> 15 million lines of code**

The effort to write and maintain the documentation, tests and infrastructure is not viable.

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

ISO26262 Dilemma:

Linux is too complex for **Part 8.12**

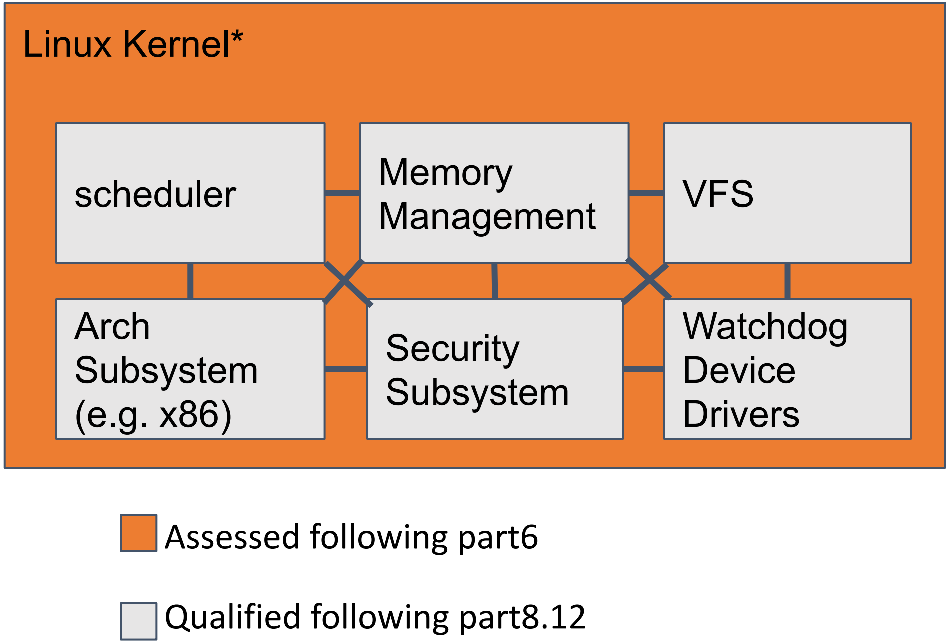
×

Part 6 is too complex for Linux

The hybrid approach

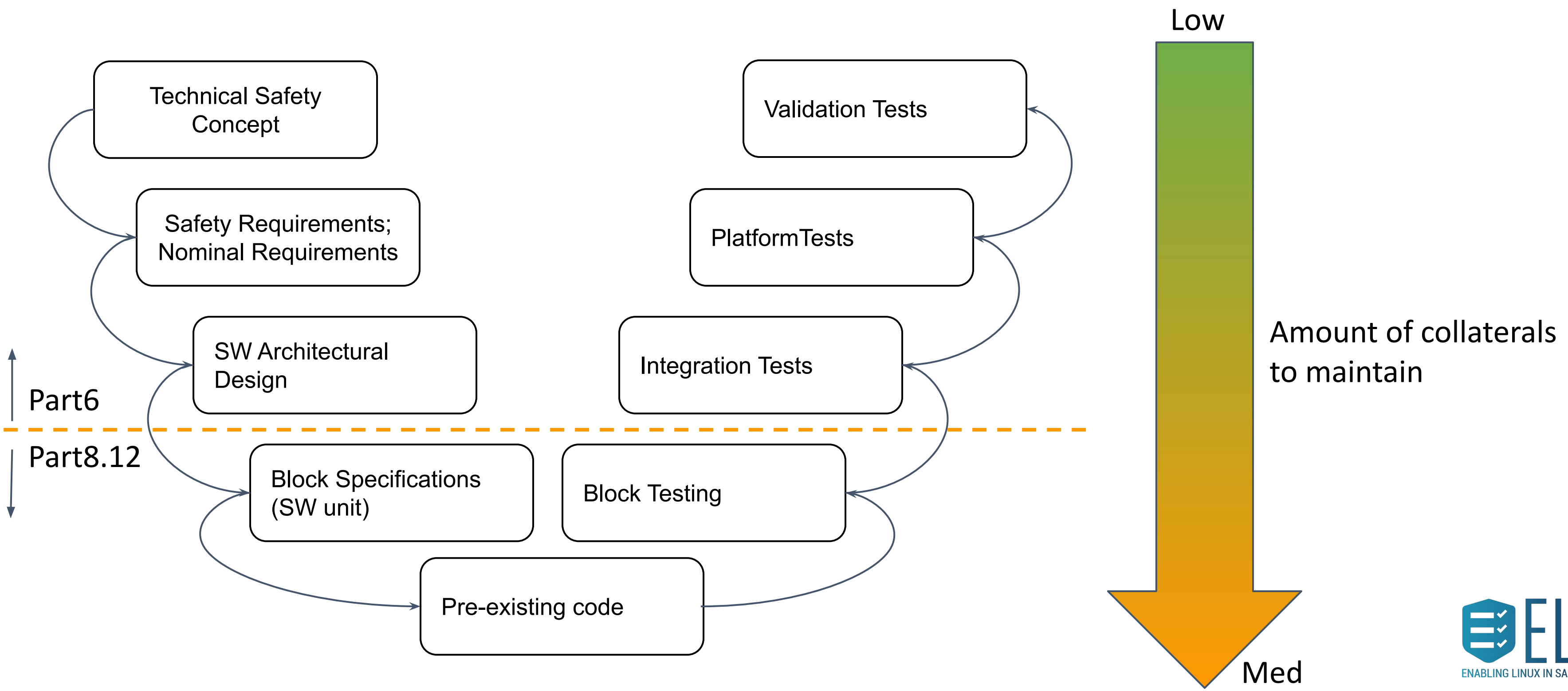
A hybrid safety approach

- Partition Linux in blocks of SW elements
- Define each block as a SW **unit**
- Qualify each SW **unit** according to ISO26262 **Part 8.12**
- Follow ISO26262 **Part 6** to assess the Kernel as an **integration of multiple FuSa qualified units** working together to meet a certain ASIL target



(*): The map of subsystems/drivers is incomplete and is intended to present the concept only

