LINUX PLUMBERS CONFERENCE September 20-24, 2021

A maintainable, scalable, and verifiable SW architectural design model for the Linux Kernel

Gabriele Paoloni (Red Hat)
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..

Agenda

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

LINUX PLUMBERS CONFERENCE September 20-24, 2021

In/out of the scope of this presentation

In Scope:

- Proposal and high level description of an architectural and unit design model suitable to meet ISO26262 requirements
- SW Verification methodology associated to the architectural model

Out of Scope:

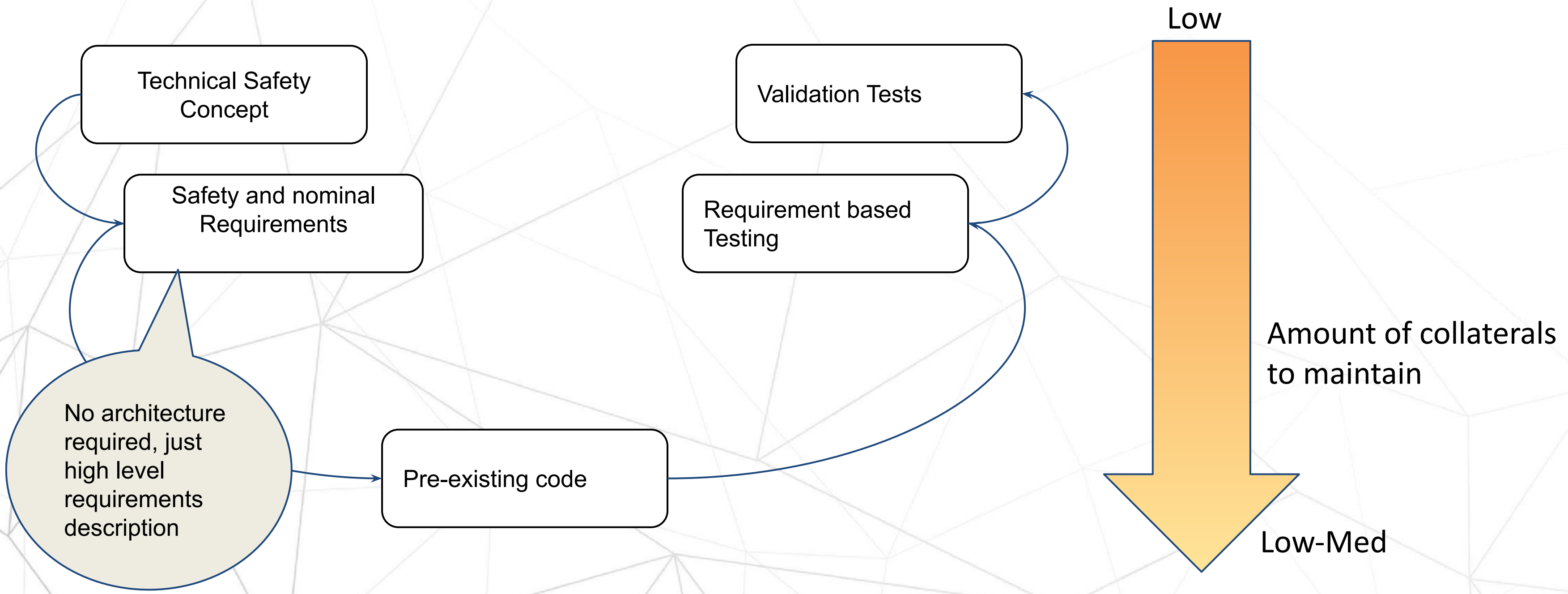
- Overall FuSa Qualification Strategy of Linux
- Any safety standard beyond ISO26262

ISO26262 Introduction

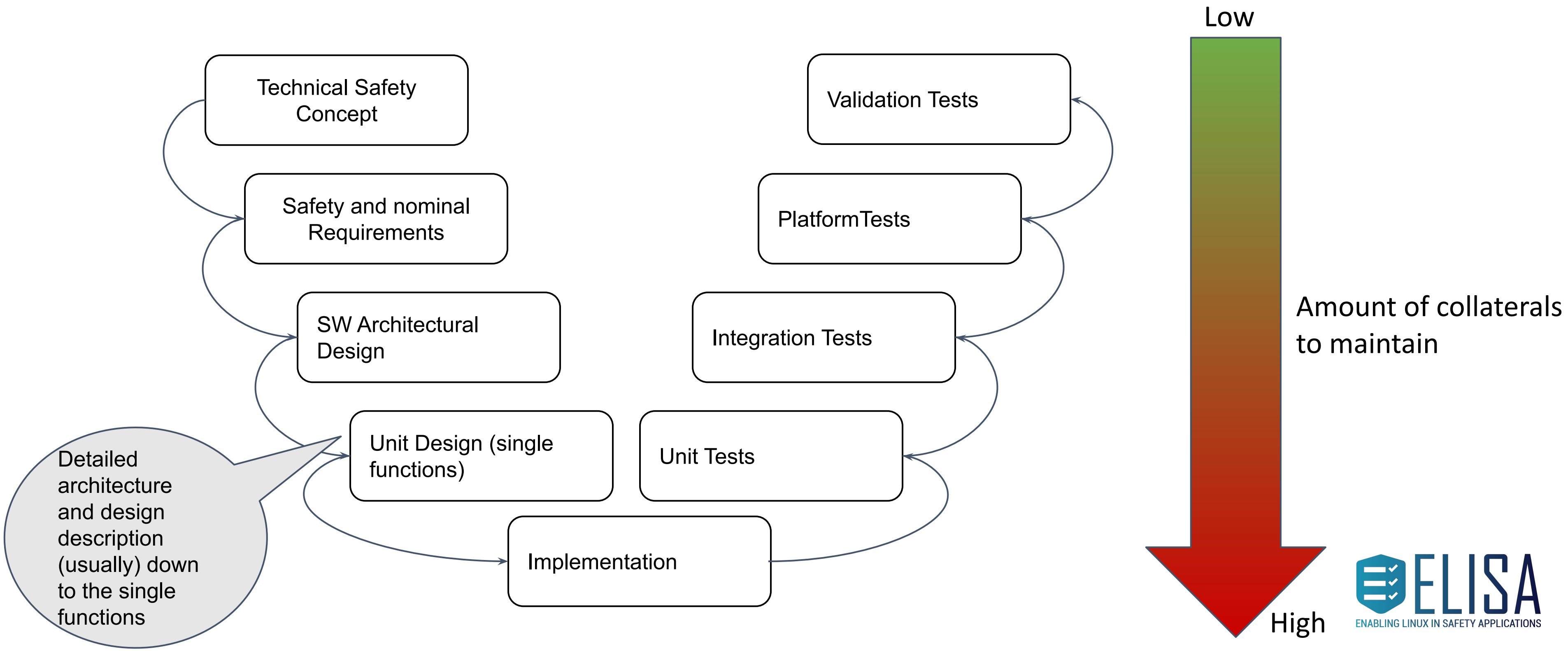
Our Architectural Design approach is tailored to leverage both part6 and part8.12 together

