PROFESSIONAL PRACTICE GUIDELINES

HIGH TECHNOLOGY

DEVELOPMENT OF SAFETY-CRITICAL SOFTWARE

VERSION 1.0
PUBLISHED JULY 16, 2020

ENGINEERS &
GEOSCIENTISTS
BRITISH COLUMBIA

FIRST PUBLISHED: VERSION 1.0, JULY 16, 2020
©2020 ENGINEERS AND GEOSCIENTISTS BRITISH COLUMBIA
ALL RIGHTS RESERVED.
COVER PHOTOGRAPH ©METAMORWORKS/ADOBE STOCK

PREFACE

These *Professional Practice Guidelines – Development of Safety-Critical Software* were developed by Engineers and Geoscientists British Columbia (the Association) to guide professional practice related to the discipline of Software engineering. More specifically, these guidelines provide guidance for Engineering Professionals involved in the specification, design, implementation, verification, deployment, or maintenance of Safety-Critical Software. These guidelines focus on the application of engineering practice to Software engineering in Safety-Critical applications.

In order to protect the public, these guidelines identify the standard of practice to be followed by Engineering Professionals when developing Safety-Critical Software. Furthermore, these guidelines reference standards that apply to areas of practice associated with Safety-Critical systems, but do not necessarily require compliance to those standards for any particular project.

The scope of these guidelines includes some treatment of Software security topics for Safety-Critical Software. In modern Safety-Critical Software-Intensive Systems, Safety and security are often interdependent and complementary. The scope of Software security guidance is limited to the extent of which it is required to ensure Safety in the system(s) in question.

This document was prepared for the information of Engineering Professionals, statutory decision-makers, regulators, the public, and other stakeholders who might be involved in, or have an interest in, the development of Safety-Critical Software in British Columbia.

These guidelines outline the appropriate standard of practice to be followed at the time that they were prepared. This is a living document that is to be revised and updated as required in the future, to reflect the developing state of practice.

TABLE OF CONTENTS

PREFACE i

ABBREVIATIONS v

DEFINED TERMS vi

VERSION HISTORY viii

1.0 INTRODUCTION 1

1.1 PURPOSE OF THESE GUIDELINES 1

1.2 ROLE OF ENGINEERS AND GEOSCIENTISTS BC 2

1.3 INTRODUCTION OF TERMS 2

1.3.1 Safety-Critical Software 2

1.4 SCOPE OF THESE GUIDELINES 3

1.4.1 Industry-Specific Practice 3

1.4.2 Hardware 3

1.4.3 Software Security 4

1.4.4 Software Engineering Process 4

1.5 APPLICABILITY OF THESE GUIDELINES 4

1.6 ACKNOWLEDGEMENTS 5

2.0 ROLES AND RESPONSIBILITIES 6

2.1 COMMON FORMS OF PROJECT ORGANIZATION 6

2.2 RESPONSIBILITIES 6

2.2.1 Clients 6

2.2.2 Software Engineers 7

2.2.3 Software Developers 7

2.2.4 Software Verification 8

2.2.5 Specialist Roles 8

3.0 GUIDELINES FOR PROFESSIONAL PRACTICE 10

3.1 OVERVIEW 10

3.2 SOFTWARE ENGINEERING PROCESSES AND LIFE CYCLE 10

3.2.1 Phases of Safety-Critical Software Development 10

3.2.2 Use of Third-Party Software Artifacts 14

3.3 SAFETY ENGINEERING FOR SAFETY-CRITICAL SOFTWARE 16

3.3.1 Hazard Analysis 16

3.3.2 Risk and Criticality Analysis 19

3.3.3 Reliability Engineering 20

3.3.4 Safety Cases 21

3.4 SECURITY ACTIVITIES FOR SAFETY-CRITICAL SOFTWARE 23

3.4.1 Security Risk and Threat Analysis 23

3.4.2 Security Controls and Policies 24

3.4.3 Security Verification and Validation 25

3.4.4 Security Assurance Cases 25

3.4.5 Assessment of Third-Party Libraries 25

3.5 OBSERVATION OF DEFICIENCIES 26

3.6 RELEVANT EXTERNAL STANDARDS AND GUIDELINES 26

4.0 QUALITY MANAGEMENT IN PROFESSIONAL PRACTICE 29

4.1 QUALITY MANAGEMENT REQUIREMENTS 29

4.1.1 Professional Practice Guidelines 29

4.1.2 Use of Seal 29

4.1.3 Direct Supervision 31

4.1.4 Retention of Project Documentation 32

4.1.5 Documented Checks of Engineering and Geoscience Work 33

4.1.6 Documented Field Reviews During Implementation or Construction 33

5.0 PROFESSIONAL REGISTRATION & EDUCATION, TRAINING, AND EXPERIENCE **34**

5.1 PROFESSIONAL REGISTRATION **34**

5.2 EDUCATION, TRAINING, AND EXPERIENCE **34**

5.2.1 Educational Indicators 35

5.2.2 Experience Indicators 35

5.2.3 Examples of Education and Experience 35

6.0 REFERENCES AND RELATED DOCUMENTS **37**

6.1 REGULATIONS **37**

6.2 REFERENCES **37**

6.3 CODES AND STANDARDS **38**

6.4 RELATED DOCUMENTS **39**

7.0 APPENDIX **41**

LIST OF APPENDICES

Appendix A: Authors and Reviewers 43

LIST OF FIGURES

Figure 1: Block Diagram of an Electronic Brake Controller System 18

Figure 2: Sample Risk and Criticality Table 20

Figure 3: Sample Safety Case Argument Using Goal-Structuring Notation 22

LIST OF TABLES

Table 1: Definitions of Risk Likelihood and Severity Categories 20

Table 2: List of Relevant External Standards and Guidelines 27

ABBREVIATIONS

ABBREVIATION	TERM
BC	British Columbia
FMEA	failure mode and effects analysis
FTA	fault tree analysis
GSN	goal-structuring notation
HAZOP	hazard and operability study
IEC	International Electrotechnical Commission
IEEE	Institute of Electrical and Electronics Engineers
ISO	International Organization for Standardization
RTOS	real-time operating system
STPA	system theoretic process and analysis
UML	Unified Modeling Language
XML	Extensible Markup Language

DEFINED TERMS

The following definitions are specific to these guidelines. These words and terms are capitalized throughout the document.

TERM	DEFINITION
Accident	An event or sequence of events that culminate in one of the following: 1) harm, injury, illness, or death to one or more persons; or 2) damage to the environment.
Act	*Engineers and Geoscientists Act* [RSBC 1996], Chapter 116.
Association	The Association of Professional Engineers and Geoscientists of the Province of British Columbia, also operating as Engineers and Geoscientists BC.
Bylaws	The Bylaws of the Association made under the *Act*.
Causal Factor	An action, omission, event, or condition of a system during its operation within its deployed environment or during its development that contributes to the occurrence of a Hazard.
Client	The party who commissions a Software engineering work.
Engineering Professional(s)	Professional engineers and licensees who are registered or licensed by the Association and entitled under the *Act* to engage in the practice of professional engineering in British Columbia.
Engineers and Geoscientists BC	The business name of the Association.
Hazard	A set of conditions or an operational situation that might lead to an Accident.
Residual Risk	The Risk remaining after all implemented mitigations have been applied to the system.
Risk	A combination of two factors: 1) the severity of an anticipated Accident resulting from a Hazard; and 2) the likelihood of a Hazard occurring and leading to Accident (alternatively referred to as “exposure”).
Safety	Freedom from unacceptable Risk of an Accident occurring due to non-malicious causes.

