Safety of Software-Intensive Systems from First Principles

Authors: Paul Albertella and Paul Sherwood

1 Safety of Software-Intensive Systems from First Principles

Over the last several decades, the scale and complexity of critical software has increased by orders of magnitude. In cars for example, we have seen a tremendous growth from simple microcontrollers running a few thousand lines of code, to multi-function ECUs, advanced infotainment and driver assist capabilities; systems in a typical vehicle may involve over 100 million lines of code.

The international standards that are relevant to safety were mainly established long before this massive expansion took place. IEC 61508 and ISO 26262, for example, mainly describe approaches that are viable for less complex systems based on microcontroller architectures, but difficult or impossible to apply for modern systems involving multi-core microprocessors.

Many software and safety professionals recognise this challenge, and teams around the world are exploring how to assure safety in highly complex systems, by extending and improving upon existing methods, or by devising new ones.

We believe that the key problems can be summarised as follows:

- It is not feasible to consider safety of software independent of the system context in which the software operates. The “Safety Element out of Context” approach is fundamentally inapplicable for systems including complex software.
- Modern systems run so much software that rigorous analysis of all functionality and behaviours, while theoretically possible, is not achievable in practice.
- Software and systems are evolving so quickly that traditional safety approaches can not keep up.
- The behaviour of modern microprocessors, and many modern algorithms (e.g. AI/ML), is not deterministic in any case.

In spite of these problems, many international-scale organisations have achieved highly reliable and critical services based on complex software, and we aim to learn from approaches taken by other disciplines.

Thesis

We propose the following reasoning about safety for software-intensive systems, starting from the perspective of what we know about real software on real systems:

1. There is a requirement for safety-related promises about the behaviour of systems running complex software on multi-core microprocessors, including pre-existing open source software such as Linux.

2. When considering any typical modern system involving multi-core microprocessor and a range of inputs and outputs (e.g. IoT Device, Edge Device, Connected Vehicle):
    - The system will include at least hundreds of thousands of lines of code, since just the firmware bundled with modern processors is typically at that scale.
    - Counting also the software for OS and applications, typical systems have 10-100 million lines of code.
    - Most of the code (in a Linux-scale system, perhaps more than 99%) is not present to deliver safety-relevant functionality.
    - Most of the code is present to provide either generic functionality (e.g. non safety related communications, graphics, data processing) which may not even be activated for the target system, or system-specific functionality that is not directly relevant to the safety case.
    - Most of the code does not have associated design documentation or requirements definitions that are specific to the expected use cases for the entire system.

3. When considering modern multi-core systems involving functionality or algorithms relying on randomisation, probabilities or AI/ML models:
    - expected behaviour can not be considered deterministic, by definition
    - validation of behaviour can only be achieved up to a statistical measure of the reliability of the result
    - any validation of behaviour must be considered in the context of the provided input or training datasets.

4. Therefore the behaviour of complex software such as Linux running on multi-core microprocessors is not deterministic. However, over many operations we may establish a confidence level for the presence of specific required behaviours, or the absence of specific misbehaviours

5. **But** software behaviour can be trivially and completely changed
    - by modifying one line of code; this is true no matter how complex the software, e.g.
        - insert `exit()` before any required behaviour.
        - insert `not` in any if statement.
    - by modifying other code outside the target software, e.g:
        - adjust the compiler to interpret specific instructions differently
        - adjust the boot loader to configure resources differently
    - by modifying the hardware running the software, e.g:
        - make resources unavailable
        - adjust clock speeds

6. In addition, modern system software evolves, changing many times over the lifetime of the system. As a result, promises must be re-evaluated at every change.

7. From the above it follows that any promise may not be satisfied from time to time, potentially as a result of software changes, hardware failures and/or environmental factors.

8. To minimise the impact of promises not being satisfied, we propose the following strategy:

    - Analyse the sources of disturbance which can lead to promises not being satisfied[1]
    - Demonstrate reliable construction of known versions of the software[2]
    - Provide stochastic tests to check that known versions of the software provide the promised behaviour and do not provide misbehaviour.
    - Measure and analyse test results to establish confidence level
    - Construct alternate versions of the software with simple patches applied, which do not provide the required behaviours, and demonstrate misbehaviours[3]
    - Use both normal and fault-injection software versions to test overall system behaviours, both in normal running and under duress with promises broken
    - Provide guidelines, scripts and tests to support users/developers of the software based on the results of the analysis and testing process
    - Maintain all of this in a continuous integration framework, with ongoing analysis and the objective of improving confidence levels.

[1] Using Systems Theoretic Process Analysis (STPA)
[2] By establishing binary, bit-for-bit reproducible builds and by avoiding unnecessary dependencies
[3] Fault Injection

2 Software safety: a system-theoretic perspective

This section summarises our understanding of safety topics, which has informed the preceding thesis and the new approach to software safety described in chapter 3.

Functional safety

Discussions about safety in the context of software most commonly focus on **functional safety** as it applies to electronic and electro-mechanical systems. This is a set of engineering practices that seek to reduce the level of risk in a device or system to an acceptable level, where the definition of 'acceptable' is determined by the nature of the system, its intended purpose and the type of risks that are involved.