Proposal: Hybrid Approach



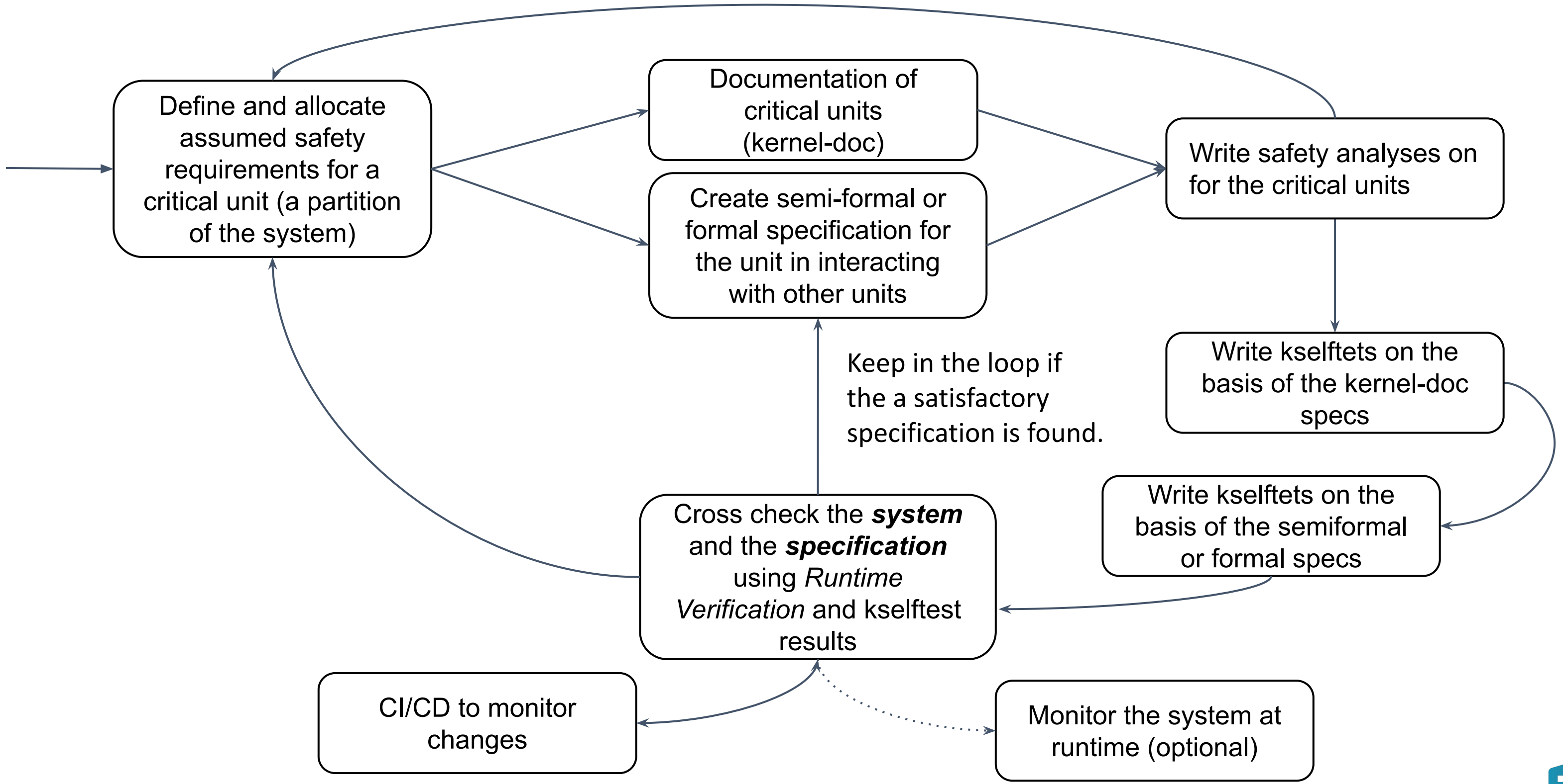
Safety validity of the hybrid approach

- ISO26262-**8.12** is already used to qualify pre-existing SW components of limited complexity
- If a SW component is FuSa qualified it can be integrated into a SW framework assigned with the same or lower ASIL target
  - as long as the assumed safety requirements as well as the conditions of use are defined and met
- In Linux we integrate multiple qualified drivers and subsystems in a hierarchical, scalable way

It is important to decide the criteria for a single subsystem/driver (**unit**) to be ‘simple’ enough to be qualified according to 8.12 or not.

ENABLING LINUX IN SAFETY APPLICATIONS

Proposal – hybrid approach in Linux



ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

ISO26262 Dilemma

How to partition the system into SW **blocks/units** to be qualified according to part8.12?

What is the granularity that makes a **unit simple enough** to be qualified according to 8.12?

What is **the criteria** providing confidence on the right granularity?

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Granularity Criteria

Part8.12 requires the specification of the SW component under qualification in terms of:

- Known safety requirements;
- Functional requirements;
- Behavior in case of failure
- Resource usage
- Description of required and provided interfaces and shared resources
- Configuration Description

If we are able to **specify comprehensively** in natural language all of the specs above, the level of granularity for the **single unit** is the right one

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Linux is already partitioned!

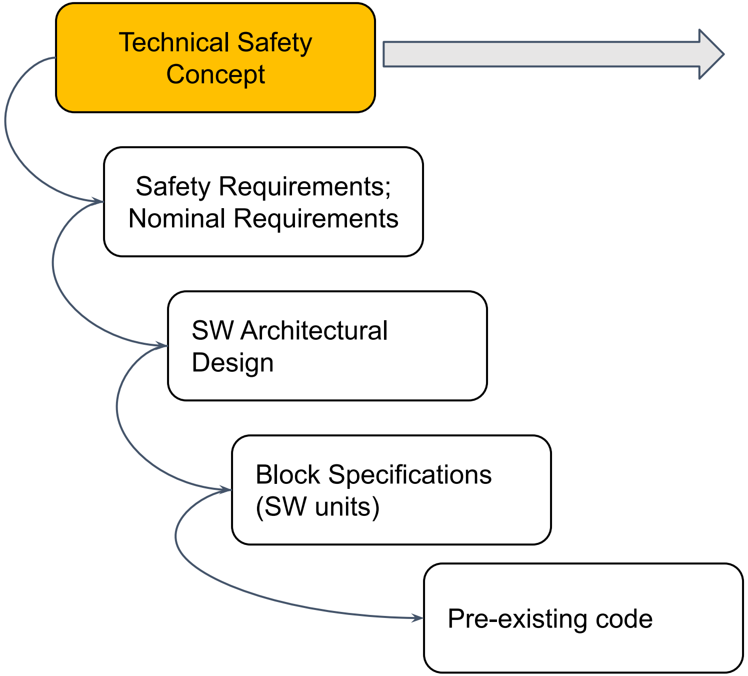
- Linux is already partitioned in subsystems by the MAINTAINERS file[1]
  - **Use the MAINTAINERS granularity as starting point**
- Maintainers are humans!
  - It is **easy to map the code to the responsible for it**
  - But we will need the support from them
- If a subsystem or driver is too complex it can be divided further
  - it is trivial to maintain a **new file defining the partitioning of Linux into our safety units**