TERM	DEFINITION
Safety-Critical	Refers to an engineering work or element of a system that meets one of more of the following criteria: 1) the item's incorrect response, inadvertent response, failure to respond, out-of-sequence response, or response in combination with other responses is capable of contributing to a Hazard; 2) the item is intended to mitigate the effect of a Hazard or result of an Accident; and/or 3) the item is intended to recover from the occurrence of a Hazard or Accident. See also Section 1.3.1 Safety-Critical Software.
Software	One or more digitally encoded instructions that are executed by a computer or similar computing hardware
Software Engineer	An Engineering Professional qualified by education, training, and/or experience who is engaged in the application of a systematic, disciplined, and quantifiable approach to the specification, design, implementation, verification, deployment, or maintenance of Software. For the purposes of these guidelines, a Software Engineer is an Engineering Professional engaged in the development of a Safety-Critical Software system, regardless of their registered discipline and declared areas of expertise with the Association.
Software-Intensive System	A system whose function depends on the execution of a principal Software element to achieve the desired objective.
Source Code	One or more commands expressed in a programming language that may be interpreted, compiled, or assembled into Software.
Threat	Anything that might exploit a vulnerability to breach security and cause a Hazard.

VERSION HISTORY

VERSION NUMBER	PUBLISHED DATE	DESCRIPTION OF CHANGES
1.0	July 16, 2020	Initial version.

1.0 INTRODUCTION

Engineers and Geoscientists British Columbia (the Association) is the regulatory and licensing body for the engineering and geoscience professions in British Columbia (BC). To protect the public, the Association establishes, maintains, and enforces standards for the qualifications and practice of its registrants.

The Association provides various practice resources to its registrants to assist them in meeting their professional and ethical obligations under the *Engineers and Geoscientists Act* (the *Act*). One of those resources are professional practice guidelines, which establish the standard of practice for specific professional activities. The Association works with experts in their fields to develop professional practice guidelines where additional guidance is beneficial or required.

Across many application domains and industries, Software forms an integral part of Safety-Critical systems. In such systems, Software has an essential role in system functions that have the potential to cause harm to persons or to the environment. These systems may be referred to as "Safety-Critical Software-Intensive Systems." Given the Risk associated with Safety-Critical systems, it is important that engineering work involving Software be undertaken by qualified and experienced Engineering Professionals.

1.1 PURPOSE OF THESE GUIDELINES

This document provides guidance on professional practice to Engineering Professionals who are involved in the specification, design, implementation, verification, deployment, or maintenance of Safety-Critical Software.

Following are the specific objectives of these guidelines:

1. Describe the standard of practice that Engineering Professionals should follow when providing professional services related to Safety-Critical Software.
2. Specify the tasks and/or services that Engineering Professionals should complete to meet the appropriate standard of practice and fulfill their professional obligations under the *Act*. These obligations include the Engineering Professional's primary duty to protect the Safety, health, and welfare of the public and the environment.
3. Describe the roles and responsibilities of the various participants/stakeholders involved in Safety-Critical Software. The document should assist in delineating the roles and responsibilities of the various participants/stakeholders, which may include the Software Engineer, developers, the client, and others.
4. Define the skill sets that are consistent with the training and experience required to provide professional services in relation to Safety-Critical Software.
5. Provide guidance as to how Engineering Professionals should communicate the outcomes of their professional activities related to Safety-Critical Software, including the use of declaration documents as described in Section 4.1.2 Use of Seal, so that stakeholders are properly informed that the appropriate considerations have been addressed (both regulatory and technical) for the specific professional activities that were carried out.
6. Provide guidance on how to meet the quality management requirements under the *Act* and Bylaws when carrying out the professional activities identified in these professional practice guidelines.

1.2 ROLE OF ENGINEERS AND GEOSCIENTISTS BC

These guidelines were prepared by subject matter experts and reviewed at various stages by a formal review group. The final draft of the guidelines underwent a final consultation process with various committees and divisions of the Association. These guidelines were approved by the Association's Council and, prior to publication, underwent final legal and editorial reviews. These guidelines form part of Engineers and Geoscientists BC's ongoing commitment to maintaining the quality of professional services that Engineering Professionals provide to their clients and the public.

An Engineering Professional must exercise professional judgment when providing professional services; as such, application of these guidelines will vary depending on the circumstances, including where project- or application-specific conditions need to be addressed or in the event that there are changes in legislation or regulations subsequent to the publication of these guidelines. Where an Engineering Professional intends to substantially deviate from applying these guidelines, consideration should be made to obtain a second opinion on the merits of the deviation.

The Association supports the principle that appropriate financial, professional, and technical resources should be provided (that is, by the client and/or the employer) to support Engineering Professionals who are responsible for carrying out professional activities, so they can comply with the standard of practice outlined in these guidelines. These guidelines may be used to assist in the level of service and terms of reference of an agreement between an Engineering Professional and a client.

These guidelines are intended to assist Engineering Professionals in fulfilling their professional obligations, especially regarding the first principle of the Association's Code of Ethics, which is to "hold paramount the safety, health and welfare of the public, protection of the environment and promote health and safety in the workplace." Failure to meet the intent of these guidelines could be evidence of unprofessional conduct and lead to disciplinary proceedings by the Association.

1.3 INTRODUCTION OF TERMS

Note that the terms defined in this section are likewise provided in many other articles and standards found in the literature. For the purposes of these guidelines, the terms defined below represent the interpretation of these terms in the context of this document.

Also see the Defined Terms section at the front of the document for a full list of definitions specific to these guidelines.

1.3.1 SAFETY-CRITICAL SOFTWARE

Software is almost always used to fulfill a larger purpose. That is, Software's existence is usually necessitated by the needs of a larger technical, social, scientific, or economic system. As a result, the level of Risk associated with the Software must be derived from Risks inherent to the use of the Software within a larger system. Some types of Software also have the potential to contribute to harm to life and health of the public or harm to the environment (for example, an Accident); such Software is considered to be Safety-Critical Software.

Specifically, Safety-Critical Software is defined as meeting one or more of the following conditions:

- Software whose incorrect action, inadvertent response to stimuli, failure to respond when required, out-of-sequence response, or response in combination with other responses is capable of contributing to a Hazard
- Software that is intended to mitigate the effect of a Hazard or the result of an Accident
- Software that is intended to recover from the occurrence of a Hazard or the result of an Accident

For example, Software deployed in a nuclear power generation facility that is intended to intervene if the

reactor overheats would probably be considered Safety-Critical. Similarly, Software that translates braking inputs from a human to the braking actuators in an automobile likely would be considered Safety-Critical. Conversely, the Software in a video game would not likely be considered Safety-Critical.

The above definitions of Accident and of Safety-Critical Software do not explicitly consider the potential for loss or damage of property or the potential for economic or financial loss. For example, financial Software that tabulates an organization's bi-monthly payroll might contribute to a financial loss due to incorrect calculations; however, in such a scenario, it is unlikely that harm to a person's health or life, or harm to the environment, would occur. Indeed, the potential for property and/or financial loss is worthy of serious consideration by Engineering Professionals who have an obligation to uphold the public well-being in these matters. However, the ramifications of these types of losses are beyond the scope of these guidelines.

Furthermore, the above definitions of Accident and of Safety-Critical Software do not explicitly include "mission-critical" Software; that is, Software whose failure might compromise the overall objectives ("mission") of a person, business, organization, or government. There may be certain circumstances where mission-critical Software could also be considered Safety-Critical; however, if the mission-critical Software performs only non-Safety functions, it would not meet the definition of Safety-Critical Software.