- 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
  - 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**

Part 8.12 Standard Approach


Technical Safety Concept
Safety and nominal Requirements
No architecture required, just high level requirements description
Pre-existing code
Requirement based Testing
Validation Tests
Low
Amount of collaterals to maintain
Low-Med

Part 6 Standard Approach


Technical Safety Concept
Safety and nominal Requirements
SW Architectural Design
Unit Design (single functions)
Implementation
Unit Tests
Integration Tests
PlatformTests
Validation Tests
Detailed architecture and design description (usually) down to the single functions
Low
Amount of collaterals to maintain
High

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
  - Linux 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.

Out Of Scope for this session

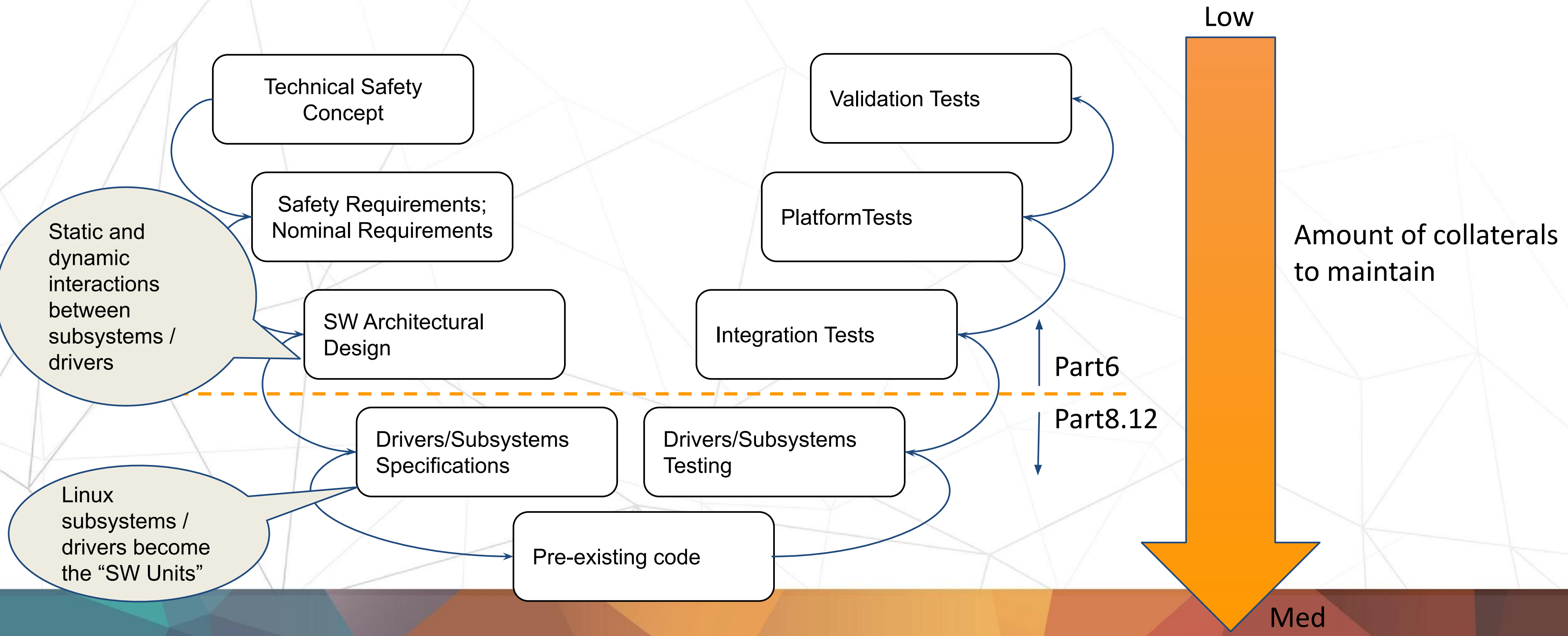
ISO26262 Dilemma:

Linux is too complex for **Part 8.12**

✕

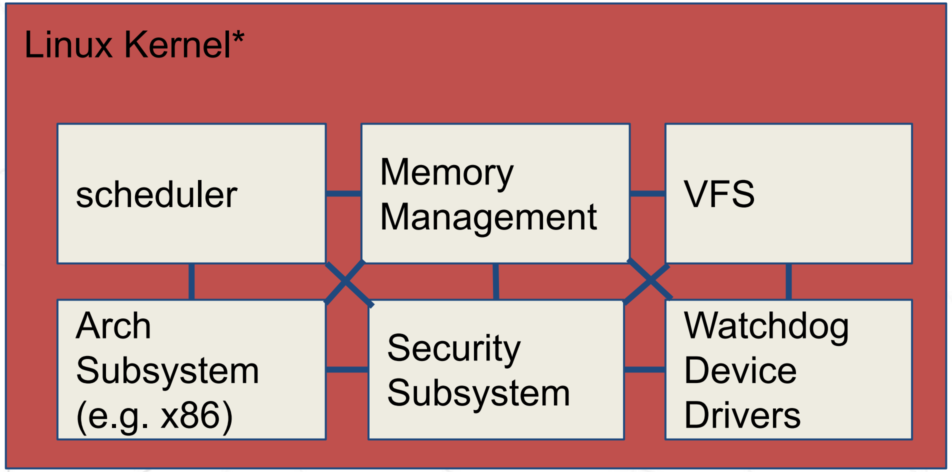
Part 6 is too complex for Linux

Proposal: a Tailored Architectural Model


Technical Safety Concept
Safety Requirements; Nominal Requirements
SW Architectural Design
Drivers/Subsystems Specifications
Pre-existing code
Drivers/Subsystems Testing
Integration Tests
PlatformTests
Validation Tests
Static and dynamic interactions between subsystems / drivers
Linux subsystems / drivers become the “SW Units”
Part6
Part8.12
Low
Med
Amount of collaterals to maintain