As summarised[4] by the International Electrotechnical Commission (IEC):

> "Functional safety identifies potentially dangerous conditions that could result in harm and automatically enables corrective actions to avoid or reduce the impact of an incident. It is part of the overall safety of a system or device that depends on automatic safeguards responding to a hazardous event."

Functional safety engineering practices, and internationally-recognised standards such as IEC 61508 that formally describe them, focus on identifying the failures that lead to accidents and other hazardous events, and specifying how the system as a whole - and features identified as 'safety functions' in particular - can detect and respond to these, in order to avoid or minimise harmful consequences.

One of these practices is **hazard and risk analysis**, which is used to identify and examine the conditions that can lead to harmful outcomes (*hazards*), and to evaluate both the probability of these conditions occurring and the severity of the potential consequences (*risks*). This kind of analysis will always be informed by previous problems or accidents, and the known limitations of hardware or software components, but it also uses formal techniques to try to identify new problems.

It's important to note that functional safety practices are not expected to eliminate all potential hazards, or to reduce the identified risks to zero, but only to reduce those risks to an acceptable level. A key role for the safety standards is in qualifying what is meant by acceptable in a given context, and describing what can be considered *sufficient* when evaluating how those risks have been mitigated.

[4] https://www.iec.ch/safety

Safety objectives

To understand what this means, it is useful to distinguish between four broad objectives in safety, which safety practitioners can characterise using the following questions:

1. What does ‘safe” mean, and how can this desired state be achieved and maintained?
2. How can the *safety measures* identified in #1 be realised or implemented?
3. How can confidence be established that the criteria in #1 and the measures in #2 are *sufficient*?
4. How can confidence be established that the *processes* and *tools* that we use to achieve #1, #2 and #3 are *sufficient*?

It is useful to make these distinctions when reading about safety, because a proposition or a technique that might apply to one of these objectives will not necessarily be helpful when applied to another.

Safety standards such as IEC 61508 and ISO 26262 are primarily concerned with objectives #2, #3 and #4, but they also list techniques for elaborating #1, and identify some general criteria that are typically applied for this objective. Standards for other domains, such as those covering the manufacture of toys, for example, may contain more concrete examples of #1.

Standards such as ISO 26262 permit safety practitioners to break down (decompose) *systems* into *components*, and to examine the distinct roles that software and hardware components play in the safety of a system. They also encourage practitioners to consider the *tools* that are used to create or refine these components, to examine how they contribute to objectives #2 and #3, and to consider how objective #4 applies to them.

Part of the motivation here is the desire to have reusable components and tools, with clearly-defined and widely-applicable safety-relevant characteristics, which practitioners can feel confident about using for different contexts and systems. There is explicit support for this at a system component level in ISO 26262 (the functional safety standard for road vehicles), in the form of the 'Safety Element out of Context" (*SEooC*) concept.

System vs component

The "Safety is a system property" assertion is associated with a school of thought that might be labelled the 'system theoretic safety perspective', which has been notably articulated by Dr. Nancy Leveson, who observed in *Engineering a Safer World: Systems Thinking Applied to Safety* [5]:

> "Because safety is an emergent property, it is not possible to take a single system component, like a software module or a single human action, in isolation and assess its safety. A component that is perfectly safe in one system or in one environment may not be when used in another."

This might at first seem to be incompatible with a desire for reusable components. In our opinion, however, this does not mean that safety tasks can only be undertaken in reference to a complete and specific system. Rather, it asserts that it is impossible to perform a complete safety analysis for a given component without also considering the wider context of that component.

Furthermore, this perspective applies primarily to objective #1 (although it also has some bearing on #3), which means that there are some tools and techniques relating to safety whose merits can be examined without explicit reference to a *system context*.

When trying to define what is meant by safe, and considering the completeness of that definition, the system context cannot be ignored. For a software *component* in particular, this means answering the following questions:

- What *system* (or kind of system) is under consideration?
- What is this *system*'s intended purpose, and how does the software contribute to it?
- What *environment* (or kinds of environment) is the *system* intended to operate within, and how might this affect the software *component*?
- What does 'safe' mean in the context of this *system*, and how is the software *component* relevant to achieving or maintaining this desired state?

The *system context* under consideration might be concrete and complete (e.g. a specific integration of hardware and software as part of a product), or partial (e.g. only describing how the component interacts with the the rest of the system) or an abstraction (e.g. a set of assumptions and constraints for a given category of system). Without defining a context, it is not possible to make meaningful statements about what may be considered safe, and there is no way of evaluating the completeness (i.e. whether it is *sufficient*) of the analysis.

[5] https://direct.mit.edu/books/book/2908/Engineering-a-Safer-WorldSystems-Thinking-Applied

Of course, it is possible to consider a given piece of software in isolation, examining how its properties might contribute to the safety of a system, or represent a threat to its safe operation. But without at least an implied *system context* - and an understanding of what 'safe' means in that context - it is potentially misleading to label the software, or its properties, as 'safe'.