Importantly, the above definitions require that Engineering Professionals engaged in creating Safety-Critical Software use their judgment to identify the potential for an Accident and, as a result, apply appropriate techniques and methods throughout the life cycle of the Software to eliminate or mitigate Safety Risks.

1.4 SCOPE OF THESE GUIDELINES

These guidelines apply to Engineering Professionals involved in the specification, design, implementation, verification, deployment, or maintenance of Safety-Critical Software in BC. In particular, guidance is provided with respect to the Safety of Software for use within a Safety-Critical system whose function depends on Software.

These guidelines are not intended to provide step-by-step instructions for providing Software engineering services. Rather, these guidelines outline considerations for professionals engaged in Software engineering work. See Section 1.5 Applicability of These Guidelines.

1.4.1 INDUSTRY-SPECIFIC PRACTICE

These guidelines aim to be generic with respect to application domain and therefore should be considered a supplement to engineering practices required in a specific industry (for example, compliance with ISO 26262 Road Vehicles – Functional Safety is required by most automotive manufacturers in North America and Europe). It is the responsibility of the Engineering Professional to ensure up-to-date and relevant industry-specific engineering standards are considered during Software engineering work.

1.4.2 HARDWARE

The Safety and security of the overall system within which the Safety-Critical Software is deployed (including but not limited to other mechanical, electrical, electronic, or integrated systems) is not wholly addressed by these guidelines.

Such system elements are only considered to the extent that their behaviours and their interfaces with the Safety-Critical Software may influence the Software engineering approaches, methods, and processes that are applied to mitigate Safety Risks for the overall system.

This scope restriction applies to systems and components, not to individuals who are involved in creating or maintaining systems and components. Engineering Professionals who specialize in mechanical, electrical, or electronic systems may also be involved in creating and maintaining Software. These guidelines would then apply to the extent that the Software has a Safety-Critical role in the system in question, regardless of the declared discipline of practice of the Engineering Professional.

1.4.3 SOFTWARE SECURITY

The protection of Software (and associated data) from inadvertent or malicious actions of an agent falls within the purview of what is called "Software security." Software security is a broad field with a wide range of applications, not all of which apply to Safety-Critical Software.

These guidelines consider the security of Safety-Critical Software to the extent that security is required to maintain the safe operation of the Software. For example, it is likely that Safety-Critical Software involved in the control of a hydroelectric-power–generation facility should include security measures to limit the ability of malicious agents (persons or Software) to cause an Accident.

These guidelines recognize that Safety-Critical Software is deployed into a larger system context, and thus the security features inherent in the Safety-Critical Software are only a portion of the overall system security profile.

Software is often used as part of a system that involves the storage or transmission of sensitive personal or corporate information or data. The failure of Software to protect the privacy of such information might be detrimental to public welfare. As noted in Section 1.3.1 Safety-Critical Software above, unless privacy of the information or data in question is required to maintain safe operation of the Software, these guidelines do not apply. For example, the release of sensitive financial data might result in economic loss for the affected persons or corporations. However, unless the associated Software is considered Safety-Critical, the privacy of this information is not considered by these guidelines.

1.4.4 SOFTWARE ENGINEERING PROCESS

These guidelines describe a combination of engineering processes and techniques that should be followed when creating and maintaining Safety-Critical Software. Importantly, these guidelines focus on properties of the processes and techniques used to create the Software, rather than on the properties of the resulting Software itself. This perspective is consistent with many industry-specific standards for Safety-Critical Software.

For example, due to the complexity of most Software, it is often infeasible to verify the individual behaviour of the Software for each possible set of inputs. Instead, a combination of several verification activities (for example, testing or peer review) are mandated as part of a larger engineering process. Confidence in the behaviour of the resulting Software under all operational conditions is increased by the knowledge that a process with an appropriate level of rigour was followed to create and maintain the Software, along with appropriate provisions for ongoing maintenance and in operation service support.

1.5 APPLICABILITY OF THESE GUIDELINES

These guidelines provide guidance on professional practice for Engineering Professionals who carry out Safety-Critical Software engineering. These guidelines are not intended to provide systematic instructions for how to carry out these activities; rather, these guidelines outline considerations to be aware of when defining and subsequently carrying out the activities required to develop Safety-Critical Software.

An Engineering Professional's decision not to follow one or more aspects of these guidelines does not necessarily mean a failure to meet his or her professional obligations. Such judgments and decisions

depend upon weighing facts and circumstances to determine whether other reasonable and prudent Engineering Professionals, in similar situations, could have conducted themselves similarly.

Engineers Canada (EC 2016) provides the following position as to whether or not a Software development project should be considered Software engineering:

> "In the case of software engineering, a piece of software (or a Software Intensive System) can therefore be considered an engineering work if both of the following conditions are true:
>
> – The development of the software required 'the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software.
> – There is a reasonable expectation that failure or inappropriate functioning of the system would result in harm to life, health, property, economic interests, the public welfare or the environment."

The standard of practice that these guidelines define in relation to Safety-critical Software satisfies the first condition above, and the definition of Safety-Critical Software in Section 1.3.1 satisfies the second condition. As such, the Association considers all Safety-Critical Software projects to contain engineering work that must be conducted by an Engineering Professional.

Further details regarding the nature of the Software engineering work required for such projects are described in subsequent sections of these guidelines.

1.6 ACKNOWLEDGEMENTS

These guidelines were prepared on behalf of the Association by primary authors who are subject matter experts practicing in this field.

This document was also reviewed by a group of technical experts, as well as by various committees and divisions of the Association.

Authorship and review of these guidelines does not necessarily indicate the individuals and/or their employers endorse everything in these guidelines.

See Appendix A: Authors and Reviewers for a list of contributors.

2.0 ROLES AND RESPONSIBILITIES

This section describes common roles and responsibilities of various persons or organizations who typically contribute to the creation of Safety-Critical Software. Note that the roles and responsibilities identified below are a sample of those currently found in industrial settings so should not be considered a comprehensive list.

2.1 COMMON FORMS OF PROJECT ORGANIZATION

Roles and responsibilities might vary from project to project, and may vary significantly depending on the type of project execution model (for example, agile, scrum, or waterfall methodologies). Other project methodologies may combine, separate, or duplicate the responsibilities described below to suit the specific methodology being used by the project. However, in doing so, it is vital that the overall project organizational structure addresses the professional practice requirements described in Section 3.0 Guidelines for Professional Practice and the quality management requirements in Section 4.0 Quality Management in Professional Practice.

The discussion in this section is focused on roles that may be attributed to organizations or persons, and mostly reflects a waterfall type of development model and project management methodology. Although job titles may in some cases imply a particular role, this will not necessarily be the case. It is possible for one person to assume more than one of the roles below, provided they have the appropriate qualifications and experience.

2.2 RESPONSIBILITIES

2.2.1 CLIENTS

Clients typically commission Safety-Critical Software engineering work. The commissioned work may constitute a standalone system or be part of a larger engineering effort.

In these guidelines, a Client is defined as any party that commissions Software engineering work. This could include, but is not limited to, an independent person or organization, a separate department within the same organization, or an individual person (such as a manager) within the same department of an organization. Importantly, a client is not required to be technically qualified or skilled in the practice of Safety-Critical Software engineering.

The client should:

- define the overall scope and design of the system under development;
- define the deliverables to be produced as part of Software engineering work;
- ensure allocation of system and/or Safety requirements to the commissioned Safety-Critical Software;
- accept Software engineering artifacts (for example, specifications, Source Code, binaries, verification evidence, analysis reports) prepared or created as part of the Software engineering work;
- where applicable, ensure the proper management of aspects of system engineering work that fall outside the explicit scope of the Safety-Critical Software project (for example, a client might be responsible for assurance cases prepared for the entire system, which might include assurance cases for Safety-Critical Software); and

- where applicable, ensure proper collaboration is provided regarding design decisions that have a shared impact between the Software and the larger system.