Tailored Architectural Model

- Partition Linux in blocks of SW elements
- Define each subsystem/driver (or part of it) as a SW **unit**
- For each SW **unit** the design specs can be defined through natural language using the kernel-doc headers
- Static and dynamic interactions between SW units are described using semi-formal or formal notation



🟥 The interactions between SW units follow part6.7

⬜ SW Units design specs become the top level requirements according to part8.12

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

ISO26262 Dilemma

How to partition the system into SW **blocks/units**?

What is the granularity that makes a SW **unit simple enough** to describe its design using kernel-doc headers ?

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

Granularity Criteria (proposal)

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

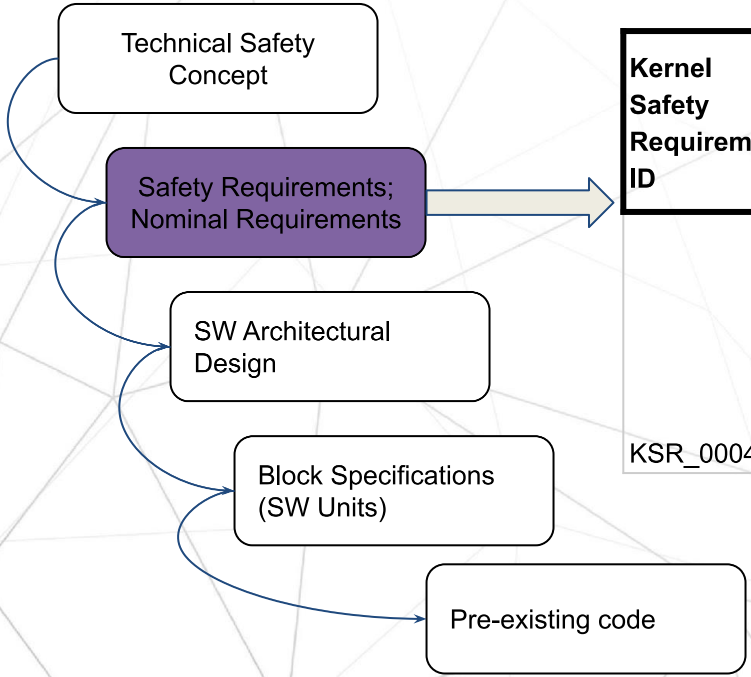
Linux is already partitioned!

- 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

Example: watchdog timeout setting

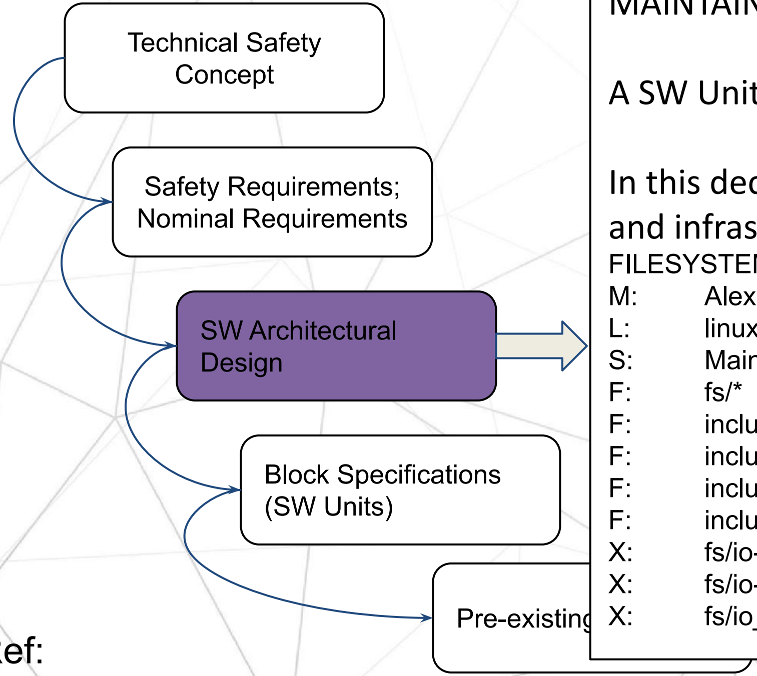

Technical Safety Concept
Safety Requirements; Nominal Requirements
SW Architectural Design
Block Specifications (SW Units)
Pre-existing code

Kernel Safety Requirement ID	**Kernel Safety Requirement Title**	**Kernel Safety Requirement Description**	**Kernel Entrypoints**
KSR_0004	Watchdog Timeout Setting	**The watchdog subsystem shall ensure the WTD timeout to be set according to the IOCTL input parameter**	start_kernel() SYSCALL_DEFINE3(ioctl, unsigned int, fd, unsigned int, cmd, unsigned long, arg) [calling watchdog_ioctl()]

Ref:
https://docs.google.com/spreadsheets/d/1EbuVvhXo-xZc2aPTfMgQtPNPDQYtcozs/edit#gid=584539121

LINUX PLUMBERS CONFERENCE September 20-24, 2021

Example: watchdog timeout setting


Technical Safety Concept
Safety Requirements; Nominal Requirements
SW Architectural Design
Block Specifications (SW Units)
Pre-existing

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:

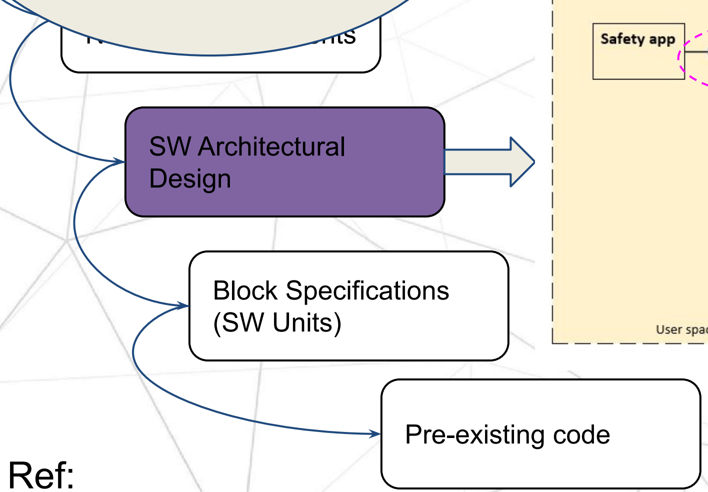
```
FILESYSTEMS (VFS and infrastructure)
M:	Alexander Viro <viro@zeniv.linux.org.uk>
L:	linux-fsdevel@vger.kernel.org
S:	Maintained
F:	fs/*
F:	include/linux/fs.h
F:	include/linux/fs_types.h
F:	include/uapi/linux/fs.h
F:	include/uapi/linux/openat2.h
X:	fs/io-wq.c
X:	fs/io-wq.h
X:	fs/io_uring.c
```

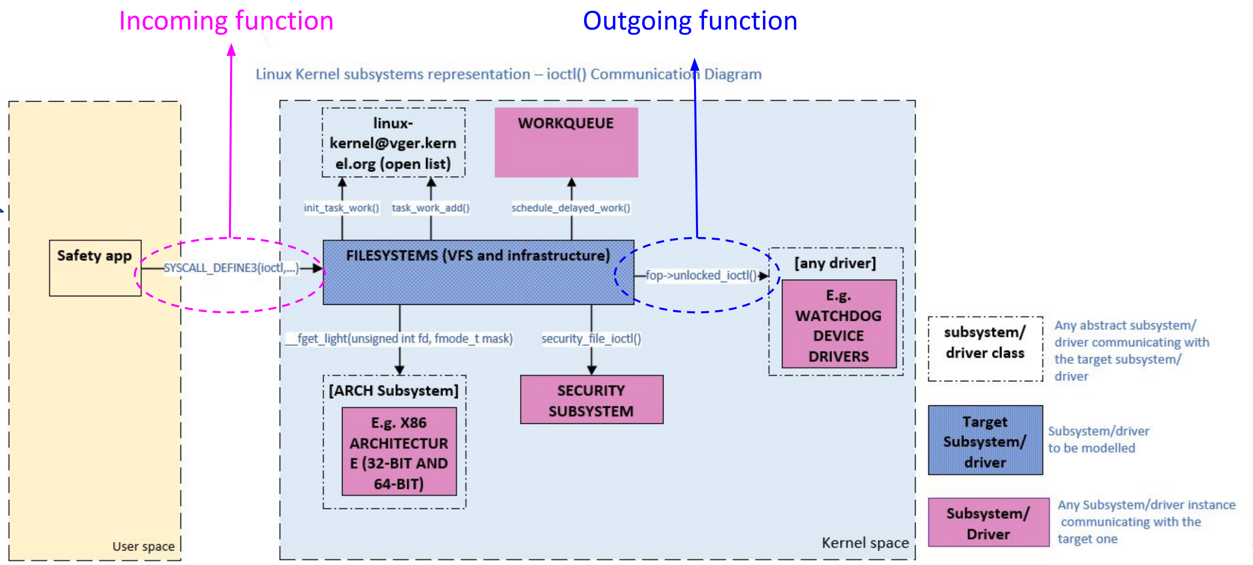
Ref:
https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/MAINTAINERS?h=v5.12#n6896

LINUX PLUMBERS September 20-24, 2021

Example: watchdog timeout setting

The Communication Diagram provides a static view of the relationships between the target SW Unit (here VFS) and the ones it communicates with (for this use case)



