Qualification and Certification of Open-Source Software

Authors: Rainer Faller, *exida.com* – Excellence in Dependable Automation GmbH

Reviewers: Jonathan Moore, Prof. Nicholas Mc Guire, Dr. Giovanni Dallara, Arne Haas, Peter Müller, Mike Medoff, Dr. Robert Maier, Alexander Eggerer, Piotr Serwa

Version: 0.12; Mar. 1, 2026

Status: **Released for feedback**

Table of Contents

Qualification and Certification of Open-Source Software ***1***

1. Purpose of the Paper **2**

2. Opportunities and Challenges of assessing Open-Source Software **3**

2.1 Offer-driven versus Order-driven 3
2.2 Meritocracy and Gatekeeping in OSS versus Corporate Promotion 3
2.3 Change Management and Robustness by Feedback Loops from a large Community versus Top-Down V-Model Driven 4
2.4 Cybersecurity Focus of OSS 5
2.5 Requirements versus Behavior expected by the Community 6
2.6 Hierarchical Community-Driven Testing versus Descriptive Testing 6
2.7 Software Architecture documentation or the lack thereof 6

3. How *exida* assesses Open-Source projects **8**

3.1 Limited Feature Set and Software Element in Context Qualification 8
3.2 *exida* Assessment Justification Model for OSS (Safety Case and Layers of Protection) 8

Annex – Key OSS Considerations for Functional Safety Certification **13**

A.1 How do some FuSa standards ensure risk mitigation while promoting openness? 13
A.2 Strength and Weaknesses of OSS in the Light of Functional Safety 13

Literature and Terms **19**

Literature and Standards 19
Terms and Abbreviations 19
Versions 21
ToDo items for later releases of the paper 22

1. Purpose of the Paper

The openness of open-source software (OSS) development should be a good fit with functional-safety and cybersecurity. Furthermore, it is unlikely it will remain possible to continue to avoid the use of OSS in the future.

Open-source software (OSS) certification projects are extremely diverse, but all deviate from the V-model approach of classical safety-related software development, and even more so from the focus on work products and not technical information content introduced by ISO 26262.

exida welcome all comments and recommendations for including in future versions of this document. This explains the *exida* approach to assessing and certifying OSS used as part of Linux and/or GNU/Linux projects for functional-safety applications. It should contribute to the Open Certification philosophy of the author. In recent years, the practice of publishing comprehensive assessment reports has largely been replaced by the publication of certificates alone. For complex topics such as OSS qualification, greater transparency of assessment approaches would support technical discourse and mutual learning.

The *exida* approach considers the arguments and strength of the respective distributors (such as Red Hat, Codethink). It weights characteristic differences between OSS and proprietary development models and discusses how these can be addressed in qualification. The paper is intended for users of OSS, maintainers, assessors, and integrators involved in OSS safety qualification.

The class of systems targeted by the OSS Linux related approach in this document are characterised by

- Large-scale and high-complexity applications requiring a standardized (POSIX) operating system;
- Mixed-criticality up to medium level of required safety integrity (≤ SIL 2, ≤ ASIL B);
- Software executed on modern embedded MCUs / GPUs, which in themselves exhibit non-deterministic behavior;
- Very high change rates of the applications and adaptation to different hardware platforms;
- Significant security challenges by a larger attack surface then usual in traditional embedded systems
- High level of software re-use to make long-term maintenance commerically viable (requirement of the European Cyber Resilience Act (CRA)).

The paper will discuss in chapter 2 the particularities of OSS versus proprietary source software (PSS). Chapter 3 outlines the certification fundamentals of *exida* for safety-related OSS projects. The certification fundamentals are not intended for a “one approach fits all”, but shall allow tailoring to the strength of the individual OSS projects on a case by case basis. The annex details the principles introduced in chapters 3.

2. Opportunities and Challenges of assessing Open-Source Software

Open-source software (OSS) with the current focus on Linux is contained in most modern safety-related systems—from operating systems such as Linux to compilers, libraries, and analysis tools. OSS typically adopts lessons learned faster than proprietary software.

Both the OSS and Functional Safety (FuSa) communities achieve high software quality, but by very different means. Several parties have attempted to justify OSS under FuSa standards, yet many assessments stalled because applicants and assessors interpreted those standards differently. The key questions are therefore:

- How do proprietary source (PSS) and OSS projects differ while achieving comparable quality?
- How does OOS support independence of mixed-criticality and mixed complexity software subsystems?
- How can OSS projects be assessed up to ASIL B / SIL 2 given these differences?
- How do FuSa standards allow openness while ensuring risk mitigation?

Background on FuSa standards is summarized in Annex A.1. The following clauses discuss the practical differences between proprietary and open-source projects and their implications for safety qualification.

2.1 Offer-driven versus Order-driven

Proprietary software (PSS) projects are **order-driven**: management defines a product, specifies product requirements, allocates resources, and controls execution top-down.

OSS projects, in contrast, are **offer-driven**: independent upstream teams develop and test software components and features as offerings, while downstream integrators select, verify, and integrate components.

This separation of offering and selection is not new — it mirrors the process-industry model, where suppliers offer qualified devices and operators select them based on suitability and prior-use evidence rather than solely on compliance documentation.