Note that it is not required that clients (or those acting on behalf of clients) be Engineering Professionals. However, in the event that the responsibilities and activities of the client constitute engineering work (as is often the case), the client must be an Engineering Professional, or an Engineering Professional should be engaged to directly supervise and assume responsibility for the client's engineering work.

2.2.2 SOFTWARE ENGINEERS

Software Engineers, as defined in these guidelines, are typically engaged in the design and implementation of commissioned Safety-Critical Software or components of the Safety-Critical Software.

The responsibilities of a Software Engineer typically include those listed below and the elements discussed in Section 3.0 Guidelines for Professional Practice.

Software Engineers should:

- specify, review, and refine Software requirements and Software designs based on inputs from clients;
- resolve (or participate in resolving) design trade-offs that affect the Safety-Critical Software;
- plan, manage, and undertake the implementation of Source Code and binary generation mechanisms (and related artifacts such build scripts);
- conduct, plan, and manage Software verification activities;
- conduct, plan, and manage analyses of the Software or related artifacts (for example, conduct a security analysis or perform a timing analysis for a real-time system);
- conduct, plan, and manage integration activities, both internal to the Software and with respect to deployment into the target environment;
- conduct, plan, and manage maintenance of Safety-Critical Software; and
- prepare assurance cases for the Safety-Critical Software.

Software Engineers must undertake to define the risks inherent in Software projects that they are involved with, and must make reasonable effort to oversee particular elements of the project at a level commensurate with the assessed risks.

Activities performed by this role may be delegated under the principle of direct supervision, as described in Section 4.1.3.

2.2.3 SOFTWARE DEVELOPERS

For the purposes of these guidelines, Software developers are typically engaged in the creation and partial verification of Source Code and Software binaries (and related artifacts).

In this role, Software developers should:

- participate in defining requirements and design specification(s);
- participate in implementing design trade-offs that affect the Safety-Critical Software;
- create Source Code and related artifacts;
- create unit tests to accompany the Source Code;
- create binaries or manage utilities that create binaries; and
- review Source Code (or related artifacts), utilities, or unit tests.

Individuals who actively create Source Code have a critical role in ensuring the Safety of Software. Individuals fulfilling the Software developer role may be highly specialized and have a deep knowledge of the technologies being employed (such as programming language, hardware interfaces, algorithms). As a result, they may make decisions about the function of the Software that cannot reasonably be expressed by Software requirements or design. Therefore, this role may contain work that must be directly supervised by a Software Engineer.

The title of "Software developer" might have other meaning(s) outside the context of Safety-Critical

Software projects. Consideration of additional activities associated with this title for other types of Software projects are beyond the scope of these guidelines.

2.2.4 SOFTWARE VERIFICATION

Verification of Software is concerned with ensuring the Source Code or generated Software binaries satisfy the specified requirements or design.

Individuals participating in Software verification should:

- participate in the planning and management of Software verification activities;
- create and review Software verification plans or specifications (such as test specifications), including choosing verification techniques and environments;
- configure, manage, and document Software verification environments and instrumentation;
- perform Software verification according to applicable specifications;
- prepare Software verification reports (and related artifacts); and/or
- participate in the creation of Software assurance cases related to Software verification and testing.

Note that Software testing is one of many possible verification activities that can be employed. Other techniques might include formal verification, code inspection, or static and dynamic analysis.

The verification of Software is a critical aspect of Software engineering for Safety-Critical systems. Modern verification techniques and related utilities and instrumentation are highly complex, and demand a deep level of knowledge and skill, so individuals who perform Software verification may be highly specialized. Furthermore, in a Safety-Critical context, inadequate Software verification could result in otherwise preventable defects being present in the final version of the Software. Therefore, this role may contain work that must be directly supervised by a Software Engineer.

2.2.5 SPECIALIST ROLES

Software engineering is a complex field with numerous speciality areas. It is not reasonable to expect an individual Software Engineer to have the level of competency demanded by some Safety-Critical applications across all specialties of Software engineering.

Indicators that the project involves a specialty area of Software engineering that might require engaging a specialist include:

- the speciality area is not considered part of a standard Software engineering undergraduate curriculum;
- the speciality area requires an advanced degree, professional certificate, and/or extensive practical experience to develop the prerequisite knowledge required to practice the subspecialty;
- the speciality area is an active area of research that requires practitioners to read specialized academic journals and regularly attend conferences to maintain competency; and/or
- the speciality area involves unique domain- or application-specific knowledge that limits the available resource pool.

Where possible, the Software Engineer should engage a specialist with an appropriate combination of skills, education, and experience. The specialist may then deliver a specific aspect of the engineering work specified by the Software Engineer.

It is permissible for a specialist to contribute to a project with an appropriate level of autonomy from the Software Engineer provided the following criteria are satisfied:

- The specialist has a combination of skills, education, and experience that is demonstrated by appropriate verifiable credentials and work history that is acceptable to the Software Engineer.
- The Software Engineer is able to define or agree to a clear scope and expectations for the work to be conducted by the specialist, including adherence to expected quality requirements and guidelines.

Ideally, this involves allocating explicit requirements to the specialist, and itemizing deliverables that will result from the specialist's work.

- The work undertaken by the specialist has a restricted scope, so the extent to which the work impacts aspects of the greater project can be clearly defined and understood by the Software Engineer. Any interface work performed between the subspecialty domain and the wider project is performed under the direct supervision of the Software Engineer.
- The findings and conclusions of the specialist's work are expressed using language that can be understood by an individual with a level of knowledge expected of a reasonable Software Engineer.
- Where relevant, the specialist provides compliance information regarding allocated technical requirements.
- The Software Engineer reviews the work of the specialist, based on a level of knowledge expected of a reasonable Software Engineer, and ensures compliance with applicable quality requirements and guidelines.

For example, a specialist with expertise in formal Software verification might be engaged to mathematically demonstrate that certain properties of the Software requirements, design, or implementation are satisfied. The output of such an analysis might be a report prepared by the specialist that describes the methods employed, provides detailed technical results, and summarizes (at a level that can be understood by a reasonable professional) the specialist's findings.

In many cases, industry interest groups or technical societies have been formed to address specific sub-specialities of Software engineering. For example, the Institute of Electrical and Electronics Engineers (IEEE) maintains a number of special interest groups on these topics. Software Engineers seeking a specialist might benefit from contacting these groups or societies.

3.0 GUIDELINES FOR PROFESSIONAL PRACTICE

3.1 OVERVIEW

This section describes the processes and activities required to meet the standard of practice for developing Safety-Critical Software. Broadly, the sections are presented according to Software engineering processes, Safety activities, security activities, and relevant external standards and guidelines.

3.2 SOFTWARE ENGINEERING PROCESSES AND LIFE CYCLE

Many Software engineering process models and project management methodologies exist (such as agile, scrum, or waterfall). Different process models can be applied successfully in different contexts, when properly selected and managed.

Rather than recommending a particular process model, these guidelines identify essential phases that are accepted within process models; any Safety-Critical Software development process must include consideration of these phases in some form.

3.2.1 PHASES OF SAFETY-CRITICAL SOFTWARE DEVELOPMENT

The following phases are identified in these guidelines:

- Elicitation of Software requirements
- Development of Software architecture
- Development of Software Source Code
- Generation of Software binaries
- Verification of Software
- Maintenance of Software

The selected Software engineering processes and life cycle management approaches should be considered within the context of a larger system's engineering effort.