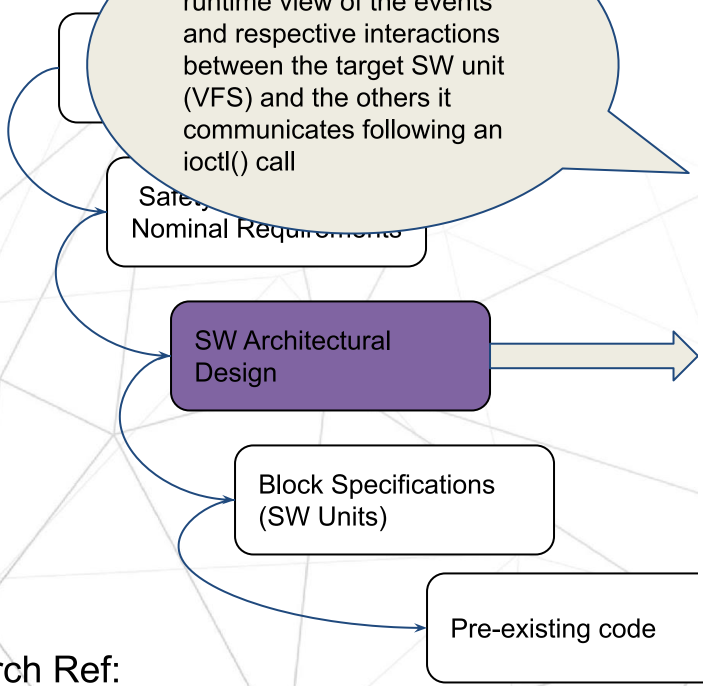


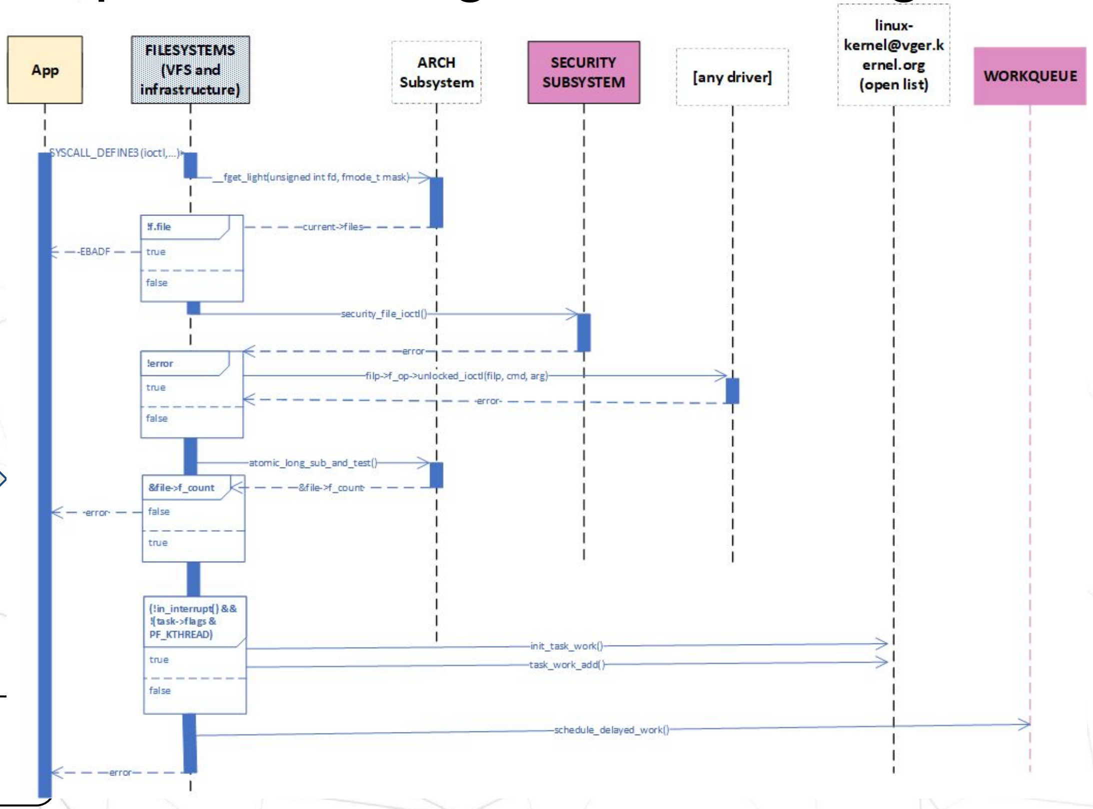
Incoming function
Outgoing function
Linux Kernel subsystems representation – ioctl() Communication Diagram
Safety app
User space
SYSCALL_DEFINE3(ioctl,...)
linux-kernel@vger.kernel.org (open list)
WORKQUEUE
init_task_work()
task_work_add()
schedule_delayed_work()
FILESYSTEMS (VFS and infrastructure)
fop->unlocked_ioctl()
[any driver]
E.g. WATCHDOG DEVICE DRIVERS
__fget_light(unsigned int fd, fmode_t mask)
security_file_ioctl()
[ARCH Subsystem]
E.g. X86 ARCHITECTURE (32-BIT AND 64-BIT)
SECURITY SUBSYSTEM
Kernel space
subsystem/ driver class
Any abstract subsystem/ driver communicating with the target subsystem/ driver
Target Subsystem/ driver
Subsystem/driver to be modelled
Subsystem/ Driver
Any Subsystem/driver instance communicating with the target one

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

Example: watchdog timeout setting

The flow diagram provides a runtime view of the events and respective interactions between the target SW unit (VFS) and the others it communicates following an ioctl() call

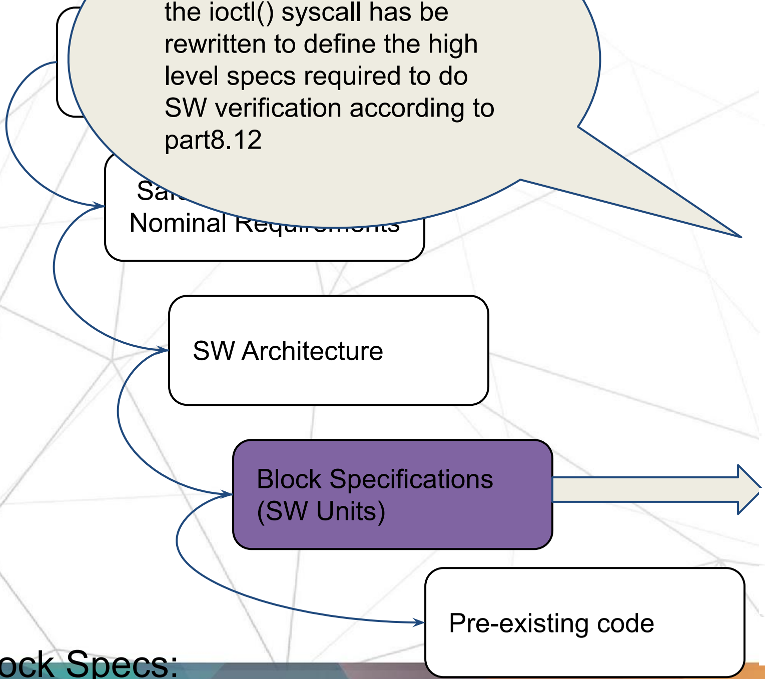

Safety ... Nominal Requirements
SW Architectural Design
Block Specifications (SW Units)
Pre-existing code


App
FILESYSTEMS (VFS and infrastructure)
ARCH Subsystem
SECURITY SUBSYSTEM
[any driver]
linux-kernel@vger.kernel.org (open list)
WORKQUEUE
SYSCALL_DEFINE3(ioctl,...)
__fget_light(unsigned int fd, fmode_t mask)
!f.file
current->files
-EBADF
true
false
security_file_ioctl()
error
!error
filp->f_op->unlocked_ioctl(filp, cmd, arg)
true
error
false
atomic_long_sub_and_test()
&file->f_count
&file->f_count
error
false
true
(!in_interrupt() && !(task->flags & PF_KTHREAD)
init_task_work()
true
task_work_add()
false
schedule_delayed_work()
error