2.2 Meritocracy and Gatekeeping in OSS versus Corporate Promotion

In safety-related projects of high complexity, irrespective of OOS or PSS, gatekeeper experts with review and release authority are crucial quality and safety barriers. These technical gatekeeper roles require high technical competence, extensive application experience, commitment, and professional responsibility for design and code quality as well as sustained contribution quality.

In OSS projects, contributors who continuously provide valuable technical input earn the trust of maintainers and the wider community. Release and integration authority therefore emerges from demonstrated technical competence and peer recognition rather than formal managerial appointment. This meritocracy-based hierarchy establishes clearly identifiable technical gatekeeping roles.

In PSS, authority is assigned by management. Promotion criteria often value communication, organizational and financial skills more than technical depth. Strict reviewers and safety analysts

can even receive lower collaboration scores in yearly performance evaluations, discouraging rigorous peer review and safety analyses.

Most kernel maintainers demonstrate a lifelong commitment to their software subsystems. Even if the maintainer is employed by a company that contributes to OSS development and receives a new assignment, he still requests the necessary time for maintainer activities. This long-term commitment leads to a continuity of subsystem knowledge that can be difficult to maintain in proprietary projects where development and long-term maintenance are often organizationally separated. The continuity can make impact analysis more accurate, provided that roles and review activities are documented and traceable within the safety-relevant configuration.

For functional safety qualification, irrespective of OOS or PSS, it is essential that the assignment of these roles, the review activities performed, and the resulting release decisions are documented and traceable. In mature OSS projects such as the Linux kernel, attributed commit histories and publicly archived review discussions provide a structured and reconstructable record of these gatekeeping activities.

While this OSS process does not replace downstream safety impact analysis, it reduces the likelihood of unsafe or technically unsound changes being introduced into the upstream code base and supports auditable accountability for release decisions.

2.3 Change Management and Robustness by Feedback Loops from a large Community versus Top-Down V-Model Driven

Mature OSS projects such as the Linux kernel benefit from continued operational exposure at extreme scale. Linux operates in high-concurrency and adversarial environments such as cloud infrastructures, financial systems, and internet-facing servers. This exposure and the OSS feedback mechanism increases the likelihood that rare race conditions, resource exhaustion paths, and corner-case behaviors are discovered and corrected prior to adoption in safety-related OSS Linux distributions.

The tight feedback loop with the large OSS user community and a hierarchical, change-driven workflow are the backbone of OSS product development. No OSS component will be considered for critical applications if it hasn't gone through multiple successful prior-use loops in non-critical applications with reviews and improvements tracked by the OSS community. The feedback loop notifies the developers of both bugs and behavior that the users did not expect. The hierarchical releases at the change-driven workflow are heavily based on experts (maintainers), which are publicly named for each patch. This level of personal public commitment is completely opposite to PSS.

In safety-related system development of PSS, the V-model is firmly established in industry practice and codified by FuSa standards. It structures the development flow in system (item) definition and functional concept → hazard analysis and resulting safety goals and threat analysis → safety & security concepts and requirements → allocation to and implementation in and testing of mechanics/ hardware/ software with related safety & security analyes → various levels of integration and testing → system level validation → field monitoring.

In modern software development, however, large software systems and OSS development deviate from the V-model and follow Continuous-Integration / Continuous-Deployment (CI/CD) with the underlying principle of *"release early, release often, fail fast"*. Note: Linux merges 10,000–12,000

patches per release every 9–10 weeks. CI/CD pipelines ensure that enhancements mature on side branches before merge. The main branch remains stable until release approval, after sufficient review, testing, and user validation. Furthermore, the Linux Long-Term Support (LTS) model enforces controlled backporting of fixes to stable branches with strict review discipline, reducing unintended feature introduction and limiting change scope.

Everyone in the OSS community can review proposed code; feedback and discussions are public. In the Linux community, such open reviews can be harsh by corporate standards but are regarded as essential quality measures. Code commits publicly record the name of the person who made the change and the maintainers who reviewed it, the patch differences and discussion traces. This openness is unthinkable in PSS.

The publicly accessible defect history, patch discussions, and CVE tracking provide transparent insight into failure modes and their resolution. This traceability supports retrospective defect trend analysis and strengthens evidence-based safety arguments.

Although the hierarchical, change-driven workflow is firmly established in the OSS community, the fundamental change management method required by FuSa standards is not applied transparently: **Impact Analysis** and returning to the applicable V-model phases. Subsystem maintainers justify the lack of well-specified impact analyses with their long-term knowledge of the components and dependencies with other components affected by the change request. Depending on the extent and severity of the change requests, this weakness must be compensated later downstream by the distributor performing formally recorded change impact analyses for each adopted upstream release and for safety-relevant patches within the qualified configuration. The impact analysis explicitly evaluates:

1. Effects on safety-related functions and assumptions of use,
2. Potential violation of freedom-of-interference arguments,
3. Required re-validation, regression testing, or safety case updates.

The structured, documented impact analysis at the distribution level mitigates for the absence of formally prescribed V-model re-entry mechanisms in upstream OSS development and ensures systematic evaluation of safety implications for each relevant change.