Each phase is discussed below, and outlines recommended activities associated with the particular phase. Note that these guidelines do not aim to be a complete reference for Software life cycle phases. Instead, they capture a minimum set of recommendations for each phase. Software life cycle phases may execute consecutively or concurrently, and combinations thereof, as appropriate. Some of the phases as described may be omitted, may not apply, or may be combined and adapted, based on the details of a particular project. However, the overall Software engineering principles should still apply. Practitioners may employ additional activities or techniques based on company policies or industry standards.

3.2.1.1 Elicitation of Software Requirements

Fundamentally, requirements describe what the Software must do, so the elicitation of Software requirements represents an important phase of any Software project. However, in many Safety-Critical systems, Software forms only part of the functionality of the overall system. Therefore, it is also important that system-level requirements be defined and then decomposed, where appropriate, to create lower-level Software requirements. It should be noted that not all system requirements are Safety requirements.

Requirements usually refrain from describing the internal design of the system, in that the requirements view the design as a black box and make statements about inputs and outputs. However, the development of

some design, such as the high-level architecture of the system, is usually a prerequisite for defining requirements. In this context, the definition of requirements and design of some aspects of the Software are essentially coupled. Iteration between the requirements and design phases is acceptable, provided changes to the requirements and design are considered in a systematic manner and all changes are documented.

During this phase of the Software engineering life cycle, high-level security-Threat modelling should be performed to identify and understand possible avenues of attack for the system. Likewise, high-level Safety-analysis activities should identify any Hazards and the functions required to mitigate those Hazards. Requirements that address any identified Threats and Safety mitigations should be included with the system and Safety requirements.

Requirements for functionality, Safety, quality, and security cannot always be decoupled and must be defined simultaneously. Software Engineers should be aware of the shared roles of such requirements.

Software Engineers should ensure Software requirements have the following properties:

- Each requirement is assigned a unique identifier.
- Each requirement is stated precisely to avoid ambiguity; this might include using formal or semi-formal notations for describing the requirements; for example, by using boilerplate requirements, templates, or controlled natural language.
- Where possible, requirements are expressed as atomic statements. That is, requirements do not use logical connectors to join statements that could otherwise be expressed as individual requirements.
- Requirements are envisaged in the context of the entire system and, where applicable, developed with input from stakeholders.
- Each requirement is verifiable, meaning that a verification test procedure (manual or automated) can be performed to verify the requirement is satisfied by the Software.
- A systematic process is in place to manage and track changes to the Software requirements baseline and to flow changes on to the Software development and verification phases.

ISO/IEC/IEEE 29148 Systems and Software Engineering – Life Cycle Processes – Requirements Engineering provides a process standard that describes the requirements engineering process.

3.2.1.2 Development of Software Architecture

Software architecture describes the structure of a piece of Software. Architecture exists at all levels of Software design, from high-level conceptual definitions to detailed definitions of Source Code elements.

During development, some level of abstraction is used to conceptualize a Software architecture, which allows Software Engineers to reason about the structure of the Software without becoming overwhelmed by details.

At minimum, Software Engineers should ensure these practices are followed when developing Software architecture:

- All elements of the Software architecture are traceable to at least one requirement, and elements are clearly marked with the requirement(s) from which they are derived.
- The Software architecture is expressed in a systematic and precise form; the use of semi-formal notation (such as UML) is highly recommended.
- An appropriate level of abstraction is selected for the Software architecture description.
- The Software architecture considers multiple perspectives, so the static structures and dynamic behaviours of the Software are comprehensively described.
- The Software architecture considers interfaces with other systems and/or hardware elements, including the systems into which, or the hardware onto which, the Software will be deployed.

- Quality attributes are described that are important for the Software to fulfill and establish how the architecture addresses the quality attributes.
- Well-known design patterns and/or approaches are considered, where appropriate; for example, a monitor component utilizing a heartbeat signal is commonly used to assess the availability of a system.
- Programming languages (and other relevant Software technologies) are determined during Software architecture development, and appropriate programming languages and utilities are chosen for the system.
- A systematic version control system is used to manage and track changes to Software architecture.
- The completed Software architecture undergoes assessment to confirm that defined Safety and security requirements are met, and that new vulnerabilities or Hazards have not been introduced during the architecture development phase.

3.2.1.3 Development of Software Source Code

Development of the Software Source Code focuses on translating the requirements and architecture into machine-interpretable instructions, typically using a programming language. In some cases, executable models or specifications may be used (for example, MATLAB©) to automatically generate Source Code from a detailed design, provided that in the judgment of the Software Engineer the tool proposed for use allows these guidelines to be followed.

It is common practice to create a Software development plan to define project-specific processes. At minimum, Software Engineers should ensure these practices are followed when developing Source Code:

- A programming language that is suitable for the application is chosen, and the impact of programming language constructs is considered (for example, typing, decomposition mechanisms, control flow).
- A relevant coding standard (such as MISRA C or SEI CERT C) is used to standardize Source Code conventions and reduce the likelihood of defects being created during Source Code development. Such a standard includes both stylistic and code best practices (for example, initialization of variables to known value before use). Any deviations from industry coding standard are documented, if applicable.
- All Source Code is traceable to an architectural element defined by the Software architecture and can ultimately be traced back to a Software requirement, and Source Code that can not be traced to an architectural element and/or a requirement is removed.
- Source Code reviews and inspections are conducted by qualified individuals using a systematic approach. Where relevant, static analysis tools may be used to analyze large amounts of Source Code. A systematic inspection methodology might then focus on inspecting notifications raised by the static analysis tool(s).
- Configuration data and files are considered to be Source Code regardless of the language in which they are written (for example, an XML file with Software configuration data that is consumed by a C program is considered Source Code).
- A systematic version-control system is used to manage and track changes to Software Source Code.

3.2.1.4 Generation of Software Binaries

Once the Source Code has been developed, typically one or more binaries are generated that will be executed on a specific hardware platform. This step is conducted by one or more additional Software tools, such as a compiler or an assembler.

In cases where Source Code is interpreted (rather than compiled) by another piece of Software, the guidance in this section might not directly apply. Therefore, Software Engineers working with interpreted Source Code should consider the following guidance and may adapt it, as required.

Software Engineers should ensure these practices are considered, when generating Software binaries from Source Code:

- An argument is made for the acceptability, quality, and integrity of the tool that will be used to generate binaries; where possible, only qualified tools are applied (meaning those that meet relevant Software quality standards).
- The availability of high-quality tools used to generate binaries is considered when choosing a programming language(s) for developing the Software Source Code.
- Software artifacts related to the binary generation, such as configuration files or build scripts/utilities, are treated with the same care and rigour as the primary Source Code.
- Tools related to generation of binaries are clearly described in the Software documentation, which includes the names and versions of the tools, the scripts and utilities used, and the environment (such as an operating system) within which the tools must be deployed.
- Repeatability and independence of the binary generation process is considered; this may be facilitated by using automated build utilities.
- Binaries are uniquely identified and labelled, so their source inputs are noted and the binary's use in testing activities and subsequent distribution to other project parties and/or the client is traceable.

In some applications, input data (as well as a binary) dictate the Software's behaviour. It is possible to have very simple Source Code for the Software along with data that significantly impacts the correct function of the Software. For example, the schedule for a traffic light (data) could be fairly complex, yet the Source Code controlling the traffic light might be very simple. Consequently, an error in the input data could contribute to an Accident. In these scenarios, per Section 3.2.1.3 Development of Software Source Code above, the input data should be treated as part of the Software itself, and be subject to an appropriate engineering process.