Hence, when safety is a necessary factor to consider, attempts should always be made to answer these questions, even if the answers are unsatisfactory or provisional (e.g. "We have not considered the impact of any environmental factors on the operation of the system"). If the context in which the software will operate has not been considered or described - or if reasoning is based upon an assumed context that has not been described - then gaps or flaws in that reasoning may go unnoticed, with possibly dangerous consequences.

Furthermore, when considering an *abstract context* - a *system context* that is partial, provisional, or conditional upon a set of assumptions or requirements - then any claims made about safety must be equally partial, provisional or conditional. As a minimum, these safety claims will need to be re-validated when the software is used in a real-world product or application (a *concrete context*), to ensure that the stated conditions apply. To have confidence in such claims, however, the analysis that underpins these claims will also need to be repeated or re-validated, to identify new hazards, and to consider whether the risks associated with existing hazards are altered by the new context.

The value of performing a full safety process for a *component* using such an *abstract context*, as with an *SEooC*, is thus questionable. Performing hazard and risk analysis in such a context can certainly be valuable, by identifying and characterising the *hazards* that may apply for a defined class of *systems*, and develop *safety measures* that can be used to mitigate them. However, a safety practitioner cannot be completely confident that either the analysis or the measures are *sufficient* until they have been considered for a *concrete context*. If a claim is made that they are *sufficient* for the *abstract context*, then there is a risk that the necessary analysis and re-validation will be omitted when they are used in a *concrete context*.

Failures and hazards

The importance of considering a wider system context is only one aspect of the system theoretic safety perspective. Equally important is the observation that *hazards*, particularly in complex systems, can manifest even when all of the *components* in a system are performing their specified function correctly.

Functional safety practices have tended to focus on reducing the *risk* of a hazardous outcome in the event of a *failure*, by which we mean the manifestation of a fault: something that prevents a component or system from fulfilling its intended purpose. Commonly used hazard and risk analysis techniques, such as Failure Mode and Effect Analysis (FMEA) and Fault Tree Analysis (FTA), involve systematically examining the ways in which components can fail in order to understand the effect of this on the system. The probability of these failures occurring is then calculated and used to evaluate the associated *risks* of a hazard, which are then used to identify where *safety measures* - activities or technical solutions to detect, prevent or respond to failures - are required.

When the *root cause* of such *failures* is known (e.g. a hardware component with specified physical limitations or need for periodic maintenance), then this kind of analysis can be an effective way to mitigate the consequences of a failure, or to identify how the consequences of a failure can cascade through the system in a 'chain of events' to cause an accident. This enables fault-tolerant systems to be developed, enabling them to remain safe even when one or more components fail. However, this analytical approach has acknowledged limitations when used in isolation, and for identifying multi-factorial systemic failures.

Part of the reason for this is the focus on *failures* and the event-chain model. When examining accidents from a different perspective, it becomes apparent that accidents can occur even in the absence of any *failure*, due to unforeseen interactions between components or environmental conditions (external factors that affect the system state), or a combination of factors that may involve several *components*. Furthermore, these factors may be external to the *system* or *components* as specified, most notably when human interaction contributes to a hazard.

To identify this kind of hazard, application of a broader perspective is required, examining not just how individual *components* or elements of the system may fail, but how their interactions, and the influence of external factors that can affect their state (its *environment*), may combine to cause undesirable outcomes.

At the simplest level, this perspective can be applied by re-framing safety objective #1:

- What undesirable outcomes (*losses*) can occur?
- What sets of conditions (*hazards*) can lead to a loss?
- What criteria (*constraints*) must be satisfied in order to avoid these hazards?

Having identified *hazards* in this way, safety goals and requirements may then be described in terms of the *constraints* that must apply to the behaviour of the *system* in order to avoid or minimise the impact of these *hazards*.

STPA

This is the approach taken by **System-Theoretic Process Analysis (STPA)**, a hazard analysis technique based on a relatively new accident model (STAMP: Systems-Theoretic Accident Modeling and Processes), which was developed by Dr. Nancy Leveson of MIT in direct response to the perceived limitations of existing event-chain models. These include Failure Mode and Effects Analysis (FMEA) and Fault Tree Analysis (FTA).

Comparisons of STPA and FMEA[6], and of both with FTA[7], examining their relative ease of use and effectiveness at identifying unique hazards, found that they "deliver similar analysis results" and have their own strengths and weaknesses; a more recent paper[8] explored the potential benefits of combining STPA and FMEA to address some of these weaknesses. However, the merits of applying systems thinking when applied to safety go beyond hazard analysis.

STPA's approach to modelling control structures equips safety practitioners with an analytical technique that is flexible enough to include factors ranging from signals exchanged between microprocessors to the impact of new legislation on the organisation producing them. While this can open up a breadth of analysis that may seem counter-productive, when applied correctly the method ensures the correct focus, by explicitly defining the scope of analysis at the outset.

The most immediate application of this technique is as a hazard and risk analysis technique, but it also provides a framework for developing reusable and extensible safety requirements. These can be iteratively developed and refined at different levels of abstraction and examine different levels of the control hierarchy as the system is developed. The technique can also be applied at a much earlier stage in the safety and development lifecycles, and remains useful throughout. It can also be used to analyse the processes involved in these lifecycles, to identify how measures intended to increase safety might best mitigate risks introduced via these processes (e.g. applying a security patch to a piece of software).