In summary MAINTAINERS can be a starting point

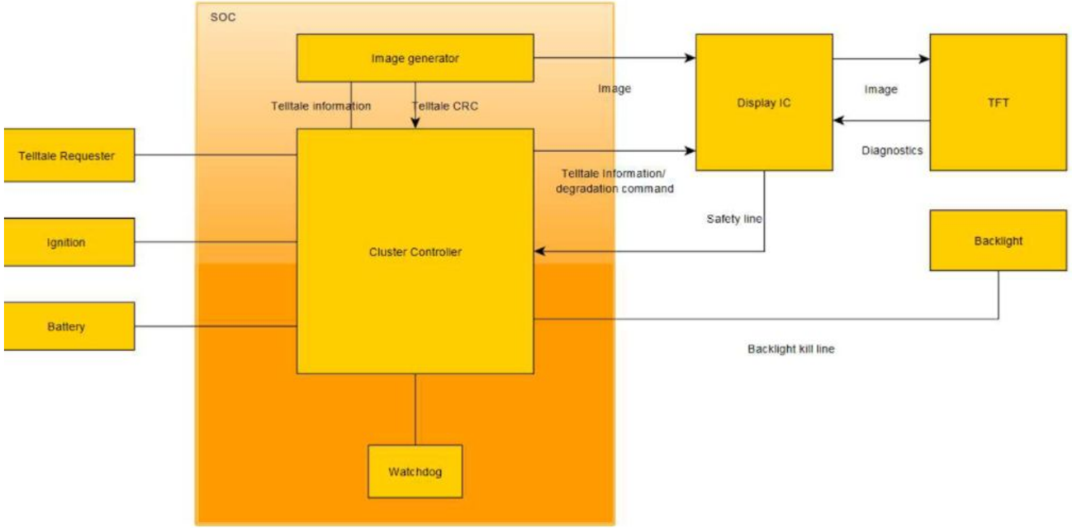
[1] https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

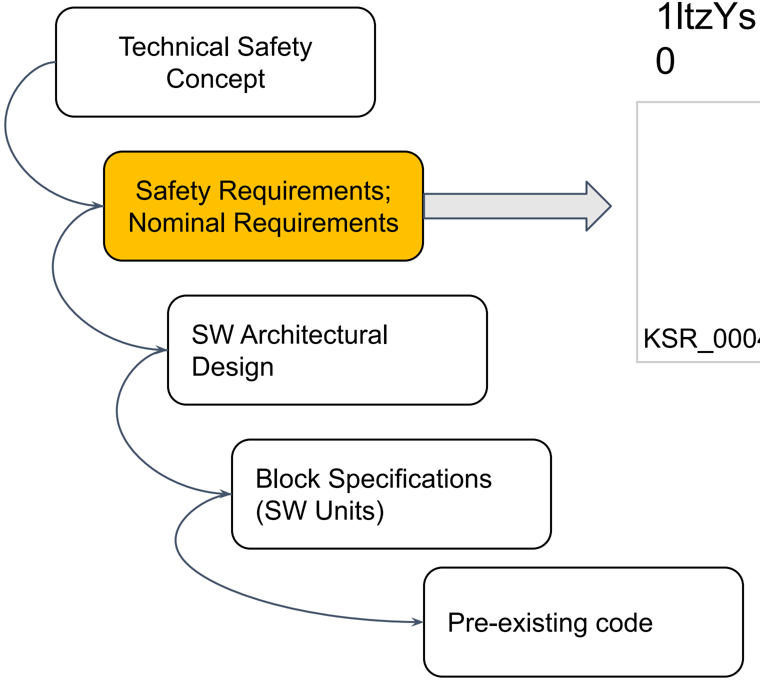
Example: telltale use case



https://github.com/elisa-tech/wg-automotive/blob/master/use-cases/telltale.md



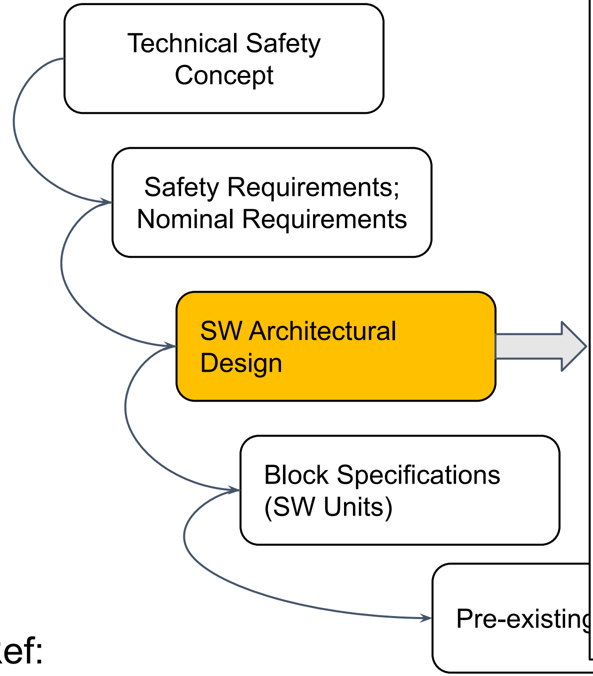
Example: telltale use case



https://docs.google.com/spreadsheets/d/1EKwDUjBGGmO1ltzYsIbZi4K1Zhfr7WB_VfcZNQ0v3UM/edit#gid=385937700

KSR_0004	Watchdog Timeout Setting	**The watchdog subsystem shall ensure the WD timeout to be set according to the IOTCL input parameter**	start_kernel() SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg) [calling watchdog_ioctl()]

Example: telltale use case



To scope the different SW blocks/units supporting ioctl() we used the MAINTAINERS file (a starting point).

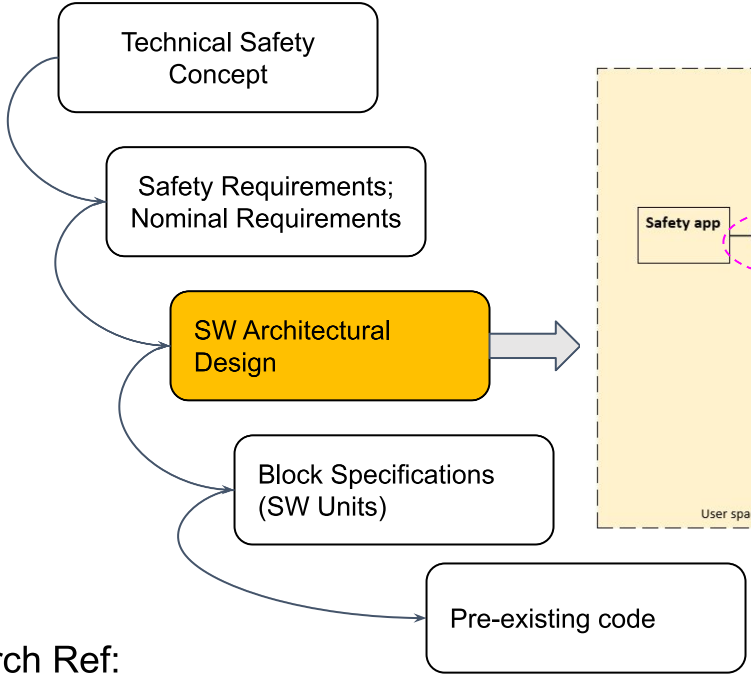
A SW Unit Block is defined as a group of C and H files