3.2.1.5 Verification of Software

Verification of Software focuses on ensuring the Software will behave as specified in the requirements and interface control documents. Verification techniques may be applied during all life-cycle phases. At minimum, verification activities should focus on substantiating the outputs of the requirements, architecture, and Source Code phases of the life cycle.

Software Engineers should ensure these practices are followed with respect to verification of Software:

- A verification plan is created that details the verification techniques to be applied at each phase of the Software life cycle. Deviations from this plan are documented along with a rationale for the deviation.
- Verification is conducted on the requirements to ensure they are free of conflicts, complete, and consistent. Requirements may be verified using systematic inspection or design review, semi-formal techniques, and/or formal techniques (for example, application of formal logic).
- Verification is conducted on the Software architecture to ensure it is free of conflicts and is complete and correct with respect to the requirements. The architecture may be verified using systematic inspection or design review, semi-formal techniques, and/or formal techniques.
- Software Source Code units (for example, a function, method, procedure, or class) are tested by exercising the Source Code on multiple diverse inputs to ensure the unit's Source Code fulfills its designed intent.
- Integration testing is used to verify collections of many units to ensure the units collectively fulfill the designed intent and requirements.
- A set of tests of the Software Source Code exists for each functional requirement of the Software to ensure the requirement was successfully implemented.

- Formal Source Code verification methods (for example, Hoare logic) may be applied to gain additional confidence in the correctness of the Source Code.
- Where applicable, hardware-in-the-loop testing is used to determine that the Software behaves as required on the hardware upon which it will be deployed.
- The statement and branch coverage of Source Code tests is measured using an appropriate metric; a high level of coverage is desirable. The acceptable level of coverage is based on the requirements for the application.
- Records of all verification activities are kept, which include a description of the verification conducted and relevant results.

Often, the tasks of binary generation (see Section 3.2.1.4) and Software verification are combined using continuous integration or continuous deployment methods.

3.2.1.6 Maintenance of Software

Software in Safety-Critical systems might have a long lifetime. A particular version of the Software will not age in the traditional sense typically ascribed to mechanical or electrical components. However, Software may become outdated for a given application if the Software's environment changes. Additionally, changes made to fix defects discovered after deployment might introduce additional defects.

To address this, Software Engineers must consider how Software for Safety-Critical systems will be maintained and evolve over time, and should ensure these activities related to the maintenance of Software after deployment are followed:

- A maintenance plan is created that addresses
  - how the Software's health will be monitored over the course of its lifetime;
  - how defects that arise during use of the Software will be addressed, including prioritizing defects and appropriate responses;
  - with respect to elements provided by third parties (such as operating systems or open-source libraries), how available updates, in-service bulletins, and obsolescence are assessed and managed;
  - how redeployments or updates of the Software will be handled, including change verification and required regression testing (note that changes to Software support tools, such as compilers, FPGA generation tools, simulators, or test data generators, may affect the Software generation and test environment, so changes to tools are considered part of Software change management, including required regression testing and Safety re-tests); and
  - how to address discovered security vulnerabilities in a manner that does not impact the operation of Safety-Critical components.
- A decommissioning plan should be created that addresses scenarios where the Software is removed from use. Decommissioning plans should include methods for destroying sensitive material that could impact the security of the system (for example, key material, sensitive configuration data).

3.2.2 USE OF THIRD-PARTY SOFTWARE ARTIFACTS

Not all Software included in a Safety-Critical Software project is directly developed by the project team. This section discusses the use of third-party Software artifacts within Safety-Critical Software projects. The topic of third-party Software artifacts is relevant to many of the other topics discussed in Section 3.0 Guidelines for Professional Practice.

For a specific project, Software Engineers should consider how the guidance in this section can be integrated with the relevant processes and life cycle phases described in Section 3.2.1 Phases of Safety-Critical Software Development above.

Early in a project, Software Engineers should identify which functions of the Safety-Critical Software (or supporting infrastructure) will be fulfilled by third-party Software artifacts. This might include Software that is part of the final binary (such as libraries), Software that provides services or an environment (such as an operating system), or Software that handles the binary (such as a compiler).

When Software Engineers rely on third-party Software artifacts as part of a Safety-Critical system, they should use their judgment to determine whether the Software is suitable for the application.

3.2.2.1 Types of Third-Party Software

Third-party Software may generally be divided into two categories:

1. Software of unknown provenance: Software that is widely available but was not developed with the intent of being incorporated into Safety-Critical Software or for which adequate engineering records do not exist.
2. Off-the-shelf Software: Software that already exists and was created with the intent of being incorporated into Safety-Critical Software but for which the Software Engineer cannot directly claim professional responsibility. This includes commercial off-the-shelf Software.

Furthermore, Software of unknown provenance or off-the-shelf Software might be either proprietary (closed source) or open source, and might also have been developed by a vendor or by a community group. Note that many permutations of these factors are possible. For example, it is possible to have open-source Software of unknown provenance that was developed by a vendor.

In some cases, off-the-shelf Software might include a certificate demonstrating compliance with a particular standard. Where appropriate, such certificates may be used as evidence that a rigorous process was used to develop the Software. However, not all standards are sufficiently rigorous for Safety-Critical Software applications, so Software Engineers should use their judgment when evaluating the applicability of certified off-the-shelf Software for use in a specific Safety-Critical application.

3.2.2.2 Assessing the Suitability of Third-Party Software for Safety-Critical Software Projects

When considering third-party Software artifacts for use within a Safety-Critical Software project, Software Engineers should ask the following questions:

- Will the third-party Software fulfill its intended purpose as part of the Safety-Critical Software?
  - This could be established by reviewing the Software's documentation, reviewing evidence of prior use in Safety-Critical (or similar) applications, and conducting systematic testing.
- What is the potential impact of the third-party Software artifacts on the integrity of the Safety-Critical Software?
  - For example, if the third-party Software malfunctioned, would it contribute to the occurrence of a Hazard. This is particularly relevant for Software of unknown provenance, which should only be integrated with Safety-Critical Software with great care.
- If it is difficult to directly assess the quality of the third-party Software, what indicators can be reviewed to determine the suitability of such Software for use in a particular project?
  - The following indicators can be reviewed when assessing third-party Software:
    - Vendor and/or community credibility: For example, for open-source Software, this can include looking at the community organization structure, in particular whether there is a well-defined process for contributing to the community.
    - Vendor or community compliance with established quality or Safety standards (for example, ISO 26262 Road Vehicles – Functional Safety, for automotive Software).

- Maturity of the third-party Software and history of the Software's use in other projects.
    - Availability of Software documentation (for example, requirements or application programming interface [API] specifications, test procedures).
    - For open-source Software, accessibility of the Source Code and ability to conduct static or dynamic analysis and verification. Such evaluation is typically not possible for proprietary (closed source) Software.
    - Independent testing of the Software or library for existence of security defects.
- What long-term support and maintenance of the third-party Software artifacts is available?
  - This is particularly relevant for community-supported Software, as it might not be possible to arrange support contracts (or similar support) with the overall community.
- Have the license terms under which the third-party Software artifact is provided understood and appropriate for the intended use?
  - Many Software licensing models can affect how a third-party Software artifact is used in a specific project. For example, some open-source licenses prohibit commercial use of the Software artifacts, while others may require that project-specific additions to the third-party Software be contributed back to the development community.
- What methods will be used to incorporate the third-party Software artifacts into the infrastructure (for example, build process, version control) of the Safety-Critical Software project?