STPA facilitates a progression from identifying *hazards*, through defining *constraints*, to validating design and verification measures, and identifying the causes of an issue during integration and testing. If used in this wider mode, the technique has the potential to deliver far greater benefits, complemented by classic bottom-up hazard and risk analysis techniques (where appropriate) rather than replacing them.

[6] https://link.springer.com/article/10.1007/s11219-017-9396-0
[7] https://arxiv.org/abs/1612.00330
[8] https://www.mdpi.com/2076-3417/10/21/7400

Conclusions

Complex software is playing an ever-increasing role in systems where safety is a critical consideration, notably in vehicles that include advanced driver-assistance systems (ADAS) and in the development of autonomous driving capabilities, but also in medical applications, civil infrastructure and industrial automation.

When thinking about safety as it applies to software, it is necessary to consider the system and the wider context within which that software is used. This is because hazards are an emergent property of this *system context*: they cannot be fully understood in the context of the software *component* alone. Furthermore, hazards do not only occur when a component fails: they may result from unanticipated interactions between components, or the influence of external conditions (including human interactions), even when all of the components involved are working correctly.

This system-level perspective is often missing from conventional functional safety practices and the associated standards, because they focus on hazards that arise from component failures. ISO 26262, for example, defines functional safety as *"absence of unreasonable risk due to hazards caused by malfunctioning behaviour of E/E systems."* The limitations of this perspective are explicitly recognised in the more recent ISO 21448[9], which notes that:

> "For some systems, which rely on sensing the internal or external environment, there can be potentially hazardous behaviour caused by the intended functionality or performance limitation of a system that is free from faults addressed in the ISO 26262 series."

However, while this standard is a step in the right direction, and specifically mentions STPA as a technique for identifying hazards that arise from "usage of the system in a way not intended by the manufacturer if the system", it stops short of recommending this kind of analysis as a matter of course for systems involving complex software.

One explanation for this omission may be found in the historical origins of functional safety practices, which were largely developed in the context of electro-mechanical systems and relatively simple software components. In the automotive industry in particular, standardised functional safety practices are also explicitly built around components, reflecting the way that responsibility for safety is distributed throughout a vehicle manufacturer's supply chain.

Breaking down systems into discrete components is a familiar and necessary strategy in systems engineering, allowing different developers to work more efficiently, and promoting specialisation and re-use. However, because safety must be understood

[9] https://www.iso.org/obp/ui/#iso:std:iso:pas:21448:ed-1:v1:en

at a system level, it can be counter-productive to evaluate a component in isolation and then label it as 'safe', even for a tightly-specified use. This can lead to missed hazards, but it also means that the majority of safety engineering effort is expended on defining and validating a component in isolation, instead of examining its role in a wider, more concrete context.

As an alternative, top-down analytical techniques such as STPA can be used to identify and characterise *hazards* at both a *system* and a *component* level, and to analyse the *processes* used to develop them. Safety requirements are then derived from this analysis in the form of *constraints*, which can be iteratively developed at various levels of abstraction, or levels of a system-component hierarchy. By providing a common language to inform safety activities at all levels, these constraints can then be used to validate component behaviour and safety measures across the system, not only at the level of the component, or in an *abstract system context*.

In our opinion, this approach is not incompatible with the bottom-up, failure-focussed techniques that are prevalent in functional safety practices. Rather, by providing a way to reframe and refocus safety engineering efforts at the system level, it may ensure that those efforts are more effective at identifying and mitigating the hazards that slip through existing nets.

3 A new approach to software safety

In order to make safety-related promises about the behaviour of systems running complex software on multi-core microprocessors, including pre-existing open source software such as Linux, we believe that a new approach to software safety will be required, which:

- Can be applied to a huge body of pre-existing code, within a justifiable cost and time frame
- Can handle a very high rate of ongoing development and integration change
- Recognises the unique characteristics of modern software components in comparison with other system elements

The activities described below are expected to form part of an iterative process for a given system, with each iteration refining the system's specification and associated safety analysis, and the measures required to verify and validate it.

Define OS responsibilities

The first challenge is to specify, with sufficient clarity, what Linux needs to do as part of a system, and how these responsibilities are relevant to the safety-critical elements of that system.

For a general-purpose operating system component like the Linux kernel, which has evolved to support a very wide variety of hardware and applications, the potential scope of its responsibilities may be very large. However, not all of this functionality will necessarily be related to the system's safety-critical applications; depending on the nature of the responsibilities, it may be possible to limit the specification to a subset.

This task may still be non-trivial, however, because Linux, in common with most free and open source software (FOSS), lacks a formal specification of its design and the goals that inform it. A wealth of technical information is available, and anyone is free to examine its source code in order to understand more, or read kernel developer mailing lists to understand the decisions behind changes. However, the Linux development process does not include the formal documentation of requirements, architecture and design.