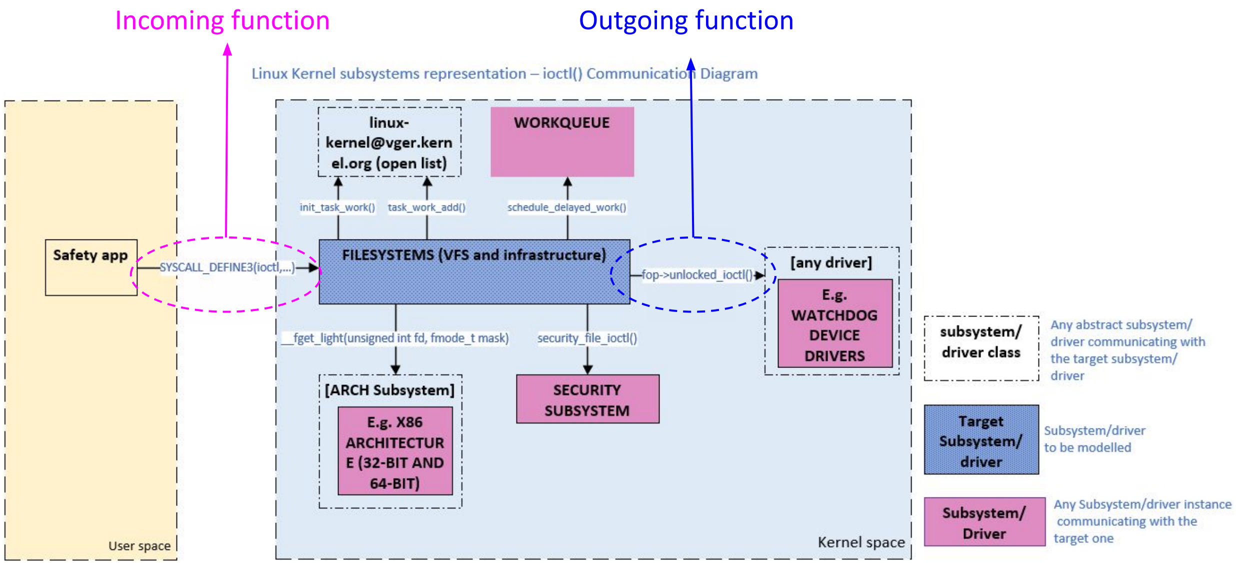
In this deck we focus on the interactions of the SW Unit “FILESYSTEMS (VFS and infrastructure)” with the other SW Units/Blocks:

FILESYSTEMS (VFS and infrastructure)
M: Alexander Viro <viro@zeniv.linux.org.uk>
L: linux-fsdevel@vger.kernel.org
S: Maintained
F: fs/*
F: include/linux/fs.h
F: include/linux/fs_types.h
F: include/uapi/linux/fs.h
F: include/uapi/linux/openat2.h
X: fs/io-wq.c
X: fs/io-wq.h
X: fs/io_uring.c

Ref: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.12#n6896

Example: telltale use case

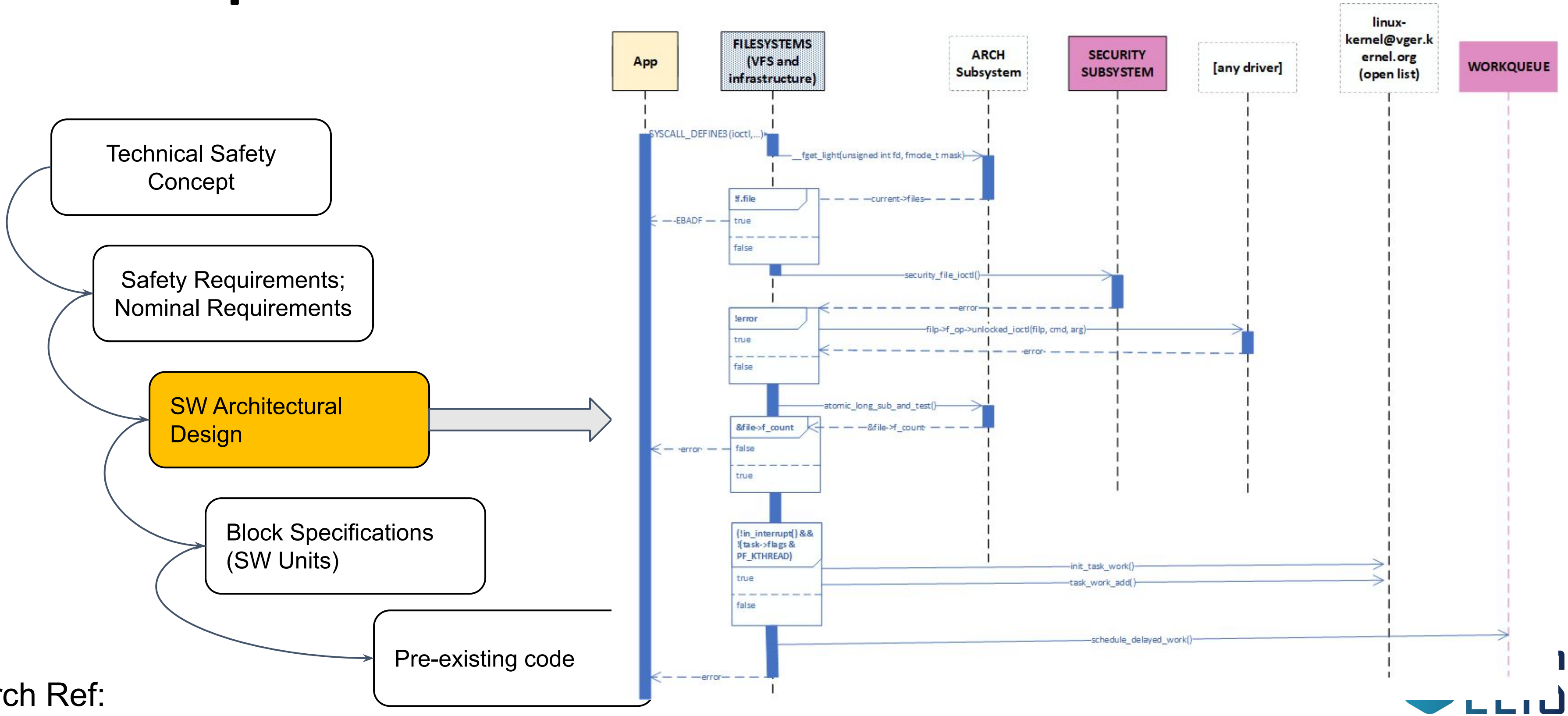




Arch Ref:
https://drive.google.com/file/d/13KJiBJ0XN1SA7So0lVawRWe_3_USQTPN/view?usp=sharing

