NVIDIA

A Systematic Approach to Using the Linux Kernel in a Safety Scenario

Igor Stoppa <istoppa@nvidia.com>

ELISA Workshop October 17th, 2023

Agenda

- Expectations
- The Problem
- ELISA: Limitations and improvements
- The need for completeness
- A possible solution, and why it’s needed
- Examples
- Call to action

Expectations

What will i get out of this?

What you WILL NOT get:

- Solution that makes Linux safer
- Argumentations that show Linux to be safe

What you WILL get:

- **A way to expose latent safety issues**
- **A way to categorise and proritise safety issues**

The Problem

The Problem

Strong Demand for using Linux in Safety Application

HOWEVER

Linux is not designed for Safety

How to use Linux in Safe Applications?

ELISA

ELISA: Limitations and constraints

- Linux evolves rapidly
- Many different customisations with different HW

IT IS NOT POSSIBLE TO GIVE GENERIC GUARANTEES ON LINUX SAFETY

What CAN be done?

- The Linux core architecture is fairly stable
- Most safety issues lie within the core design
- Anybody using Linux will have to address them

We can identify common safety issues related to using Linux

Motto:

We cannot help you with proving that Linux is safe

But we can help you identifying core safety issues

Do we really need to identify these issues?

A Common Pitfall:

Thinking the “Proven in Use” argument to be sufficient

But it works only in very narrow cases.

Can’t I make Proven In Use claims? Linux is everywhere

“Proven In use” requires the following:

- **Large** fleet of **specific** HW/SW combination
- **Large** amount of **historical** data **collected** from said fleet (e.g. MTBF)
- **Proof** that the **new** HW/SW combination is **equivalent** to the **historical** one
- **Proof** that the **new** safety scenarios are **equivalent** to the **historical** ones

In practice:

- HW evolves rapidly - using historical HW would be infeasible most of the time
- There is no such a thing as “Linux”, there are **many** “Linux releases”
  - using historical SW would mean rolling back many years of progress
  - the monolithic nature of the kernel would **prevent** any form of **partitioning**
- Even preserving HW & kernel, a change in the workloads might **trigger pre-existing latent causes** of interferences, that would **void** the “proven in use” argument.

At most, “Proven in Use” can be applied to a very small niche of situations

But it must count for something, that Linux is everywhere!

INDEED: ISO PAS 8926 for example

“Qualification of pre-existing software products for safety-related applications”

- Enables using SW that was not designed with safety requirements
- Not as strict as the requirements for Proven in Use
- Safety analysis STILL needed
- Freedom From Interference must STILL be proven

The PAS 8926 alone is not sufficient for safety claims

A Common Pitfall:

Thinking the Top-Down Analysis to be sufficient

But it can miss key interference scenarios

Example: Top-Down STPA analysis of the Linux kernel

- The Linux kernel is very complex
- STPA(*) allows for making simplifications (Exploratory Analysis)
- How to NOT MISS safety-critical aspects?

One should first analyse EVERYTHING, then simplify

(*)System-Theoretic Process Analysis Handbook: https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf

Example: STPA missing key safety issues

- STPA uses top-down analysis
- The Linux kernel is highly parallelised
- Certain safety issues are **buried deep down** various subsystems, in performance optimisations.
- Critical issues within the Linux infrastructure might **not be visible** through the STPA.

Critical Spatial Interference can go unnoticed

(few major examples will follow)

Back to the Solution

What CAN be done in practice?

- Perform Low Level Inductive Analysis on core Linux components.
- Identify dependency between Linux subsystems
- Create a Prioritised Checklist of known issues

What comprises the Checklist?

- Effects of specific failures in low level components
- Analysis of low level components dependencies
- Correlations between failures in low-level components (cascading)

Why a checklist?

- Low Level safety issues are known **a priori**
- Ensure they are **not missed** (like with STPA)
- Perhaps a **shared methodology** can emerge, once the low level problems are formalised

Advantages

- Pool of causes for possible failures
- Define criteria for evaluating failures
- Standardise the evaluation of core issues

BUT evaluation and mitigations are still going to be application-specific

Actual Content of the Checklist

Example of Low Level Items

- Memory Integrity & Memory Allocation
- Call Stacks
- IPCs
- I/O
- Scheduling
- Selected device drivers for each subsystem
  (e.g. drivers supported by QEMU):
  Storage, Networking, Graphics, etc.