In part, this reflects the nature of Linux: it does not have a specific and formally circumscribed purpose, but is instead iteratively adapted, refined and extended to address an ever-expanding set of purposes and applications, by anyone who is willing to contribute. Linux is extremely configurable by design, which means that one system deployment including Linux may differ radically from another, even if they both share a common hardware platform.

In this first stage, therefore, it is essential to document both the specific system context in which we intend to use Linux, and the subset of functionality that it must provide in that context.

Define system context

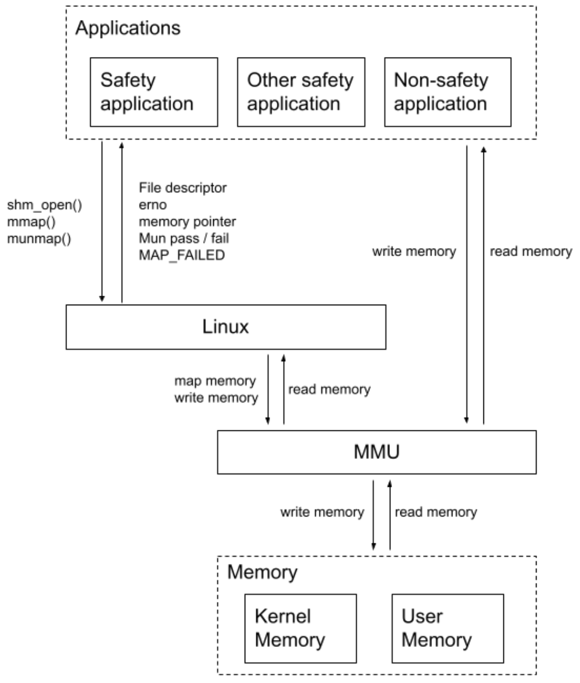
The system context for the OS must encompass all of the components that will, together with the Linux kernel, provide the operating system services. It must also include the kernel configuration used to construct the kernel, as this will determine which features are available, and how they behave. It must also include the system hardware components, most notably the CPU architecture and any components that will be used by the system's safety application(s).

Define OS functions

The next step is to define the functions that the OS (including Linux) will provide in this system context. It may be possible to limit this to a subset of the complete functionality provided by the OS, by focussing on the services that it provides to safety-critical applications as part of the system. However, the wider scope of OS functionality should also be defined if applications without a safety-related function are expected to run in parallel on the system, as it is necessary to consider how that functionality might impact the safety application(s)

Define control structure

The components of the system that are involved in these functions are then specified in *control structure diagrams*, which describe the system as a hierarchy of control feedback loops. The OS, and the components that it interacts with are *controllers* in this hierarchy, and their interactions are represented as *control actions* (down arrows) and *feedback* (up arrows). Controllers may represent hardware or software components, and may be an abstraction of multiple components or categories of components.



Example control structure diagram examining shared memory

Define safety requirements

The next challenge is to specify a set of detailed safety requirements for the OS, using STPA to perform a top-down hazard analysis of the system defined in the previous activity. For more information on STPA, see the preceding section and the STPA Handbook[10]

Define system-level losses and hazards

A critical first part of this analysis is to identify and understand hazards in the context of the wider system. How might failures or unintended behaviour of the system as a whole lead to losses? As discussed in the previous section, a component's responsibilities with regard to safety cannot be understood without examining its role for a particular system, because the losses to be prevented, and the hazards that may lead to them, will be specific to that system.

This stage also includes identification of *system-level constraints*, which are system conditions or behaviours that need to be satisfied to prevent hazards; they may also define how to minimise losses if hazards do occur. These are defined at the level of the overall system incorporating the OS, not the OS itself.

Identify potential hazards involving the OS

Hazard analysis is performed using the control structure diagrams created in the previous stage, as part of our definition of the system and the specific responsibilities of the OS.

The system conditions that can lead to system-level hazards may be specific to the OS (e.g. handling of hardware failures), but may also relate to the behaviour of other system components (e.g. a safety application's handling of failures reported by the OS). These causes, which STPA calls *unsafe control actions (UCAs)*, are identified by examining how interactions involving the OS may lead to the system-level hazards identified in the previous step. Further analysis of these produces *loss scenarios*, which describes the causal factors that can lead to the UCAs and the associated hazards.

This analysis should focus on the overall responsibilities of the OS and its interactions with other components at its boundaries (i.e. via kernel syscall interface or equivalent), not the internal logic of the OS. It may be necessary to examine some of that internal logic in order to form an understanding of the behaviour, but it is assumed that these internal details will not need to be referenced in UCAs in most cases.

Specify constraints required to prevent these hazards

Constraints are specific, unambiguous and verifiable criteria that are applied to the behaviour of either the system as a whole, or of a specific controller in the control system hierarchy, in order to prevent a hazard.

10 https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf

Controller constraints are derived directly from the UCAs and loss scenarios identified in the previous step.

Constraints may be implemented in many ways: design features of the OS, external safety mechanisms such as a hardware watchdog, offline verification measures applied during development or software integration (e.g. tests, static analysis rules) or online verification measures implemented by another software component (e.g. a monitoring process).