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

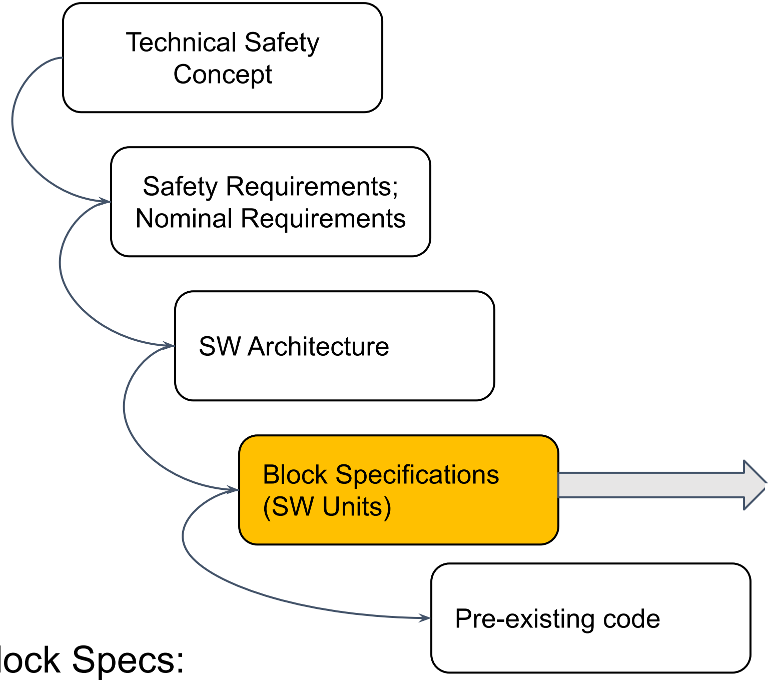
Example: telltale use case



Arch Ref:
https://drive.google.com/file/d/13KJiBJ0XN1SA7So0lVawRWe_3_USQTPN/view?usp=sharing

ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case

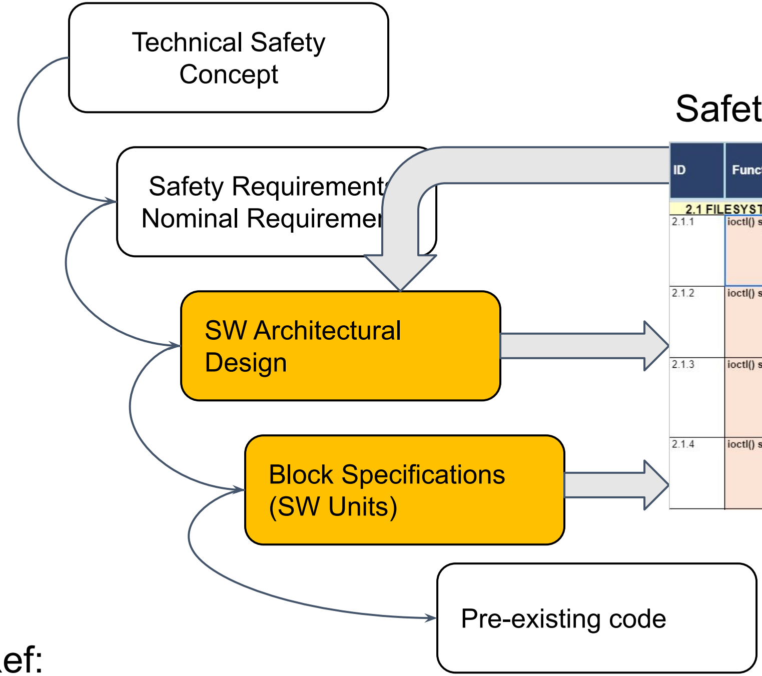



```
/*
 *      SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned
 *   long, arg): Kernel entrypoint for the ioctl() syscall.
 *      @fd: input file descriptor
 *      @cmd: command value
 *      @arg: pointer address to user data
 *
 *      When ioctl() is invoked, the following steps are
 *      performed:
 *      - the file descriptor structure is retrieved from the file descriptor
 *        table associated with the current task. If the file descriptor table
 *        is shared the associated reference count is incremented.
 *        Failing to retrieve the fd structure results in -EBADF being returned
 *      - security_file_ioctl() is called to check if permissions are in place
 *        to execute the ioctl(); if no permissions an error code is returned
 *      - if permissions are in place; the file structure associated to the file
 *        descriptor is retrieved, the unlocked_ioctl() registered callback is
 *        checked and, if not NULL, it is called.
 *        If the unlocked_ioctl() function pointer is NULL -ENOTTY is returned.
 *        If unlocked_ioctl() succeeds 0 is returned, otherwise the driver
 *        specific error value is returned
 *      - the reference counter is decreased, if zero the last reference to the
 *        file is released (see __fput())
 *
 *      Return: on success zero is returned, otherwise one of the appropriate
 *      error codes as per description above
 *
 *      TODO: documentation is missing for the following CMDs: FIOCLEX,
 *      FIONCLEX, FIONBIO, FIOASYNC, FIOQSIZE, FIFREEZE, FITHAM, FS_IOC_FIEMAP,
 *      FIGETBSZ, FICLONE, FICLONERANGE, FIDEDUPERANGE, FIBMAP, FIONREAD,
 *      FS_IOC_RESVSP, FS_IOC_RESVSP64
 *
 */
```

Block Specs:
https://docs.google.com/document/d/1BV1dysXPXoUH2_A5dMxZwoXStqni-hVlJKqeLoaeS4I/edit

ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case



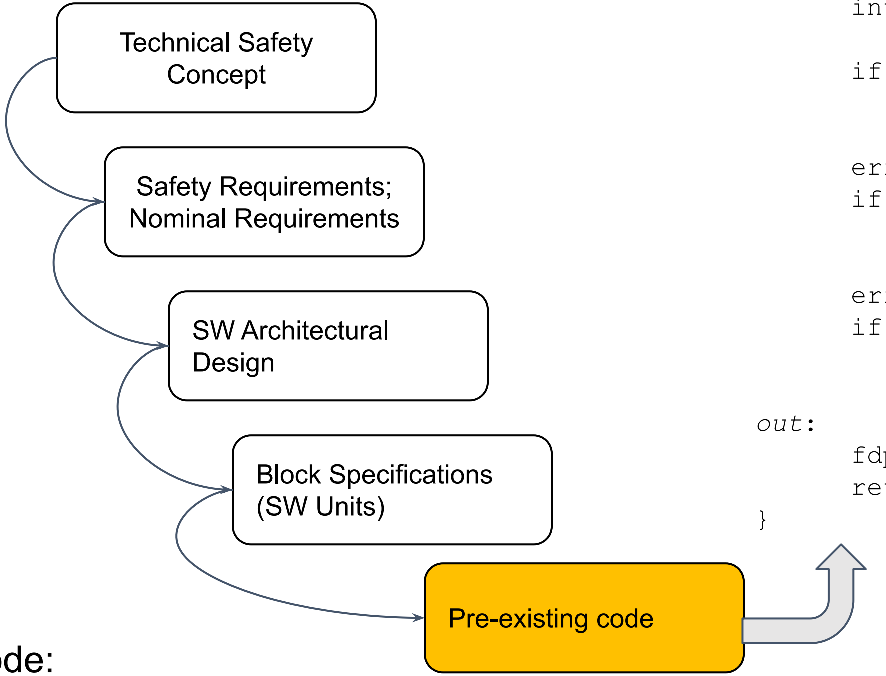
Safety Analysis

ID	Functionality	Kernel Subsystem	Guide Word	API / Function	Component Failure Mode	Component Failure Description
2.1 FILESYSTEMS (VFS and infrastructure)						
2.1.1	ioctl() syscall		Erroneous data management and data corruption	SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)	ksys_ioctl() corrupts the fd paramenters before retrieving the associated file descriptor structure	the user input parameter fd gets corrupted before the file struct is retrieved; hence a parameter that is different from the intended one is passed to retrieve the file struct
2.1.2	ioctl() syscall		Erroneous data management and data corruption	SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)	wrong file pointer is returned	wrong file pointer is retrieved from the input file descriptor
2.1.3	ioctl() syscall		Erroneous data management and data corruption	SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)	file reference counter is not increased (in case of shared fd) or it is corrupted	
2.1.4	ioctl() syscall		Erroneous data management and data corruption	SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg)	execution continues even if access is denied	