2.4 Cybersecurity Focus of OSS

The continuously evolving threat scenarios targeting OSS components and Linux have prompted the OSS community to adopt a workflow driven by changes, with code-level analyses of root causes and impacts. Cybersecurity researchers have access to the source code and its version history to identify vulnerabilties for newly discovered threat scenarios. Vulnerabilities are tracked in the CVE (Common Vulnerabilities and Exposures) database. The OSS community has developed an excellent understanding of code vulnerabilities and tools that can automatically detect them. This knowledge has led to the development of new OSS languages such as Rust, which is designed to prevent certain vulnerabilities from being introduced during code development.

This focus on avoiding or detecting code vulnerabilities is highly correlated with FuSa software safety objectives.

2.5 Requirements versus Behavior expected by the Community

OSS development is driven by features and behaviors expected by users. The OSS feedback loop highlights unexpected behavior, even if the OSS developers did not anticipate the requirements. . ISO 26262 enforces a highly prescriptive and hierarchical requirements-driven V-model interpretation. IEC 61508, in contrast, applies the V-model to traceably identify and develop safety functions and their safety properties, thereby avoiding excessive requirements refinement.

In practice, the IEC 61508 approach keeps engineers closer to the actual safety intent of the system. Therefore, **feature-driven development in OSS aligns more naturally with the IEC 61508 philosophy**, while ISO 26262's strict requirements decomposition can distance developers from the underlying safety goals and safety functions.

From a functional safety assurance perspective, the challenge is **not to force OSS into a prescriptive ISO 26262 interpretation but to demonstrate — within a more open, risk-based framework such as IEC 61508 or an equivalent interpretation of ISO 26262 — that its processes and evidence achieve the safety objectives**.

2.6 Hierarchical Community-Driven Testing versus Descriptive Testing

All safety standards enforce precisely specified static code analysis and testing methods. In contrast, the OSS Linux community has developed multiple independent layers of diverse code analysis and testing workflows, tools and test suites.

1. Review, static code analysis and testing by the upstream developers and maintainers.
2. Quality consistency and enhancement projects for the upstream development. Large companies such as IBM, Google, Red Hat, ARM, and Qualcomm support these quality consistency and enhancement projects. Examples of these projects include KernelCI, Linux Kernel Functional Testing and kerneltests.org.
3. In addition to traditional analysis and testing methods, the OSS community emphasizes onto dynamic stress test suites, such as stress-ng.
4. Feedback loop from the OSS users, see chapters above.
5. In addition to upstream code analysis and testing projects, each distributor maintains a chain of quality enhancement distributions, such as Fedora and CentOS.
6. Distributors perform extensive testing on their commercialized distributions, such as RHEL.
7. Distributors perform additional testing on their safety-related distributions.

Unlike PSS testing, the daily results of the upstream OSS test projects are publicly available,

Still functional safety assessors face uncertainty regarding whether the diverse test strategies complement each other and what the achieved test coverage is. The test teams of the distributor must resolve this uncertainty in the safety qualification projects.

2.7 Software Architecture documentation or the lack thereof

System integrators, software safety analysts and assessors need system-level architectural documentation to comprehend the interactions and dependencies of components and services within large software systems, as well as the isolation between safety-related and non-safety-related

software subsystems (freedom-of-interference). In contrast, the OSS community, incl. Linux, sees little value in such documentation.

Why doesn't the lack of system-level architectural documentation mean that the Linux system architecture is weak? The reasons in the OSS history and organization and the consequences for the OSS projects to be certified are explained briefly below.

Unix, OSS and Linux are built on the credo of small and loosely coupled subsystems with a single purpose reposition and related project development and maintenance teams. OSS components are precisely defined by their code APIs, and mature APIs are stable.

Most maintainers are responsible for their subsystems (vertical), not for the system integration (horizontal) or the corresponding system architecture. A tiny hierarchy of maintainers makes system-level architectural decisions through mailing list discussions. The architecture lives in the minds of the maintainers. This is only possible in a society based on technical meritocracy[1] of maintainers, who are committed to lifelong engagement.

The frequency of change in OSS is very high, see ch. 2.3. In OSS such as Linux, architecture evolved organically as the emergent outcome of thousands of incremental design decisions visible in the code history. This "architecture-as-code" model conflicts with safety assessors' expectation of stable, reviewable architecture specified in documents. Any documentation that differs from code will quickly become outdated. Maintaining architectural diagrams continuously becomes more work than coding without contributing to bug fixes or feature and performance improvements. Only when subsystems become extremely complex in itself, the subsystem project starts to consider design documentation for their understanding of the subsystem. Even then, the subsystem developer focused documentation is usually textual (Architecture-as-Code) and not graphical, and therefore difficult to quickly grasp, and doesn't address the need of safety assessors for system-level architectural views of how subsystems interact and depend.