Note that these constraints are not confined to the OS. Some may already be fulfilled by aspects of the OS design; others may require the addition of a new feature as part of the OS, or an external safety mechanism. However, many may need to be applied as requirements on the development processes or design of safety applications, such as static analysis rules that must be applied when verifying application code, or system-level testing that must be performed to validate an application's use of the OS, or the behaviour of the hardware.

Historical and regression testing

To ensure that the OS provides the expected behaviour required by safety applications requires development of a set of tests. It may be possible to re-use or adapt existing tests developed for Linux for some of these.

Tests should be system-level wherever possible, using the external interfaces of the OS, as this is the most efficient way to verify behaviour on an ongoing basis, and ensures the long term relevance of tests. Because of the nature of Linux, the stability of internal component logic cannot be guaranteed, but the kernel development community has an explicit 'golden rule' ("Don't break userspace!") that should guarantee the stability of its external interfaces and associated behaviour.

Some functionality may require different or additional verification strategies, where a constraint cannot be verified at OS system level, but these should be the exception. An example might be a static analysis rule to check that kernel access to user memory follows the correct protocol. For more information, read about our earlier investigation[11] into this area.

Regression testing

Since the OS components will explicitly change over the lifetime of the system, *regression tests* are used to verify that current OS components satisfy the safety requirements. These will correspond directly to OS design constraints identified during hazard analysis: they exercise potentially unsafe behaviours and verify that the required behaviour is exhibited. Statistical measurement of test results (negative fault, negative detect) will be used to provide evidence to support a safety case; where tests fail, additional impact analysis will be required.

[11] https://www.codethink.co.uk/articles/2020/investigating-kernel-user-space-access/

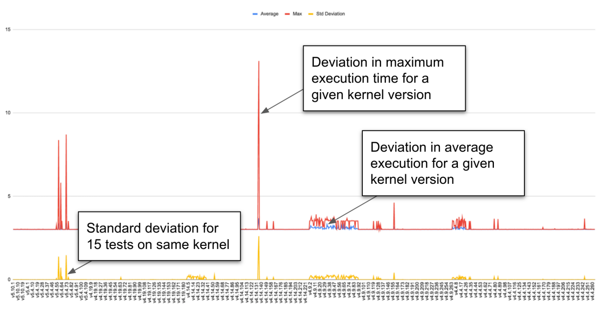
Historical testing

Because there is no formal functional specification for the OS, evidence that the expected behaviour has been present and stable over time must be constructed by repeating regression tests using historical versions of OS components (e.g. last 1000 released Linux kernel versions). Test failures will be investigated to help refine existing tests and analysis. These may reveal hazards that were not identified in original analysis, or provide examples to inform fault injection and stress testing (see below).

Statistical analysis and models

Statistical models provide increased confidence in tests, or establish a level of confidence in test coverage when complete certainty is not feasible. Examples include:

- Identifying a representative set of tests, where a comprehensive set is not possible or practical
- Using a stochastic model to show that non-deterministic behaviour falls within expected bounds
- Measurement of test results (negative fault, negative detect) and other test characteristics (e.g. execution time), to identify patterns and anomalies



Example of statistical analysis of historical test data

Confidence in test results can be increased by examining them from a different angle, and challenging assumptions in order to avoid false confidence. Simple pass/fail results may conceal an unidentified issue or hazard; examining other characteristics of tests, such as the execution time or CPU load, may help to reveal these, or to identify flaws in tests or test infrastructure that are distorting or concealing results.

Fault injection and stress testing

Simply confirming the expected functionality is not sufficient, however: in order to have confidence in the safety of a system, the test suite must validate system behaviour when things go wrong, or behaviour deviates from the expected path.

Fault injection

This is tested using *fault injection* strategies, constructing alternate versions of the software that deliberately violates the expected behaviour. The intention here is to provoke or simulate the hazards identified during analysis, in order to validate the safety measures developed to mitigate these hazards (e.g. confirm the effectiveness of tests). These are false positive checks: faults are introduced to validate they are detected or mitigated; faults are cleared to confirm they are no longer reported.

This strategy is intended for use with a complete system, not only for testing the OS in isolation. It can be used to validate safety mechanisms provided by external components, verify the handling of OS-related faults and other hazards by safety applications, and facilitate the validation of overall system safety measures in the final product.

Where possible, fault injection should be accomplished by simple code changes to software as deployed (e.g. kernel patches applied during construction); this means that the OS can be used in system testing without special preparation. Where appropriate, faults may also be injected via a test process (e.g. simulating interference by a ‘rogue’ process). In some cases, faults may need to be injected using custom kernel modules, and triggered via an `ioctl` interface, either randomly or via a test fixture. This approach should be the exception, but may be necessary for fundamental functionality, where a simple change would prevent system initialization.

OS fault-injection should be included as a matter of course in system-level smoke tests, perhaps by periodically using an OS image with arbitrarily selected fault(s). This serves to validate OS functional tests and uncover gaps in application or other system components.