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

Example: watchdog timeout setting

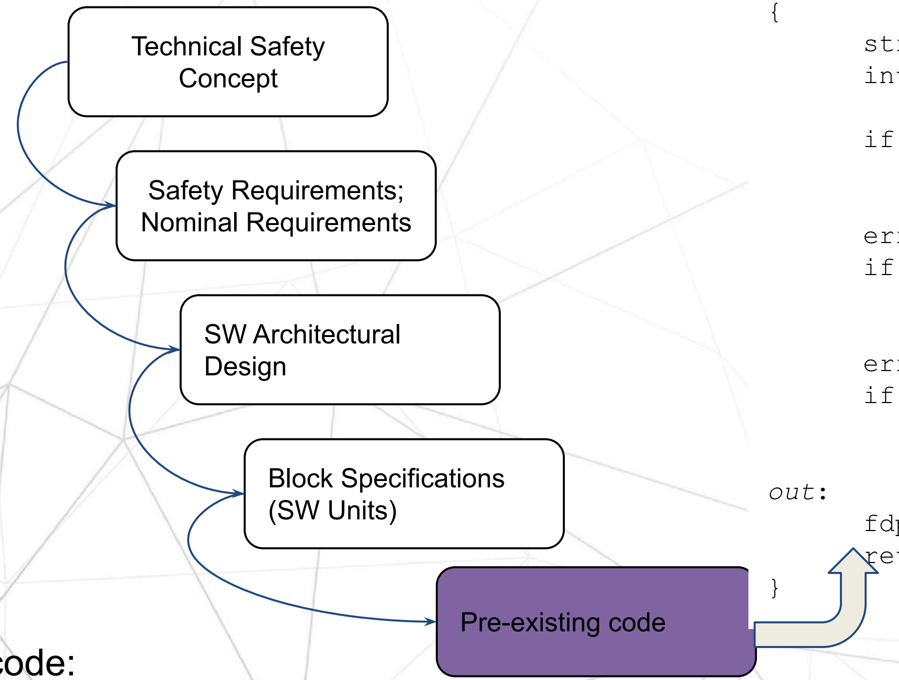
The kernel-doc header of the ioctl() syscall has be rewritten to define the high level specs required to do SW verification according to part8.12



```
/*
 *      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

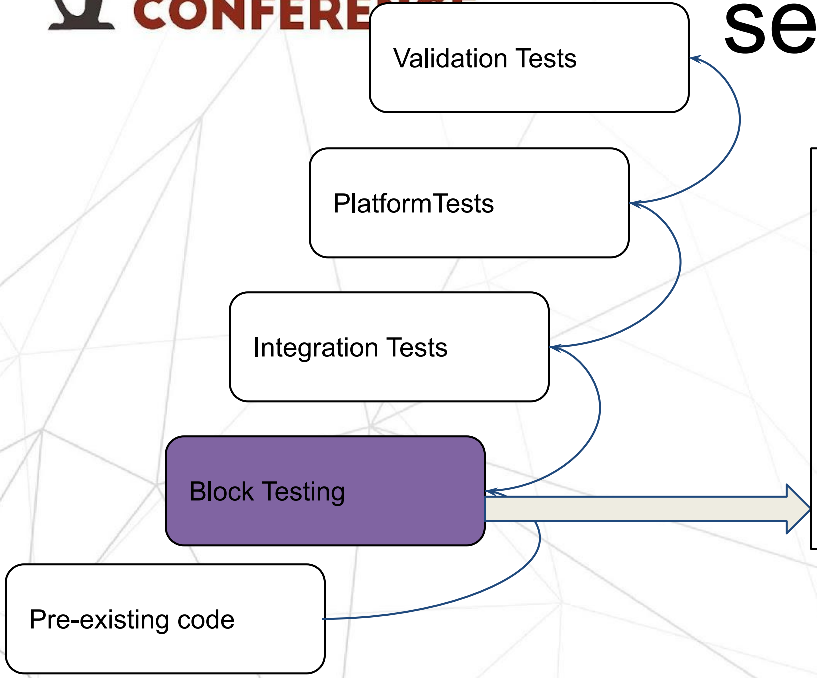
Example: watchdog timeout setting


Technical Safety Concept
Safety Requirements; Nominal Requirements
SW Architectural Design
Block Specifications (SW Units)
Pre-existing code

```
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

Example: watchdog timeout setting

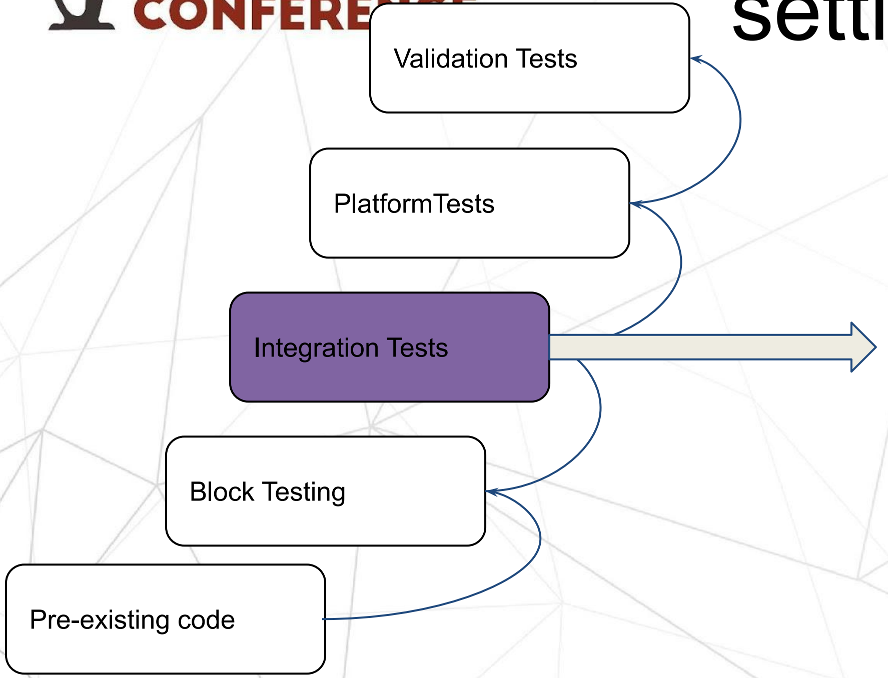


Kernel Selftests can be used to define a comprehensive test campaign for the block “FILESYSTEMS (VFS and infrastructure)” wrt the ioctl() scenario.