While the architecture-as-code approach has proven effective within the OSS ecosystem and supports rapid evolution and technical coherence, it does not by itself satisfy the explicit architectural documentation and related safety analysis expectations imposed by functional safety standards. In particular, safety assessors require a stable, analyseable representation of subsystem boundaries, interactions, and assumptions of use. Therefore, for safety qualification, the architectural transparency achieved through code history and maintainer knowledge must be complemented by explicit downstream activities that define scope, document safety-relevant interactions, safety analysis, and derived safety mechanisms. The assessed OSS projects take different action steps:

- **High-level architecture:** Description of the subsystems / packages.
- **Scope reduction and isolation**: Identify and minimize the number of subsystems / packages that matter for safety. Isolate safety-related software components from non-safety-related components.

  Justify the reduction of safety-related software components using predetermined OSS product configurations, Assumptions of Use (AoU) and external safety mechanisms.

[1] It has led to the modern term "tribal knowledge architecture".

- **Architectural interaction documentation**: Document the interactions and dependencies of packages and components, particularly across the boundary between the safety- and non-safety-related containers.
- **Extensive analysis of the risks and unmitigated failure modes** of the subsystem or **addition of safety mechanisms** monitoring the safety-related performance of the subsystem at runtime. The risk analysis may take many forms, such as traditional safety analyses (FTA, STPA), or statistical testing and evaluation, or risk assessments based on extensive interviews of maintainers. The latter is not explicitly described by FuSa standards. Also statistical testing and evaluation go well beyond statistical field experience evaluation as described in IEC 61508-7, Annex D.
- **Evidence of correct behavior:** Provide test evidence for the isolation and correct interaction of components beyond the behavior specified by their API.

3. How *exida* assesses Open-Source projects

As explained above, the lifecycle models of FuSa standards and the OSS community are very different. This clause should explain how we still argue that OSS products can be certified to FuSa standards.

3.1 Limited Feature Set and Software Element in Context Qualification

The OSS products certified by *exida* are limited to the bare minimum feature set required for the targeted embedded applications. This is particularly important for large OSS products, which can be configured extensively. Unnecessary features and related OSS components are removed from the certified distributor configuration or blocked by AoU (Assumption of Use) or wrappers.

Some OSS product qualifications specify a particular usage context for which the OSS is qualified, assessed, and certified. This is an important step as it enables more application-focused safety analysis techniques, such as STPA (System-Theoretic Process Analysis) by Prof. Nancy Leveson.

3.2 *exida* Assessment Justification Model for OSS (Safety Case and Layers of Protection)

exida assesses open-source projects against their software safety assessment and justification database [L3]. The assessment database condenses the software safety process- and product-related objectives and requirements of ISO 26262 (and IEC 61508 on request) into Top-Level Safety Requirements (TLSR).

The applicant should demonstrate by their safety case that their OSS distribution meets:

1. the technical safety requirements defined by the applicant for the specified use cases,
2. all safety-related claims in user-accessible documentation, and
3. the normative objectives and derived top-level safety requirements of the claimed FuSa standards.

As upstream OSS development does not typically follow the prescriptive lifecycle processes defined in ISO 26262, the applicant shall identify the OSS weaknesses by some form of risk assess-

ment or safety analysis. The applicant may then define and implement **Layers of Protection (LOP)** as measures and mechanisms to compensate the identified OSS weaknesses.

The LOPs described in the following tables do not in themselves establish compliance with the product safety claims and functional safety standards. Rather, they represent structured tools that may be used by the applicant's safety team within their safety case against the product safety claims and the applicable TLSRs. Therefore, the paper doesn't give a justification of each LOP against the properties specified by IEC 61508-5 Annex F.

The sufficiency of the distributor's arguments and supporting evidence is subject to independent assessment by *exida*. *exida* assessors review and judge

1. the identification of OSS weaknesses by the applicant by a risk assessment or safety analysis,
2. the set of LOPs specified and implemented or executed by the applicant to compensate the identified OSS weaknesses, and
3. the arguments and evidence given by the applicant for each requirement of the assessment database.

For OSS, *exida* identified protection layers across three domains:

LOP_A (Architecture)	Assurance through OSS component scope limitations and independent runtime safety mechanisms.
LOP_P (Process)	Process integrity enhancement through risk assessment and safety analysis, additional testing, prior use, meritocracy, and long-term support.
LOP_C (Continuous certification)	Confidence from continuous assessment and certification activities.

This approach combines community-driven quality mechanisms with risk assessments, safety analyses, and test activities performed by distributors and integrators, turning them into measurable safety evidence. Using the *exida* software safety assessment and justification database [L3], *exida* evaluates the resulting assurance arguments to determine whether they achieve an integrity level comparable to IEC 61508 or ISO 26262, even when traditional V-model work products are not available.

The following tables compile examples of LOPs leveraging the strength of OSS.

Note: The reference from the individual LOPs to safety standards are currently only to ISO 26262. Similar references will be compiled to IEC 61508.

LOP_A	Architectural LOP
LOP_A_1	**Blocking access** to OSS components and services for users/ integrators by built-time restrictions or Assumptions of Use (AoU). Note: Safety arguments may be made that specified groups of qualified users/ integrators get access to a larger set of OSS components and services. [ISO 26262-4 cl. 6.4.4.4 (well-trusted designs)]

LOP_A	Architectural LOP
LOP_A_2	**Built-in mechanisms against violation of Freedom of Interference (FFI)** between user applications of different safety-integrity. Example: Container. Note: Container-based isolation contributes to user-domain FFI. Kernel-level interference remains within the safety qualification scope. [ISO 26262-6 cl. 7.4.9]
LOP_A_3	**Independent monitors** such as logical and temporal program flow monitoring. [ISO 26262-6 cl. 7.4.12 and Annex E.4; IEC 61508-3 cl. 7.4.2.7]
LOP_A_4	A safety-related Linux distribution is a **built**, not a repository of code and configuration files.