Ref:
https://drive.google.com/file/d/1-qfyfWJasfXc3IES7RUtfnnxkVsKkOkl/view?usp=sharing

ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case

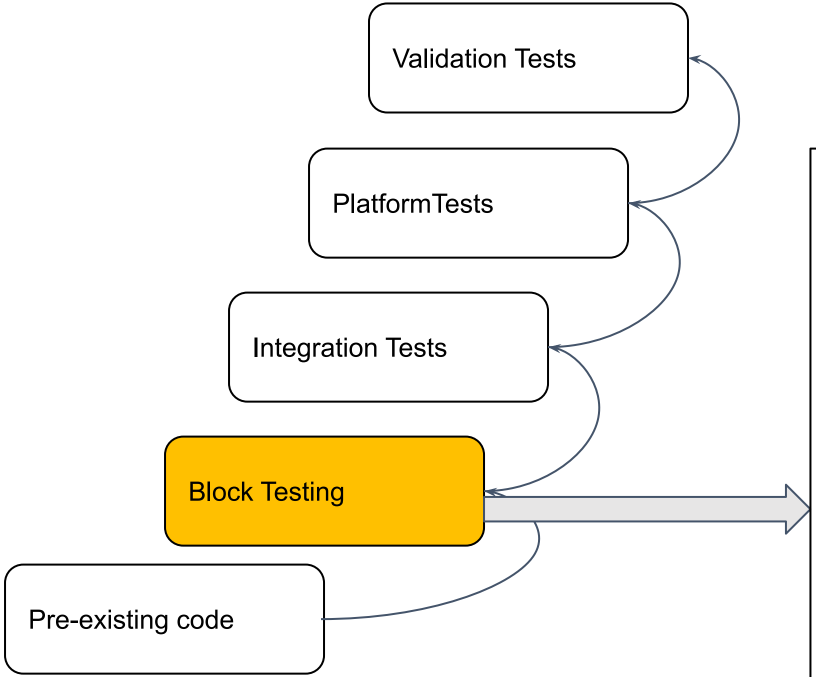



```
SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned
long, arg)
{
	struct fd f = fdget(fd);
	int error;

	if (!f.file)
		return -EBADF;

	error = security_file_ioctl(f.file, cmd, arg);
	if (error)
		goto out;

	error = do_vfs_ioctl(f.file, fd, cmd, arg);
	if (error == -ENOIOCTLCMD)
		error = vfs_ioctl(f.file, cmd, arg);

out:
	fdput(f);
	return error;
}
```

code:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/fs/ioctl.c?h=v5.12#n739

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case



Kernel Selftests can be used to define a comprehensive test campaign for the block “FILESYSTEMS (VFS and infrastructure)” wrt the ioctl() scenario.

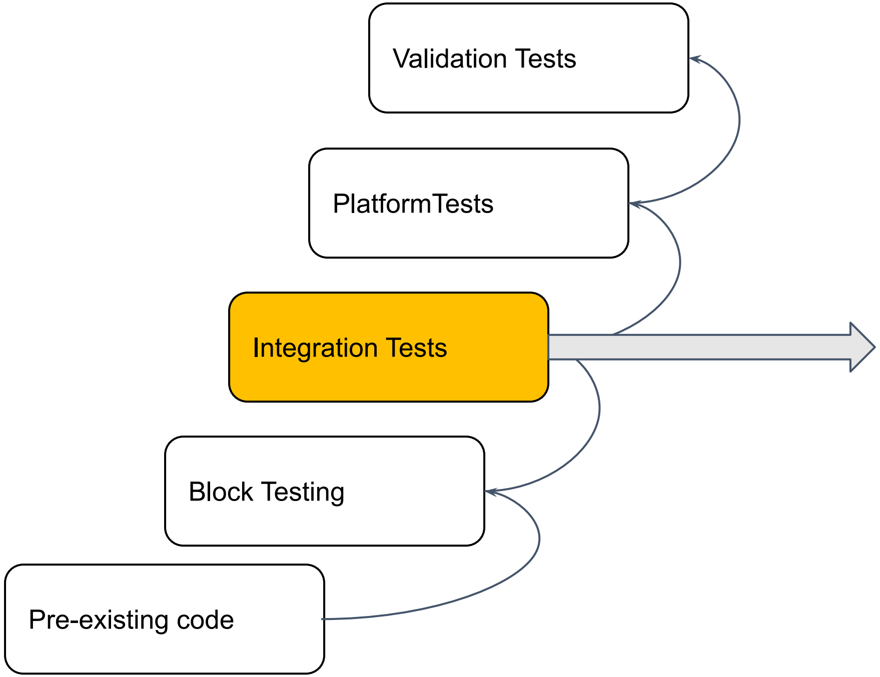
The test specifications can be compared against the SW architectural models, against the kernel-doc headers specifications and against the safety analysis to build confidence on the test campaign completeness

TODO: need to evaluate the current Kernel Selftests for ioctl() against the block specs

https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/tools/testing/selftests?h=v5.12

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case



The SW Architecture diagrams built for the ioctl() scenario are automatically implemented in **runtime verification monitors** that can be used in the verification phase to make sure the code to behave as modelled

If either the code is wrong or the model is wrong, an exception if raised and the test fails

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Runtime Verification (RV)