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

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

Example: watchdog timeout setting



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 is behaving as modelled

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

Runtime Verification (RV)

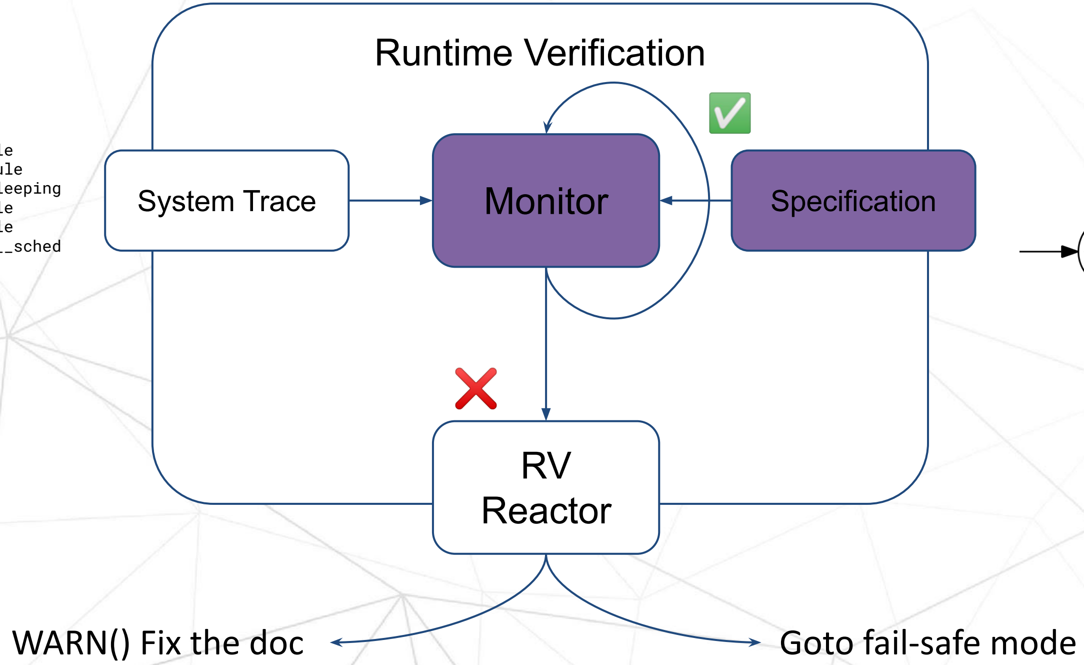
- Runtime Verification (RV) is a lightweight (yet rigorous) **formal verification method**
  - It complements other formal methods (such as *model checking* and *theorem proving*)

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

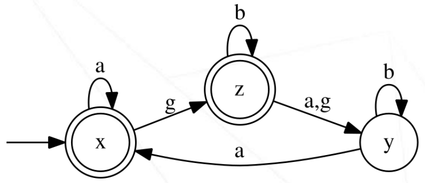
Runtime Monitor (RV)