LOP_P	Process LOP
LOP_P_1	**Prior use, confidence from use**, continued operational exposure at extreme scale, bug feedback mechanisms and maintenance history [ISO 26262-4 cl. 6.4.4.4 (well-trusted designs), ISO 26262-8 cl. 11.4.7]
LOP_P_2	**Downstream testing and extensive use** of non-commercial Linux distributions (ex: Fedora, CentOS) before large scale commercial use. [ISO 26262-6 cls.9 to 11]
LOP_P_3	**Technical Gatekeeping and Attributed Release Authority** Under the provisions of clause 2.2, maintainers represent a crucial review, release and integration authority. Attributed commit histories and publicly archived review discussions provide a structured and reconstructable record of these gatekeeping activities. Under the provisions of clause 2.2, subsystem maintainers represent recognized review, release, and integration authorities within the OSS development hierarchy. No change can be integrated into the mainline kernel without explicit maintainer approval. This hierarchical peer-review and release gating mechanism acts as a preventive barrier against the introduction of systematic faults by ensuring that each modification undergoes expert evaluation prior to integration. Attributed commit histories, review tags (e.g., “Reviewed-by”, “Acked-by”), and publicly archived technical discussions provide a structured, traceable, and independently reconstructable record of these gatekeeping activities.
LOP_P_4	**Risk assessment and safety analysis** [ISO 26262-6 cl. 7.4.12 and Annex E]
LOP_P_4.1	Use Case 1: In the Red Hat RHIVOS project an important LOP is the continuous **interviews of OSS maintainers** by the Red Hat safety team. [ISO 26262-6 cl. 7.4.10, .12 and Annex E] The interviews are guided by a safety re-engineering team - Upstream maintainers explain their processes and designs - Red Hat safety teams interview the maintainers. The interview is guided by questionnaires. - In a Risk Assessment, the Red Hat safety team identifies risks jointly with the maintainers

LOP_P	Process LOP
LOP_P_4.2	Use Case 2: Codethink uses other safety analysis techniques as **STPA** (Systems-Theoretic Process Analysis) and **statistical analysis** [L2]. [ISO 26262-6 cl. 7.4.10, .12 and Annex E, and IEC 61508-7 Annex D]
LOP_P_5	Mitigation of identified risks by LOP_A_1 Blocking of access.
LOP_P_6	Mitigation of identified risks by LOP_A_2 monitors.
LOP_P_7	**Distributor and integrator responsibilities**
LOP_P_7.1	**Minimal package selection** for the safety-related Linux distributions based on prior use history, upstream test evidence, and maintainer interviews. [ISO 26262-4 cl. 6.4.4.4 (well-trusted designs), ISO 26262-8 cl. 11.4.7]
LOP_P_7.2	Impact analysis, additional downstream testing and KPI measurement **with each release** [ISO 26262-6 many clauses; ISO 26262-8 cl. 8.4.3.1; IEC 61508-3 many clauses]
LOP_P_7.3	**Downstream testing** for the safety-related Linux distributions as defined by the Risk Assessment [ISO 26262-6 cls.9 to 11]
LOP_P_7.4	Distributors, integrators and users have the obligation to **identify weaknesses in the integration** of the product as offered. They shall perform tests as deemed required to close the weaknesses. [ISO 26262-6 cl. 7.4.2.3]
LOP_P_7.5	Distributors perform formally recorded change impact analyses as described in clause 2.3. [ISO 26262-8 cl. 8]
LOP_P_7.6	Distributors, integrators and users shall report bugs and anomalies and trace their resolution. [ISO 26262-6 cl. 7.4.2.3]
LOP_P_7.7	**Long-Term Support** Long-term support by **distributors under continuous certification** ensures ongoing maintenance, safety-relevant updates, and quality monitoring during the entire operational lifetime. [ISO 26262-6 cl. 7.4.2.3]
LOP_P_8	**User responsibility**
LOP_P_8.1	Users of the OSS product have the obligation to integrate the OSS product respecting the associated **man pages** and **safety manual**. [ISO 26262-11 cl. 4.5.4.9 Integration documentation set]
LOP_P_8.2	The users of the certified OSS product shall comply with the respective **safety manual** and its AoUs and the API man pages. [ISO 26262-11 cl. 4.5.4.9 Integration documentation set]

LOP_C	Assessment and Certification LOP
LOP_C_1	*exida* condensed the software-related ISO 26262 objectives and requirements into a **safety justification and assessment database** [L3]. The applicant shall provide arguments and evidence for meeting the items of the assessment database or for proposing alternative measures equivalent in achieved software quality and safety. [ISO 26262-2 cl. 6.4.9 and 6.4.12]
LOP_C_2	*exida* uses only highly **skilled SW assessors with long-term experience in Linux and OSS** for its OSS projects. [ISO 26262-2 cl. 6.4.9 and 6.4.12]

LOP_C_3 **Independent continuous assessment and certification**.
Feedback from a large community. Impact analysis of why issues identified by the community were not sufficiently covered by the risk assessments and independent assessment. [ISO 26262-2 cl. 6.4.9 and 6.4.12]

Annex – Key OSS Considerations for Functional Safety Certification

This annex details the discussion of section 2.x of strength and weaknesses and the key OSS considerations of safety-related OSS distributors for fulfilling ISO 26262 objectives.