Stress testing

Safety functionality may also be compromised by system load or interference from other applications running in parallel, using existing tools such as stress-ng[12] and targeted tests to simulate system conditions and/or interference from other processes. These tests will need to be tailored for a given system, as the nature and scope of other applications (both safety and non-safety) on a system will vary, but it should be feasible to define a generic set of tests, which can be extended and refined for specific systems.

Hazard analysis will help to inform these tests, by identifying specific system conditions or sources of interference. Results from historical testing and statistical analysis may also suggest further tests.

[12] https://wiki.ubuntu.com/Kernel/Reference/stress-ng

Deterministic construction

Construction is the foundation for the key engineering processes that underpin all of this analysis, verification and validation. This includes the tools, processes and inputs used to build and verify the system, the build and test environments in which these processes are executed, and the configuration and change management of these resources.

Relevance for safety

Predictable characteristics from construction support verification and impact analysis. Binary reproducibility of system artifacts and toolchain components enables cross-validation of these elements. A previously-validated system binary, which was output by the toolchain, can verify a new revision of that toolchain. If a system binary changes when no source, tool or configuration has been changed, this may indicate an uncontrolled configuration file or cached dependency. Binary reproducibility also permits verification and analysis of the impact of changes to the OS: if we compare output binaries with a previous version, and a source change has no effect on the output binary, then tests do not need to be repeated. See *Toolchain reproducibility* and *Production target reproducibility* below for more information.

A deterministic construction process enables impact analysis with very fine granularity. Using declarative construction definitions means that we have complete control over how components are constructed, and which inputs are used. This includes construction and verification tools and build dependencies as well as the system component source code. All of these inputs are managed in git repositories under direct change control; where these originate from 'upstream' open source projects, these repositories must be mirrored on infrastructure under local control, to ensure continuity of access and detection of anomalous changes.

An automated CI/CD process built on this foundation orchestrates safety analysis processes and the evidence required to support a safety case. This includes provenance for all inputs and evidence of impact analysis for changes, traceability from requirement to test to test results, and configuration management aligned around the CI/CD process. See *Controlled Process* below for more information.

Controlled Process

Processes and techniques for a controlled CI/CD process based around construction were originally devised by Codethink engineers as part of the Baserock open source project, which began in 2011. The approach has evolved since that time as we have applied it on a range of client projects and public-facing initiatives. The core ideas are:

- All production software is constructed via a CI/CD framework (e.g. GitLab CI)
- There is no alternative path to deliver software into production, so for example it is not possible for a developer to create a special version by hand.
- All input software code is captured in a SCM (e.g. Git) as source code, or input binaries if source code is not provided by the supplier.
- Input source code from upstream projects is mirrored into a local SCM via an automated process which blocks any attempt to re-write history of release branches.
- All configuration and build instructions describing how to construct the production software are also captured in SCM.
- The toolchain which is used to construct production software is itself constructed from captured source/binaries and configuration/build instructions, via the CI/CD framework.
- The toolchain is bootstrapped so as to be independent of any/all tools or libraries present on the host machines where the toolchain is executed
- The toolchain is constructed so that it is fully reproducible, i.e. when executing construction of a specific version of the toolchain multiple times via the CI/CD framework, exactly the same binary output is expected, bit-for-bit.
- The toolchain is constructed so that it generates fully reproducible outputs for inputs which support the property of reproducibility.
- Fully reproducible builds provide a viable mechanism of reconfirming that updated versions of the toolchain and surrounding framework continue to behave correctly with previous inputs.
- If a newly upgraded setup of the framework and toolchain generates exactly the same binaries as previous versions (for a comprehensive set of inputs), there is a strong basis for confidence that the new setup is functioning as expected.
- The framework is used to construct all components of the production software, including any additional tools and dependencies (e.g. Unix utilities, Python) required to achieve the final output binary files.

Toolchain reproducibility

A fully bootstrapped, sandboxed, reproducible construction toolchain allows the following:

- For a given version of the toolchain source code, the resulting toolchain is a specific fileset which is uniquely identified by hashing or checksumming.
- These filesets may be compared to a high degree of confidence by comparing hashes and checksums. To be completely certain about the

equivalence or differences between two filesets, one can do a bit-for-bit comparison or binary diff.
- Repeating construction of the toolchain generates exactly the same fileset when repeated on multiple computers, establishes confidence that the build process is isolated from all environmental factors and dependencies, and the toolchain is not reliant on any host specific tools or libraries. This avoids the "special server" problem where a project relies on a specific dedicated build machine, which becomes a single point of failure and risky to update.
- Under the same circumstances, any difference in the fileset indicates that there is indeterminate behaviour in construction of the toolchain, which requires investigation.

We assert that:

- All of the toolchain components should be reproducible by default.
- Any non-reproducible component to be introduced into the toolchain should be made reproducible before adoption.
- Any change which breaks reproducibility should be rejected.