3.3 SAFETY ENGINEERING FOR SAFETY-CRITICAL SOFTWARE

Safety engineering is a sub-discipline of systems engineering and may be conducted entirely independently of Software engineering. However, in the modern context, Software Engineers may often perform tasks that would be more typically performed by systems engineers. This is partly due to the role that Software plays in controlling the overall behaviour of complex systems.

Therefore, it is important for Software Engineers to have a working knowledge of Safety engineering techniques and to understand how to apply those techniques or engages specialists to do so when developing Software for Safety-Critical systems.

This section describes the following Safety engineering activities and techniques that may be employed by Software Engineers working on Safety-Critical systems that incorporate Software:

- Hazard analysis
- Risk and criticality analysis
- Reliability engineering
- Safety cases

3.3.1 HAZARD ANALYSIS

Hazards, as defined in these guidelines, are sets of conditions or an operational situation that might lead to an Accident. Hazard analysis in Software development is an iterative process used to assess Risk and identify different types of Hazards.

These guidelines make the following recommendations regarding Hazard analysis:

- A Hazard analysis that employs an appropriate selection of systematic techniques should be conducted for Safety-Critical Software.
- Hazards should be identified at an appropriate level of abstraction (for example, based on system requirements).

- Causal Factors that contribute to the occurrence of a Hazard should be identified.
- The scope of the system being analyzed should be clearly identified prior to determining Hazards or Causal Factors.
- If applicable, Hazard analysis should include the study of human-machine interfaces to identify related Hazards or Causal Factors.
- Hazards and Causal Factors should be clearly described and documented. It is recommended that a standard template be used to document the Hazard analysis.
- Once Causal Factors are identified, control measures should be designed to mitigate possible occurrences of the Hazard due to the Causal Factors.

Note that Hazard analysis is an iterative process, as changes made during the definition, development, and implementation process may effect changes to failure modes and Hazards that need to be accounted for.

3.3.1.1 Identifying Hazards and Causal Factors

Hazards are typically identified at an abstract or black-box level of system abstraction (see Section 3.2.1.1 Elicitation of Software Requirements); the level of abstraction should be sufficient to describe overall behaviour of the system in question and usually concerns the interaction of the system with its environment.

Hazards form the basis of Safety engineering efforts and should therefore be:

- identified as early as possible in the engineering process;
- stated clearly and unambiguously; and
- reviewed frequently to ensure they are complete as designs evolve.

Causal Factors, as defined in these guidelines, are actions, omissions, events or conditions that contribute to the occurrence of a Hazard. Importantly, Causal Factors are often distinct from the Hazards themselves, and many different Causal Factors may contribute to the occurrence of a single Hazard. For example, in a cardiac pacemaker system, a Hazard might be that "the device delivers an unsafe amount of energy to the surrounding tissue," while a Causal Factor for this particular Hazard might be that "the Software controller experiences an integer overflow when calculating the energy value."

3.3.1.2 Techniques for Hazard Analysis

Several systematic techniques exist for determining how Causal Factors might contribute to the occurrence of a Hazard. Common techniques include the following:

- Failure modes and effects analysis (FMEA): A bottom-up method that focuses on determining if and how failures (Causal Factors) can result in the occurrence of a Hazard. See Ericson (2015).
- Fault tree analysis (FTA): A top-down method that assumes a Hazard has occurred and works backwards to determine what might have caused the Hazard to occur. See Ericson (2015).
- Hazard and operability study (HAZOP): A method for determining Causal Factors using a set of guide words to identify failure modes, which originated in chemical and process engineering. See Ericson (2015).
- System theoretic process and analysis (STPA): A method that combines control-theoretic modelling and a set of guide words to identify failure modes due to inadequate control. See Leveson (2018).

Hazard analysis techniques have both strengths and weaknesses, and may provide only one specific perspective of the system. Software Engineers should select an appropriate combination of techniques to establish a reasonable degree of confidence that the resulting understanding of the Hazards is sufficiently comprehensive. In many scenarios, more than one analysis technique may be required.

Some techniques, such as FTA, have variations or extensions where likelihood values are assigned to each event or failure that might occur. In many contexts, such as mechanical system design, probabilities are readily derived based on

experimentation and known properties of devices and materials. However, in other contexts, such as Software development, probabilities might be difficult to determine. In such cases a qualitative analysis may be applicable. An appropriate technique should be selected such that the results are both comprehensive and meaningful. One example of such a technique is Markov analysis (Ericson 2015).

Techniques such as FMEA and FTA focus on the identification of failures that may lead to a Hazard. These techniques may be used independent of any Hazard definition (provided what constitutes a failure is understood). However, not all failures in a system will lead to the occurrence of a Hazard.

3.3.1.3 Example of Hazard Identification

Following is a short fictitious example of a Hazard, with a number of Causal Factors and suggested Safety control measures. This example is intended to illustrate the differences between Hazards and Causal Factors, as discussed above.

1. System and assumptions:

- Consider an electronic brake controller (EBC) in an automobile.
- Assume that all brake commands are received by the EBC, which, in turn, uses a Software routine to coordinate the application of the brakes to the wheels of the vehicle.
- The system under investigation is the EBC itself, including the system-on-chip (SoC) hardware running the Software, the real-time operating system (RTOS), and the brake control Software application.
- The EBC system receives inputs from either a vehicle controller (for example, cruise control) or directly via the brake pedal.
- This simple architecture is depicted in Figure 1. Blue blocks indicate Software elements, purple blocks indicate hardware elements, and grey blocks indicate the components are outside the scope of the system. Arrows represent data flow.

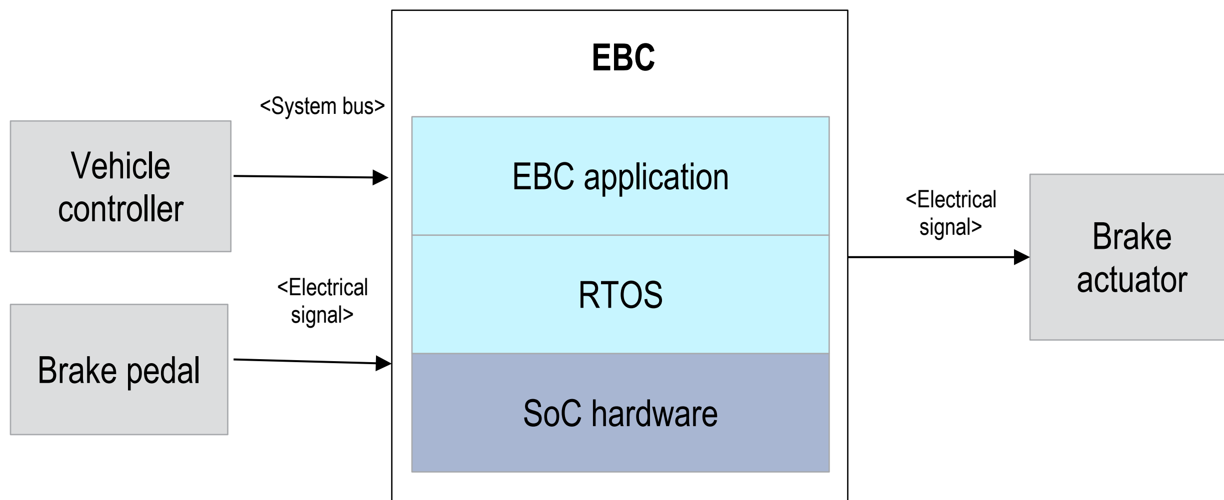

Figure 1: Block Diagram of an Electronic Brake Controller System

NOTES:
EBC = electronic brake controller; RTOS = real-time operating system; SoC = system-on-chip

2. Hazard identification:

- For the system illustrated in Figure 1, one Hazard might be stated as follows:
  - Hazard 1: The EBC system sends a braking signal to the brake actuator without appropriate stimulus from either the brake pedal or vehicle control components.