A.1 How do some FuSa standards ensure risk mitigation while promoting openness?

This clause provides background on how functional safety (FuSa) standards address openness, pre-existing software, and different development models. It summarizes the main differences between **IEC 61508** and **ISO 26262**, particularly with respect to their flexibility in accommodating open-source and iterative development practices.

IEC 61508 and ISO 26262 have different approaches to ensure risk mitigation while allowing for openness. IEC 61508 uses the V-model as guiding principle but allows in its **route 3S** for **assessment of pre-existing software**, see IEC 61508-3:2010 clause 7.4.2.13. Furthermore, IEC 61508 doesn't prescribe certain development and V&V evidence documents but requires **information**. IEC 61508 can therefore be regarded as open to both order-driven V-model development and the supply-driven OSS approach.

ISO 26262 is more ambivalent. While its safety objectives are strictly goal-oriented, its extensive use of the V-model prescribes sequential steps and associated work-products. Nevertheless, it allows for assessment against the objectives of the standard and requirements detailing the objectives.

A.2 Strength and Weaknesses of OSS in the Light of Functional Safety

What are strengths of OSS that support the objectives of FuSa and could be used to mitigate weaknesses of OSS before attempting assessment.

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
Category: Processes - Upstream	
OSS is Offering-driven - Small teams led by ambitious individuals develop their product idea based on community need they experienced - The team offers their product to the OSS community. Per the Unix philosophy, the products have preferably a single purpose. - OSS experts (often with competing goals) argue for the best technical solution in the open. - Releases are 'when ready' not to some project planning deadline or production deadline. - Other teams will use the product only if the feature set and the maturity are convincing.	**PSS is Order-driven** - Senior management specifies product requirements and processes and allocates resources and time - PSS projects must adhere to quality standards and artefacts must be documented. - Large quality and safety management teams monitor compliance - Efforts are dictated by the organization and/or regulatory or normative frameworks.

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
- The early and continuous feedback from users make requirements creeping not a dominant challenge as it is in PSS projects. - The documentation is mainly located in the code. This makes it easier to keep the documentation up to date and monitor it.	
Loose coupling - OSS components and their OSS project teams are intentionally loosely coupled. - Both are challenged very early in development by diverse parties in the OSS community and the downstream integrators. The integrators will ultimately decide whether a feature will be integrated.	
Change Management - The OSS product development is governed by a hierarchical, change-driven workflow that primarily utilizes maintainer reviews and integration testing. - Code commits publicly record the name of the person who made the change and the names of the maintainers who reviewed it. This would be unthinkable in PSS.	**Change Management** - The change-driven workflow of OSS product development lacks well-documented impact analyses and related checklists. - Depending on the extent and severity of the change requests, this weakness must be compensated later downstream [LOP_P_7.1].
Public access - Code, API definitions and development artefacts are publicly available - Invitation to extensive reviews - The open OSS repositories have a long history of detecting, patching, and research by academics to find bugs and cybersecurity vulnerabilities.	
Category: Self-Organizing Competence Structure (Meritocracy)	
- In OSS projects, release authority grows naturally from proven technical skill and steady contribution quality. Developers who repeatedly deliver solid work earn the trust of maintainers and peers. - This self-organizing competence structure creates effective quality gates, as every change is reviewed by people who have demonstrated expertise.	- OSS projects often lack clear documentation of who holds release authority and how responsibilities are transferred. - When key maintainers step back, continuity and knowledge transfer can be fragile. - Contributor expertise is recognized by peers rather than by formal qualification, which makes it harder to demonstrate in safety assessments.

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
- The open discussion of review comments and release decisions keeps the process transparent and accountable.	
Category: Processes - Downstream	
Upstream vs. downstream have very diverse processes and objectives. **Selection and Integration testing** - Downstream selects OSS offerings based on feature and the maturity (prior-use). - Red Hat enforces extensive prior-use by the integration pipeline of Fedora, ELN, CentOS, RHEL. - Downstream tends not to select or replace unmaintained OSS components. - Downstream performs extensive integration tests.	- In PSS unmaintained code is usually left in the product when a developer leaves a company.
Category: Supply Chain Integrity and Security	
- The transparency of source code, build processes, and dependencies allows independent verification of integrity and origin. - Security vulnerabilities and integrity issues are publicly disclosed and collaboratively resolved, enabling rapid response and continuous improvement.	- The absence of Development Interface Agreements (DIA), as per ISO 26262-8 clause 5, leads to undefined responsibilities and uncontrolled interfaces between upstream developers and downstream distributors and integrators. - The decentralized OSS contribution and high number of dependencies increase the risk of unnoticed modifications or unverified upstream components.
Category: Processes - Change management	
OSS Processes are Change-driven - OSS development is based on CI/CD and change-management. - Best CI/CD practices in SW development are enforced (e.g., linting, styleguides, CI/CD) - The change-driven OSS philosophy of "release frequently and provoke feedback and bug identification (failing fast)" leads to extensive use and enhancement in not critical applications. - Open bug tracking	- OSS processes and their implementation are very inhomogeneous. Note: Also, PSS processes/ style guides/ ... may differ between companies. - OSS teams are not controlled by strict processes and requirements specs. - OSS projects follow the CI/CD model and comply only partly with the sequential V-model.