Production target reproducibility

Fully reproducible builds of production target software allows the following:

- For a given version of all input source code and all the tools involved in construction, the resulting output is a specific fileset which is uniquely identified by hashing or checksumming.
- These filesets may be compared to a high degree of confidence by comparing hashes and checksums. To be completely certain about the equivalence or differences between two filesets, we can do a bit-for-bit comparison or "binary diff".
- When upgrading any or all of the construction tools, the generation of exactly the same fileset from that same input source code indicates confidence that the construction functionality of the upgraded tools is unchanged for this specific target and that there is no need to retest or revalidate the fileset as a result of the tools upgrade.
- Under the same circumstances, any difference in the fileset indicates that the upgraded tools function differently for this specific target and there is a need to retest and revalidate.
- When generating a new fileset, examination of the actual differences in binaries compared to a previous fileset may help in considering whether the differences are correct..

As an example, consider the introduction of an upgraded compiler which handles edge cases such as "dangling else" differently. If the binary output is unchanged, it indicates that our target is not affected by the "dangling else" change. Differences in output demonstrate that the target is affected by the changed compiler behaviour, requiring impact analysis.

Similarly, changes which relate to architectures other than that used for the target system, or unused language facilities, will not affect the fileset results.

Note that:

- Using the reproducibility property makes it easy to verify which tools affect the output fileset, and which do not
- SCM tools (e.g. git-bisect) can establish which specific changes in a new tool release actually affect the target.

We assert that:

- All production target software should be made reproducible by default
- Any non-reproducible component to be introduced into the target should be made reproducible before adoption
- Any change which breaks reproducibility should be rejected

Key assumptions

There are some key assumptions made in this proposed approach, which can only be validated by applying it to real examples:

- The majority of testing must be performed at the OS boundary. Specifying and verifying the functionality at a lower level would be impractical, due to its scale and complexity, and the continued evolution of the kernel by the Linux community, without a formal specification to direct it, makes this infeasible.
- The hazard analysis that is used to derive safety requirements must also be focussed at this level, for the same reasons.
- The proposed testing and fault injection strategies need to provide sufficient evidence of safety integrity, in lieu of the formal specification material that is conventionally required by safety standards.

4 Glossary

The following definitions, many of them borrowed directly from the STPA Handbook[13], clarify the meaning of some terms used in this white paper, which are highlighted in *italics* in chapter 2.

abstract context: A *system context* that is partial, provisional or conditional, where missing or unspecified aspects of the context are described using assumptions or requirements. This may be contrasted with a *concrete context*.

component: A discrete element or part of a system, or systems. A *component* in one frame of reference may be considered a *system* in another.

concrete context: A *system context* that corresponds to a real-world *system* (e.g. a product) with specified *components* and *environment*. This may be contrasted with an *abstract context*.

constraints: Unambiguous criteria pertaining to the operation of a *system*. Constraints are described using "must" or "must not" rather than "shall"; a distinction is made between requirements (system goals or mission) and constraints on how those goals can be achieved.

environment: An aspect of the *system context*, which may include any external factor that may have an effect upon it. Depending on the nature and boundaries of the *system*, this might be anything that is external to it: an aspect of the physical world (e.g. weather) for a sensor, or the CPU hardware for an operating system.

failure: The manifestation of a fault: something that prevents a *component* or *system* from fulfilling its intended purpose.

hazard: A *system* state or set of conditions that, together with a particular set of *environmental conditions*, will lead to a *loss*.

loss: An undesirable outcome associated with the operation of a *system*, involving something of value to its stakeholders (users, producers, customers, operators, etc).

process: A formalised set of practices that are undertaken as part of a development lifecycle.

risk: Describes the probability of an undesirable outcome (one that may lead to a *loss*) and the severity of the consequences.

safety measure: An activity or technical solution that is intended to prevent a *hazard*, reduce the probability of the associated *risk*, or minimise the severity of the consequences.

SEooC: Safety Element out of Context. In the ISO 26262 standard, this term is used to describe a *component* that is subjected to a safety certification process for an *abstract*

[13] https://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf

context. See the ISO 26262 definition[14] for more information.

sufficient: What is considered acceptable for a given domain or category of *systems* when considering what *safety measures* need to be undertaken to identify or mitigate *risks*, and what criteria these need to satisfy.

system: A set of *components* that act together as a whole to achieve some common goal, objective, or end. A system may contain subsystems and may also be part of a larger system. It may have both hardware and software components, and/or involve human interactions.

system context: A defined scope of analysis, which encompasses a *system* (or *component*), its intended purpose and the factors (including its *environment*) that may have a bearing upon that purpose. Some of these factors may be implied by the identified purpose (e.g. a car driven on public highways is subject to weather and traffic regulations).

tool: A software or hardware solution that is used as part of the development process for a *system* or *component*. If a tool is responsible for providing a *safety measure* (e.g. constructing or verifying a component), then it has a bearing on safety, even though it does not form part of the resultant *system* or *component*.

14 https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en

About Codethink

Codethink provides advanced technical consultancy and software engineering services to support international-scale organisations in a range of critical industries including Automotive, Medical Devices and FinTech. We collaborate in the open on a range of initiatives to advance the state-of-the-art for software engineering, including the CIP and ELISA projects which aim to improve the reliability and safety of open source infrastructure. If you would like to know more about our services please get in touch via **safety@codethink.co.uk**

www.codethink.co.uk

Follow us on Linkedin and Twitter