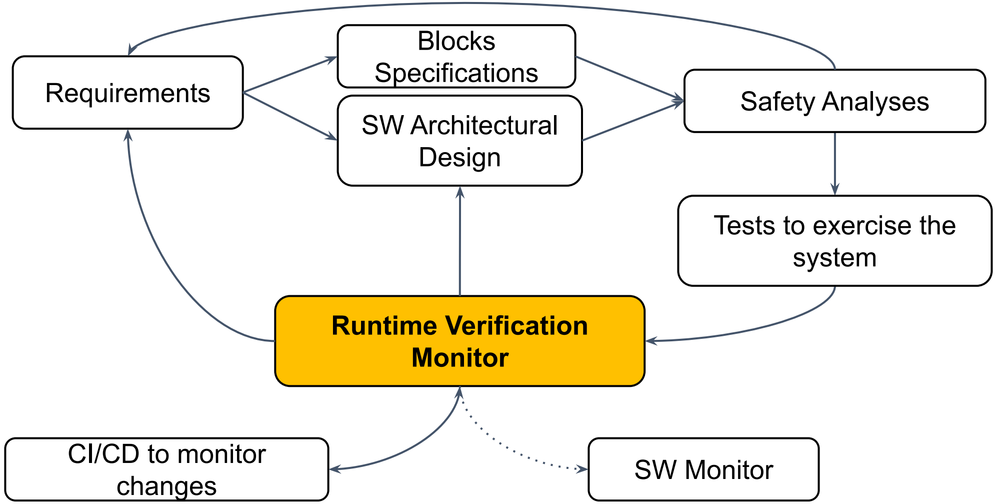
- Runtime Verification (RV) is a lightweight (yet rigorous) **formal verification method**
  - It complements other formal methods (such as *model checking* and *theorem proving*)

- RV works by analyzing the trace of the system's actual execution, comparing it against a formal specification of the system behavior

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

RV in the approach: why do we care?

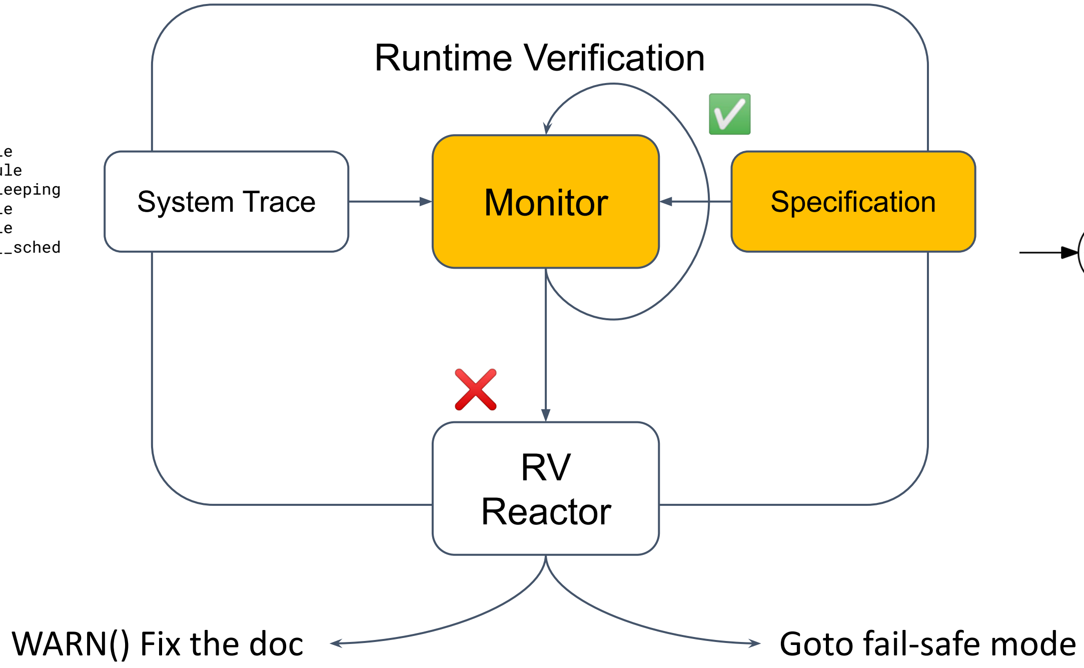
- It closes the loop between the kernel and the specification
- Cross verify the system and the documentation
  - *It allows us to "run" the documentation in kernel.*
- Enable the continuous integration tests
- Perform runtime monitoring of the system



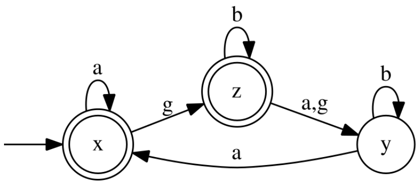
Runtime Monitor (RV)

Linux Realm

```
247309: schedule <-worker_thread
247309: preempt_count_add <-schedule
247309: wq_worker_sleeping <-schedule
247309: kthread_data <-wq_worker_sleeping
247310: preempt_count_sub <-schedule
247310: preempt_count_add <-schedule
247310: rcu_note_context_switch <-__sched
```

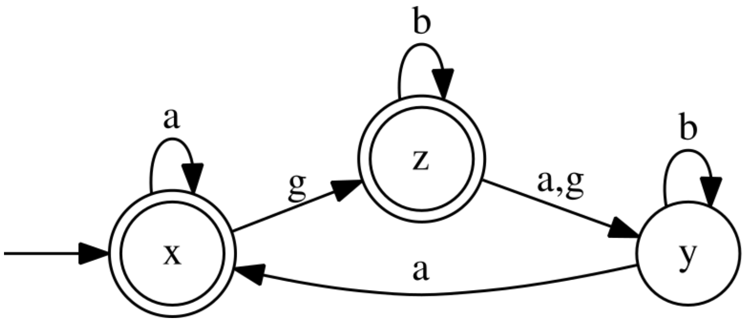


Formal Realm



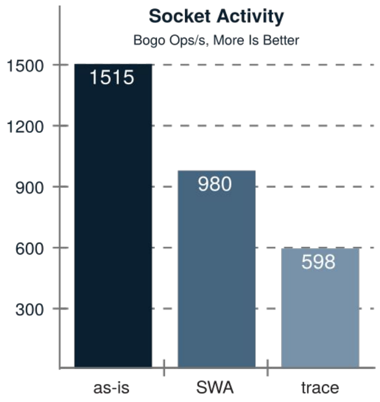
Automata based Runtime Verification

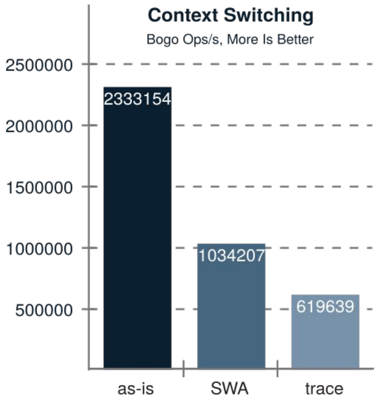
- Over the last years, a RV method using automata theory has been refined
- Automata is flexible, intuitive and can be used to specify complex parts of the system:
    - See paper: **A Thread Synchronization Model for the PREEMPT_RT Linux Kernel** (+9k states, +21k transitions)
    - Build from small specifications (all < 10 states)