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
	- Graphical design documentation of OSS projects is very weak. - Some textual documentation outside code exists such as The Linux Kernel, The GNU C Library.
Category: Long-Term Support (LTS)	
- The large OSS ecosystem and broad user base often ensure long maintenance lifecycles, frequent bug fixes, and transparent tracking of issues and patches. - Mature distributions provide stable, long-term branches maintained over many years.	- OSS maintenance depends on voluntary community effort and can decline when key maintainers leave or projects lose traction. - Long-term continuity relies on downstream distributors rather than upstream developers. - Continuous assessment and certification by independent bodies such as *exida* strengthen confidence that safety arguments and evidence remain valid throughout the LTS phase.
Category: Requirements	
OSS is offer-driven and feature-oriented - glibc features follow the POSIX standard - If an OSS product feature set doesn't attract users, it dies early, i.e., only requirements that answer user needs will survive. Unmaintained OSS components get replaced or removed from distributions. - Offered features (equivalent to requirements) are specified by the code API.	- ISO 26262 demands hierarchically refined requirements. - OSS is feature driven with high-level requirements only. - OSS projects have no classic requirements specification and tracking. - Some OSS qualification projects are verification driven using safety analyses to identify the safety requirements.
Category: Architecture	
- OSS and LINUX focus on small and loosely coupled components. This has sometimes been violated in recent years. - glibc features follow the POSIX standard - OSS components are precisely defined by their code API. The API of mature components is stable.	- The system- and component-level architecture documentation is sparse.
Category: Components	
- OSS components are well defined by their code API.	- Component development documentation and testing is very inhomogeneous

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
- User API of mature components such as LINUX kernel and glibc is very stable. - Small and loosely coupled components are a foundational objective of OSS and LINUX. - Components and their development go through a very thorough review process by the community.	when they do not share the same maintainer. - Documentation outside the code base is not accepted by the OSS community.
Category: Component Integration	
- Thorough component selection and integration testing by downstream distribution integrators - Testing performed by many independent parties o Developers o Package and distribution integrators o Independent test teams	- Integration testing is different between packages and distributions.
Category: Portability	
- Architecture is designed to support different hardware. Device specific code (processor and drivers) is clearly delineated. - Processor specific code is actively minimized. - Close collaboration between kernel and compiler developer	
Category: Usage	
- Usage of an OSS component is described by man pages on the API	- No guidelines on integration. - No safety manual of the supplier. - The integrating user has no obligation to re-run V&V as they may be required by a safety manual
Category: Prior-use	
- OSS components gain huge prior-use experience in not critical applications, before they are considered for (safety-) critical applications. - Red Hat enforces extensive prior-use by the integration pipeline of Fedora, ELN, CentOS, RHEL. RH also selects OSS components based on their reliability history. - Public access bug tracking	- Bug tracking went through a long history of fundamental changes. - Consequently, tracing of prior-use history can be difficult.

Strength of OSS compared to PSS	Weakness of OSS compared to PSS
Category: Tool chains	
- Developed by the OSS community per their agreed best practices in SW development. - Tool chains are continuously enhanced.	- Tool qualification is very inhomogeneous.

Literature and Terms

Literature and Standards

	Title
L1	ISO 26262 Objectives and Top-Level Safety Requirements for Assessment; ISO26262_TLSR_for_assessment_V1R3.xlsx; *exida* Piotr Serwa, Dave Butler; Feb. 27, 2025
L2	Fault Tree for Next-Gen Safety Systems; Nicholas Mc Guire, Imanol Allende, Kaspar Matas; 2026
L3	ISO 26262 Objectives and Top-Level Safety Requirements for Assessment; ISO26262_TLSR_for_assessment_V1R3.xlsx; *exida* Piotr Serwa, Dave Butler; Feb. 27, 2025
N1	IEC 61508:2010, parts 1 to 7, Functional safety of electrical/electronic/programmable electronic safety-related systems
N2	ISO 26262:2018, part 1 to 12, Road vehicles — Functional safety
N3	ISO PAS 8926:2022, Road vehicles — Functional safety — Qualification of pre-existing software products for safety-related applications