Start with most common items and progressively expand the scope to other items.

Long Term Goal

Establish a

shared methodology

for the evaluation of safety-relevant faults
and related mitigations

Use of the Checklist

Intended Users of the Checklist

- Newcomers to Linux for Safety
- Entities looking for a streamlined approach:
  - Customers
  - Vendors
  - Assessors

Checklist vs Functional Safety Requirements

Customising the Checklist

- Each scenario has its own specific requirements.
- Requirements on integrity, availability, latency affect which items of the Checklist are relevant.

Stakeholders assess which Checklist items affect their analysis, based on Safety scenario

Checklist vs Analysis of Safety Scenarios

- NOT a substitute for analysing safety scenarios (e.g. STPA)

- Instead, complement and gauge the simplifications made during the analysis.

- **Did the Analysis of Safety Scenarios miss something from the Checklist?**
- **Can the missing parts be addressed separately?**

Checklist vs Safety Case

Validation of safety mitigations for:

- Structural flaws
- Completeness (no loose ends)

- **Do the mitigations clear all the selected items in the Checklist?**

- **Are mitigations free from cascading issues?**

Must all the items of the Checklist be solved?

It depends.

- Some items might not apply to certain use cases.
- Some items might not affect certain use cases.
- Some items might require mitigations.

But EACH items MUST be addressed, somehow.

Even if only to say that it does not apply,
or that no mitigation is deployed.

Easy starting point: Spatial Interference

Why is Spatial Interference such a big problem?

- Linux is a monolithic kernel
- **No barriers** to intra-component interference

Anything can interfere with anything else (that is not write protected)

Practical examples of why it’s needed
(and why using STPA alone is not sufficient)

Example of direct kernel->kernel spatial interference

Spatial Interference: The userspace misconception

Misconceptions:

- The Kernel cannot trash user-space memory directly
- User-space drivers are safe from kernel interference

Facts (Tested On ARM64, should be also on x86_64):

- (Most of) The physical memory is mapped as writable in kernel
- Userspace mapping protections are irrelevant to the kernel mappings
- The kernel can alter any physical page mapped to any user-space
- No existing HW/SW configuration can currently prevent it, short of moving user-space to an enclave (e.g. ARM TrustZone)
- Using an Hypervisor would not improve anything, as long as userspace is still exposed to its underlying linux kernel

Spatial Interference: Kernel can corrupt user-space

Simulating intra-kernel interference that affects user-space

Altering process read-only memory through existing kernel mappings:

(“**help**” is an internal bash command)

Change process memory: “GNU bash” -> “KNU bash”

```
root@(none):/# help
GNU bash, version 5.2.15(1)-release (aarch64-unknown-linux-gnu)
```

Initial memory content

Simulate kernel -> kernel interference, writing to the linear map

```
root@(none):/# help
KNU bash, version 5.2.15(1)-release (aarch64-unknown-linux-gnu)
```

Altered memory content

Kernel initiated interference: implementation

```c
const volatile char *kstrnstr(const volatile void *mem_range,
                    const void *substring, size_t range_size) {
    size_t substring_len = strlen(substring);
    const volatile char *end =
            ((const char *)mem_range) + range_size - substring_len;

    for (const volatile char *ptr = mem_range; ptr <= end; ptr++) {
        if (memcmp((const void *)ptr, substring, substring_len) == 0) {
            return ptr;
        }
    }

    return NULL;  // substring not found within memory range
}
```

```c
static void interfere_with_bash(void)
{
        const char bash_string[] = "GNU bash";
        unsigned long pfn;
        const volatile char *p;
        int count = 0;
        int slp = 0;
        for (pfn = 0; pfn < max_pfn; ++pfn) {
                if (!pfn_valid(pfn))
                        continue;
                p = kstrnstr(page_to_virt(pfn_to_page(pfn)),
                             bash_string, PAGE_SIZE);
                if (slp++ == 1000) {
                        slp = 0;
                        msleep(10);
                }
                if (p && (p != bash_string)) {
                        count++;
                        pr_err("pre  count: %d  %s\n", count, p);
                        *((volatile char *)p) = 'K'; // XXX This
                        smp_mb();
                        pr_err("post count: %d  %s\n", count, p);
                }
        }
}
```