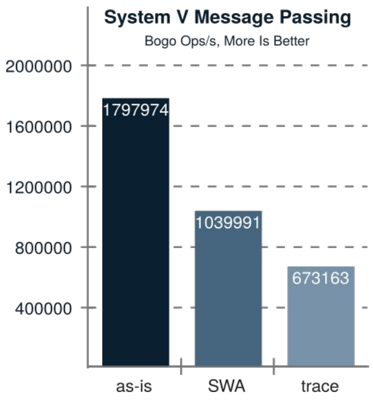


Automata based Runtime Verification

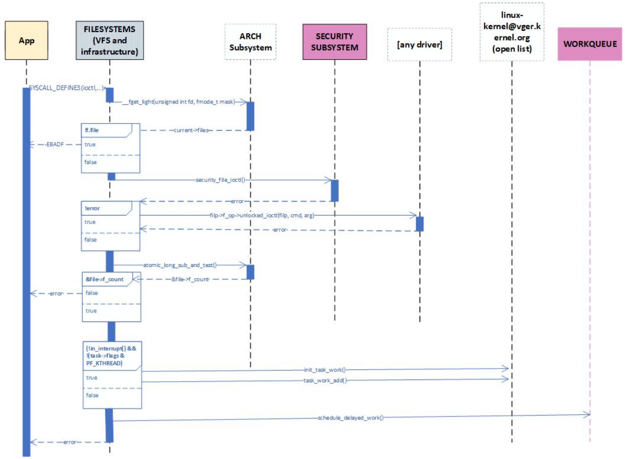
- It is faster to verify the system online than just saving the trace for later analysis
  - See Paper: **Efficient Formal Verification for the Linux Kernel**

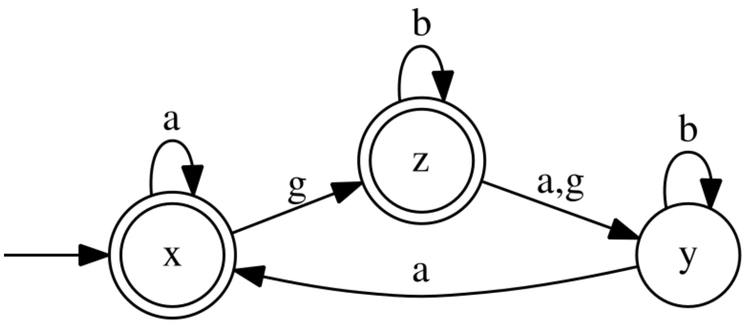






Hybrid approach and Runtime Verification





RV interface and dot2k

- Runtime Verification Interface for the Linux kernel is on submission to LKML
  - **The Runtime Verification (RV) interface**
  - https://lore.kernel.org/lkml/cover.1621414942.git.bristot@redhat.com/
- A **dot2k** tool that automatically generate the runtime monitor code
  - The developer only needs to do the instrumentation
    - Connect the specification events o the kernel events
- An **intuitive interface** to control monitors of the system
  - It is based on Linux kernel trace interface

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Automatic monitor generation

- Automatic code generation is as easy as:
  - $ dot2k -d ~/wip.dot -t per_cpu
  - See [1]
- The work left to be done is the connection between the model events and the kernel events
  - It uses the existing kernel trace infrastructure, an event can be:
    - A tracepoint
    - A function
    - A kprobe...
- See [2] for an example of instrumentation

[1] https://lore.kernel.org/lkml/84ea1e70b846e6fefdaafe4ce5e3c1a5cb49aace.1621414942.git.bristot@redhat.com/
[2] https://lore.kernel.org/lkml/8ffcb3a4c8b55ef63cc02b487aa1c8ad5bf3f800.1621414942.git.bristot@redhat.com/

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

RV user-interface

- Based on ftrace
- Enabling a monitor and instructing it to panic() the system if an exception is found is as easy as:

```
[root@f32 ~/] # cd /sys/kernel/tracing/rv/
[root@f32 ~/] # echo panic > monitors/wip/reactors
[root@f32 rv] # echo wip > enabled_monitors
```

- Developer can watch the monitor via ftrace

```
kworker/u8:0-1150    [003] ...2 12430.492850: event_wip: preemptive x preempt_disable -> non_preemptive
kworker/u8:0-1150    [003] ...2 12430.492850: event_wip: non_preemptive x preempt_enable -> preemptive (safe)
```

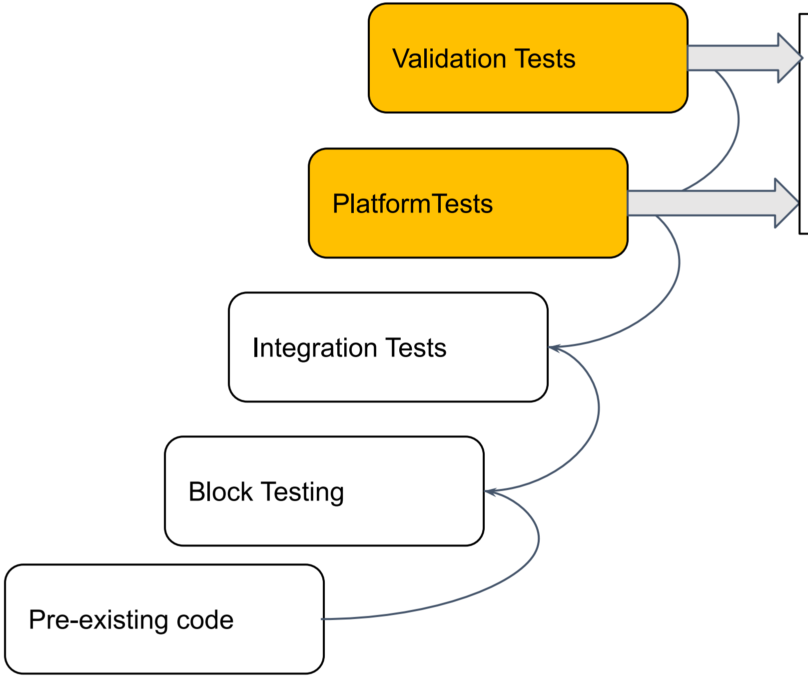
ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

For further information

- Last three Red Hat Research Quarterly presents the RV modeling and verification approach
- DE OLIVEIRA, Daniel Bristot; CUCINOTTA, Tommaso; DE OLIVEIRA, Rômulo Silva. *Efficient formal verification for the Linux kernel.* In: International Conference on Software Engineering and Formal Methods. Springer, Cham, 2019. p. 315-332.
- DE OLIVEIRA, Daniel B.; DE OLIVEIRA, Rômulo S.; CUCINOTTA, Tommaso. *A thread synchronization model for the PREEMPT_RT Linux kernel.* Journal of Systems Architecture, 2020, 107: 101729.
- Formal Verification made easy and fast (ELCE 2019)
  - https://www.youtube.com/watch?v=BfTuEHafNgg

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Example: telltale use case



Top level Platform and Validation tests can be carried out according to the allocated Kernel Safety Requirements and Safety Requirements from the Telltale Safety concept

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Next Steps

- Complete the evaluation of the hybrid approach in the context of the telltale use case from the Automotive WG
- Refine, finalize and build consensus on the hybrid approach in the Development Process WG
- Develop e refine tools augmenting and supporting the generation of SW architectural models starting from the code
- Continue the development of the Runtime Verification Interface
- Go high scale by pushing the tools and engaging with maintainers

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

Questions?

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS

BCKP

ELISA
ENABLING LINUX IN SAFETY APPLICATIONS