Terms and Abbreviations

Term	Explanation
AoU	Assumptions of Use (AoU)
CI/CD	Continuous-integration / continuous-deployment
CRA	European Cyber Resilience Act - https://www.european-cyber-resilience-act.com
CVE	Common Vulnerabilities and Exposures is a dictionary of common names (CVE Identifiers) for publicly disclosed information security vulnerabilities. - https://en.wikipedia.org/wiki/Common_Vulnerabilities_and_Exposures
FFI	Freedom of Interference ISO 26262-1: cl. 3.65: Absence of cascading failures between two or more elements that could lead to the violation of a safety requirement.
FuSa	Functional Safety IEC 61508-4: Part of the overall safety relating to the EUC and the EUC control system that depends on the correct functioning of the E/E/PE safety-related systems and other risk reduction measures
GSN	Goal Structured Notation, hierarchical graphical structuring of a safety case using claims, arguments and evidence, and sometimes defeaters.
HW	Hardware
KernelCI	A community-based open-source distributed test automation system focused on upstream kernel development. The project is currently working on improved LTS kernel testing and validation; consolidation of existing testing initiatives; and increased pool of hardware to be tested. https://dashboard.kernelci.org/tree
LKFT	Linux Kernel Functional Testing - mission to improve the quality of the Linux kernel by performing functional testing on Arm hardware. Focus on LTS, stable,

Term	Explanation
	and upstream development branches with the Arm ecosystem as our first-class citizen. https://lkft.linaro.org/tests/
LOP	Layer of Protection A LOP shall prevent or mitigate the unwanted consequences of a hazard. IEC 61508-5 Annex F defines the following safety characteristics that a LOP shall have: Specific — to a hazard and its consequences Effective — capable of mitigating the outcome of concern when all other measures have completely failed Independent — of other LOP Dependable — can be counted on to do what it was designed to do Auditable — facilitate regular verification & validation of the LOP.
LTS	Long-Term Support https://en.wikipedia.org/wiki/Long-term_support
Merit	The quality of being particularly good or worthy.
Meritocracy	Government or the holding of power by people selected according to Merit.
OSS	Open Source Software
PSS	Proprietary (Source) Software
Prior-use	IEC 61511-1: Documented assessment by a user that a device is suitable for use in a safety instrumented system (SIS) and can meet the required functional and safety integrity requirements, based on previous operating experience in similar operating environments
RHEL	Red Hat Enterprise LINUX
RHIVOS	Red Hat In Vehicle Operating System
Rust	Rust is a general-purpose programming language with an emphasis on performance, type safety, concurrency, and memory safety. https://en.wikipedia.org/wiki/Rust_(programming_language)
SQA	Software Quality Assurance
STPA	System-Theoretic Process Analysis A system safety analysis method, developed by Prof. Nancy Leveson
stress-ng	Dynamic stress test suite for Linux, considered as the industry standard for Linux stress testing. It includes over 200 "stressors" that can target the CPU, memory, I/O, network, and specific kernel interfaces. It can dynamically generate virtual memory pressure and test real-time system latencies. https://manpages.debian.org/testing/stress-ng/stress-ng.1.en.html
SW	Software
TCL	ISO 26262-8 clause 11: Tool Confidence Level

Versions

Version	Author, Date	Enhancements
V0 R1	Rainer Faller 15.09.2025	Fundamental difference of OSS and proprietary (source) software (PSS)
V0 R2	Rainer Faller 19.09.2025	Updated by CI/CD Ready for first feedback
V0 R3	Rainer Faller 01.10.2025	Difference of Features vs Requirements *exida* Assessment database added First review comments considered (Arne Haas, Dr. Robert Maier)
V0 R4	Matthew Storr Rainer Faller 15.10.2025	RH contribution Rework section on *exida* assessment
V0 R5	Rainer Faller 23.10.2025	Feedback from Jonathan Moore and Alexander Eggerer added Document converted to Word
V0 R6	Rainer Faller 02.11.2025	Review comments from Peter Müller and Mike Medoff addressed
V0 R7	Rainer Faller 04.11.2025	Main part tightened up. Details moved into annex ChatGPT review comments and recommendations integrated
V0 R8	Rainer Faller 21.11.2025	Terms definition added Integrated some of the review comments of Prof. Nicholas Mc Guire Class of target systems added Re-emphasized the importance of the V-model on system level
V0 R9	Rainer Faller 07.12.2025	Added a clause on Software architecture or the lack thereof. Changed the order of 2.1.x clauses accordingly.
V0 R10	Rainer Faller 17.12.2025	Integrated the feedback from Jonathan Moore. Agreed on the integration of the remaining review comments of Prof. Nicholas Mc Guire with Jonathan Moore New chapters 2.1.3 and 2.3.1
V0 R11	Rainer Faller 09.01.2026	Restructuring to better separate the content of OSS distributors and *exida*.
V0 R12	Rainer Faller 24.02.2026	Feedback from Giovanni Dallara integrated on – cl 2.3 Community feedback loops; cl. 2.6 Linux testing projects Feedback on cl. 2.2 Meritocracy integrated. References from LOPs to ISO 26262 clauses added.

ToDo items for later releases of the paper

Comment	Commenter	Response – Rainer Faller
Add explanation, why the OSS community sees little value in MC/DC unit code coverage testing.	Rainer Faller	Postponed, considered a subject of a distributor-specific publication.
References of the LOPs should be extended to IEC 61508.	Rainer Faller	
Explain the liability of the distributor not of the OSS Linux community	Dr. Giovanni Dallara	Postponed, considered a subject of a distributor-specific publication.