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
```

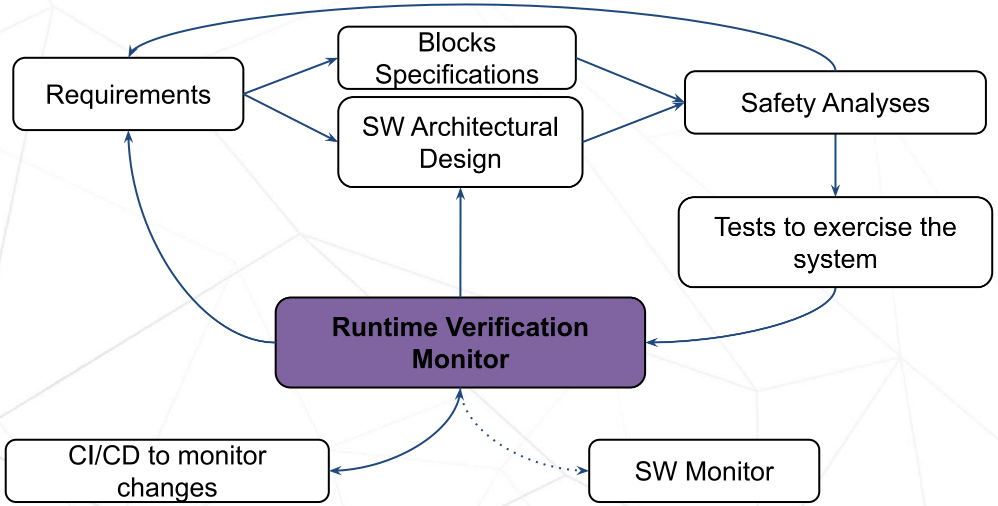


Formal Realm

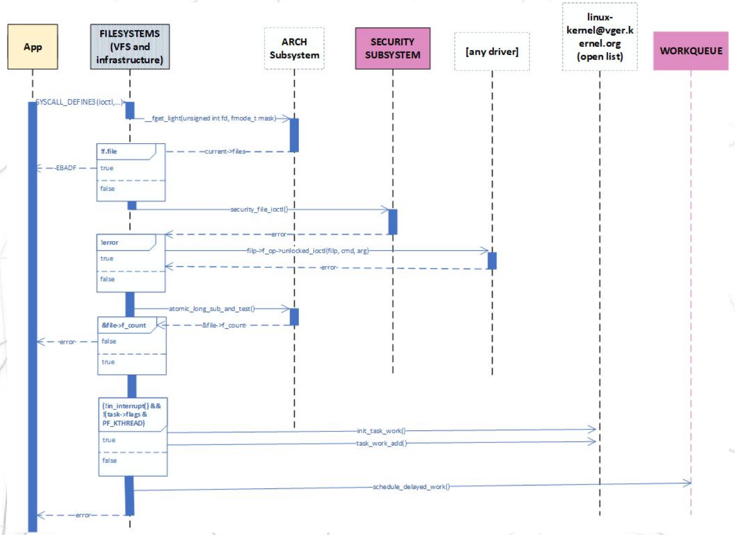


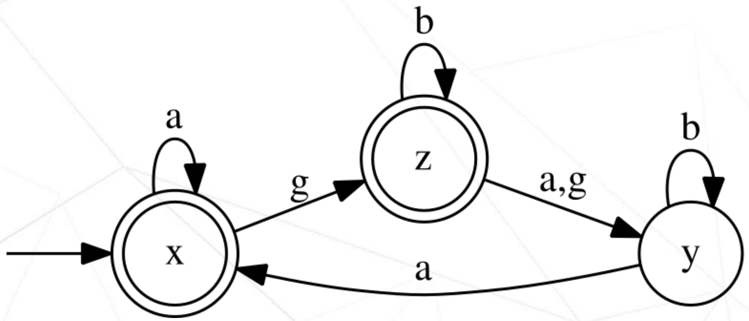
RV in the approach: why do we care?

- 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



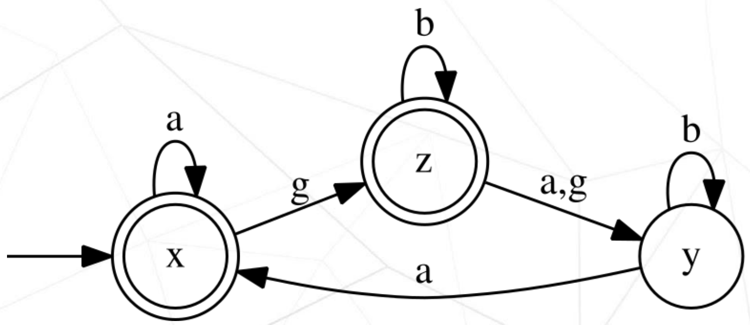
The tailored architectural approach and Runtime Verification


App
FILESYSTEMS (VFS and infrastructure)
ARCH Subsystem
SECURITY SUBSYSTEM
[any driver]
linux-kernel@vger.kernel.org (open list)
WORKQUEUE
SYSCALL_DEFINE3(ioctl,...)
__fget_light(unsigned int fd, fmode_t mask)
!f.file
current->files
-EBADF
true
false
security_file_ioctl()
error
!error
filp->f_op->unlocked_ioctl(filp, cmd, arg)
true
error
false
atomic_long_sub_and_test()
&file->f_count
&file->f_count
error
false
true
(!in_interrupt() && !(task->flags & PF_KTHREAD)
init_task_work()
task_work_add()
true
false
schedule_delayed_work()
error



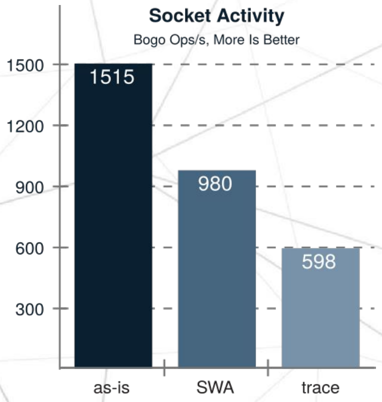
Automata based Runtime Verification

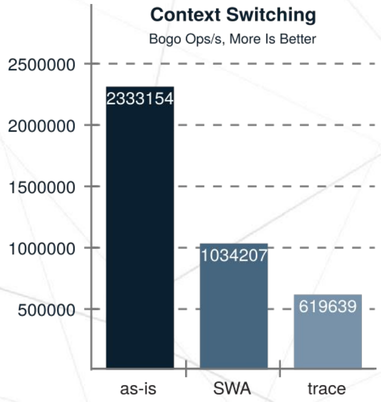
- Over the last years, a RV method using automata theory has been refined
- Automata is flexible, intuitive and can be used to specify complex parts of the system:
  - See paper: **A Thread Synchronization Model for the PREEMPT_RT Linux Kernel** (+9k states, +21k transitions)
  - Build from small specifications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

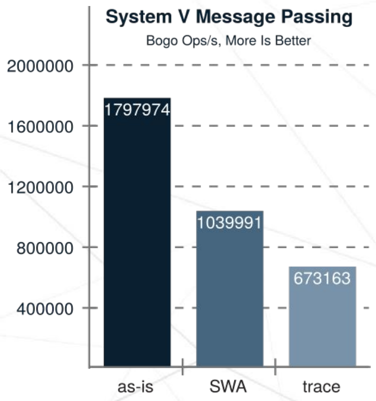

$$\begin{tikzcd} \arrow[r] & x \arrow[loop above, "a"] \arrow[r, "g"] & z \arrow[loop above, "b"] \arrow[r, "{a,g}"] & y \arrow[loop above, "b"] \arrow[ll, bend left, "a"] \end{tikzcd}$$

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**







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

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/

LINUX PLUMBERS CONFERENCE September 20-24, 2021

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
```

```
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)
```

- Developer can watch the monitor via ftrace

For further information

- 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

Pain Points and Next Steps

Pain Points

- Communication diagrams between subsystems/drivers can be supported by static analysis tools of the code (TODO: add call-tree link)
- Dynamic diagrams baseline can be generate by tracing the interfaces between subsystems once the communication diagram is complete (cat sys/kernel/tracing/trace)
- Can an Automata starting baseline be generated out of tracepoints and static graphs
- **IMPORTANT: human review and refinement of the automata starting baseline is mandatory !!!!!!!**
- Kernel-doc headers must be written mandatorily for the interfaces between subsystems

Next Steps

- 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

Questions?