Steps:

- Iterate over all the physical pages (mapped by default in kernel space)
- Look for target string “GNU bash”
- When found, change to “KNU bash”

Invoking bash help will show the altered function

No userspace mappings were involved

As likely to happen as any other kernel interference

(Simplification: Doesn’t account for process paging out)

Example of spatial interference through the memory managers

Checklist Example: Memory Managers Interference

Misconceptions:

- Kernel memory managers can be treated as safe
- Process memory can be reserved and protected

Facts:

- All Memory managers use memory for own meta-data
- Meta-data is exposed to interference
- Corrupted metadata can cascade into re-using memory already allocated for safety-relevant processes

Both old and new allocations CANNOT be trusted to be and stay safe

Example of Containers as an insufficient FFI mechanism

Is Containers-based FFI good enough?

(Containers are a user-space construct based mostly on cgroups)

Misconceptions:

- Cgroups (Containers) are sufficient to satisfy safety requirements about allocating and guaranteeing resources for safety-critical processes

Facts:

- Cgroups implementation is very intertwined with core kernel functionality
- Cgroups pulls in large amount of non-safety-qualified code that gets executed very frequently
- Cgroups is exposed to intra-kernel interference

Intra-kernel interference (see KNU) still happens inside containers

Problem: additional non-qualified code is executed more frequently

Example of SELinux as an insufficient FFI mechanism

Is SELinux-based FFI good enough?

Misconceptions:

- SELinux is sufficient to enforce safety requirements about access control and shielding from interference

Facts:

- SELinux hooked into almost any userspace event (Security Module)
- SELinux can generate a lot of churning regarding memory allocations for metadata (more chances for interference through memory managers).
- SELinux pulls in large amount of non-qualified code that gets executed very frequently and can perform high-frequency unsafe memory allocations and releases
- SELinux is exposed to intra kernel interference

Intra-kernel interference (see KNU) still affects anything protected by SELinux

Problem: additional non-qualified code is executed more frequently

Wrapping it Up

Bringing it all together

Checklist (Low Level Inductive Analysis):

- What is the safety argumentation for intra-kernel interference?
- What is the safety argumentation for memory interference?
- What is the safety argumentation for interference to processes?
- …

STPA (Exploratory Analysis):

- What are the safety-related components?
- What safety requirements are allocated to which components?
- …

Safety Mitigations and Argumentations:

- What are the safety aspects to consider, based on requirements?
- Did the STPA touch all the known issues from the checklist?
- What mitigations are necessary?
- Do they cope with the additional failure modes from the checklist?

Conclusion: A Two-Pronged Approach

1. **Checklist(Low Level Inductive Analysis), to cover basic core issues that are not use-case specific**
   Solve FIRST the fundamental Safety problems

2. **STPA Top-Down Analysis, to not miss the big picture**
   Make controlled and justifiable simplifications, based on the PREVIOUS point

Only COMPLETENESS of the analysys can makes the safety claims credible

Call to Contribute

The Checklist needs to be populated:

- **Common failures need to be identified**
- **Effects need to be analysed**

Come join the effort!

In practice

- Define a location/repository
- Define a process for contributing
  - Submission template
  - Review/Acceptance criteria
- The Checklist needs to be populated:
  - Common failures need to be identified
  - Effects need to be analysed

Seeds for the Checklist

- **Linear-map based interference**
- **Interferences through memory managers**
- **Call-Stack corruption**
- **Side effects of cgroups and SELinux**

That’s All Folks!

THANK YOU!

Licensing of Workshop Results

All work created during the workshop is licensed under Creative Commons Attribution 4.0 International (CC-BY-4.0) [https://creativecommons.org/licenses/by/4.0/] by default, or under another suitable open-source license, e.g., GPL-2.0 for kernel code contributions.

You are free to:

- **Share — copy and redistribute the material in any medium or format**
- **Adapt — remix, transform, and build upon the material for any purpose, even commercially.**

The licensor cannot revoke these freedoms as long as you follow the license terms.

Under the following terms:

Attribution — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.

No additional restrictions — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.

NVIDIA®