3. Causal Factor identification:

- By applying the various techniques described above (such as FTA), the following Causal Factors for this Hazard may be identified (these are examples, not a complete list):
  - Causal Factor 1: The SoC experiences a hardware failure in the RAM unit that corrupts memory relied upon by the RTOS or the EBC application.
  - Causal Factor 2: The RTOS incorrectly schedules the signal-emitting process within the EBC application.
  - Causal Factor 3: The EBC application experiences a deadlock between two critical processes immediately after sending a braking signal to the actuator, and subsequently fails to disable the braking actuator.

4. Control measure design:

- Once the Hazards and Causal Factors are identified, additional control measures that could mitigate the Causal Factors should be designed.
- These control measures are often expressed as requirements. For example, Causal Factor 1 above might be mitigated by the following:
  - Safety Requirement 1: For each critical piece of data, the EBC algorithm will store a checksum, then compare the value of the stored checksum to a checksum computed for the data retrieved from memory prior to using the data.

3.3.2 RISK AND CRITICALITY ANALYSIS

Risk is typically defined as a combination of two factors:

1. The severity of an anticipated Accident resulting from a Hazard; and
2. The likelihood of a Hazard occurring and leading to Accident (alternatively referred to as “exposure”).

Given the Hazards for a particular system, it is possible to determine a level of Risk associated with the Hazards. The Risk can then be used to establish the criticality of each Hazard and the corresponding engineering efforts associated with mitigations for Hazards.

These guidelines make the following recommendations regarding Risk and severity assessment:

1. A Risk and/or severity analysis should be conducted for Safety-Critical systems that incorporate Software.
2. The Risk and/or severity should be identified and documented for each Hazard.
3. The method of Risk and/or severity determination should be appropriate for the system in question such that results are meaningful.

Many methods for calculating Risk exist. In cases where likelihood can be determined as a probability (numerical), and the severity of an Accident can be quantified (for example, environmental damage, human lives directly affected), it may be acceptable to conduct a purely numerical Risk calculation.

However, for some systems (including systems that incorporate Software), probabilities of Hazards occurring and/or the associated losses are difficult to meaningfully quantify. In such cases, an ordinal scale may be appropriate.

Many original Risk and severity scales exist and are often included in industry-specific standards. A sample ordinal Risk and severity table is shown in Figure 2 below, and corresponding definitions of Risk likelihood and severity categories are listed in Table 1 below.

LIKELIHOOD					
	4	4A	4B	4C	4D
	3	3A	3B	3C	3D
	2	2A	2B	2C	2D
	1	1A	1B	1C	1D
		A	B	C	D
		SEVERITY			

Figure 2: Sample Risk and Criticality Table

Table 1: Definitions of Risk Likelihood and Severity Categories

LIKELIHOOD		SEVERITY	
1	Highly unlikely or very low probability	A	No harm or negligible harm to persons or the environment
2	Unlikely or low probability	B	Minor injuries to persons, or minor harm to environment
3	Occasional or medium probability	C	Major injuries, possible death of persons, or major harm to environment
4	Certain or high probability	D	Certain death to one or more persons, or catastrophic harm to environment

3.3.3 RELIABILITY ENGINEERING

Reliability is a property of a system related to its ability to carry out its intended function, whereas Safety is a property of a system that indicates that the system is free from unacceptable risk of an Accident occurring due to non-malicious causes.

These properties are not equivalent and may in fact be in conflict. Systems might be safe but unreliable (for example, an airplane that never leaves the ground) or reliable but unsafe. However, reliability and Safety are often concurrently desirable properties of Safety-Critical Software, which must maintain a minimum level of Safety at all times.

There are many techniques and analysis methods available that focus on improving Software and system reliability. Reliability engineering techniques and methods may be applied to Safety-Critical Software projects, with the aim of reducing the Risk associated with the occurrence of Hazards.

Reliability engineering is a large field, so the following techniques should not be considered a comprehensive list, but may be used as starting points for analyzing and improving the reliability of Safety-Critical Software:

- Apply Hazard analysis techniques (such as FTA or FMEA) with a focus on reliability (rather than Safety), to identify failures that could compromise reliability.
- Add redundant elements, such as arbitrated redundant computation or N-version design and programming, to improve reliability.
- Use robust error detection and handling mechanisms; for example, exception-handling structures in programming languages, to improve reliability.
- Use periodic backups or state checkpoints, which would may improve reliability by allowing the system to return to a known state if a failure occurs.

- Apply certain system and Software design patterns to improve reliability; for example, isolating high-reliability critical functions into a single high-integrity component.

3.3.4 SAFETY CASES

A Safety case (also referred to as an assurance case) is an evidence-based argument for the Safety of a system. All Safety-Critical systems have an inherent level of Risk associated with their application; for example, the Risk inherent in the achievement of system objectives. The aim of a Safety case is to argue that inherent Risk has been reduced and any Residual Risk is acceptable for the intended application.

In addition to their primary purpose (of arguing for Safety), Safety cases are also useful for supporting certification and regulatory efforts. First, they organize all of the Safety-related information into one location for certification authorities, which reduces effort associated with system review. Second, they provide certification authorities with concise descriptions of why the Software Engineers building the systems believe them to be acceptably safe.

A Safety case is composed of two fundamental elements:

1. An argument regarding the acceptability of Residual Risk in a system
2. Evidence to support claims or assertions made by the argument

Fundamentally, without appropriate evidence to substantiate claims, no argument about Safety can be made. Many records may be considered evidence; for example, verification records and reports, design documents, mathematical analyses, or design inspection and review reports authored by qualified individuals.

Safety cases are inevitably subject to bias, so Software Engineers creating Safety cases should carefully examine their argument(s) to identify the underlying assumptions and avoid bias. One approach to reducing bias is to attempt to prove the system is in fact not safe, and then determine if the associated Risk is acceptable.

These guidelines make the following recommendations regarding Safety cases for Safety-Critical systems incorporating Software:

- A Safety case should be created and documented for each Safety-Critical system.
- To reduce bias, Safety cases may make statements about the acceptability of Residual Risk rather than attempting to prove Safety.
- Evidence used to support claims in a Safety case should be well documented and of reliable provenance.
- Notations for structuring arguments, such as goal-structuring notation (GSN), are not required, but may be used to visualize Safety arguments and provide supporting evidence.

Further guidance regarding Safety (or assurance) case development is in ISO/IEC/IEEE 15026:2019 – Systems and Software Engineering – Systems and Software Assurance (ISO 2019).

Figure 3: Sample Safety Case Argument Using Goal-Structuring Notation below shows an example of a GSN-based Safety case that builds on the previous example of a Hazard identification for an EBC in Section 3.3.1.3 above. In Figure 3, the Safety of the control Software for the EBC is demonstrated by arguing that all identified Hazards are mitigated by the requirements, design, and implementation. Evidence is used to support claims (expressed in GSN as goals) that the requirements are correct, complete, and satisfactorily implemented by the Software. Only one Hazard (called "inaccurate braking force applied") is shown.

This example demonstrates a GSN argument syntax. It should not be considered a complete Safety case for an EBC system; in actual practice, a Safety case for an EBC Software system would be much more complex.

Other GSN-based examples, and guidance on syntax, semantics, and style for GSN Safety cases is available in the GSN Community Standard – Version 2 (SCSC 2018).

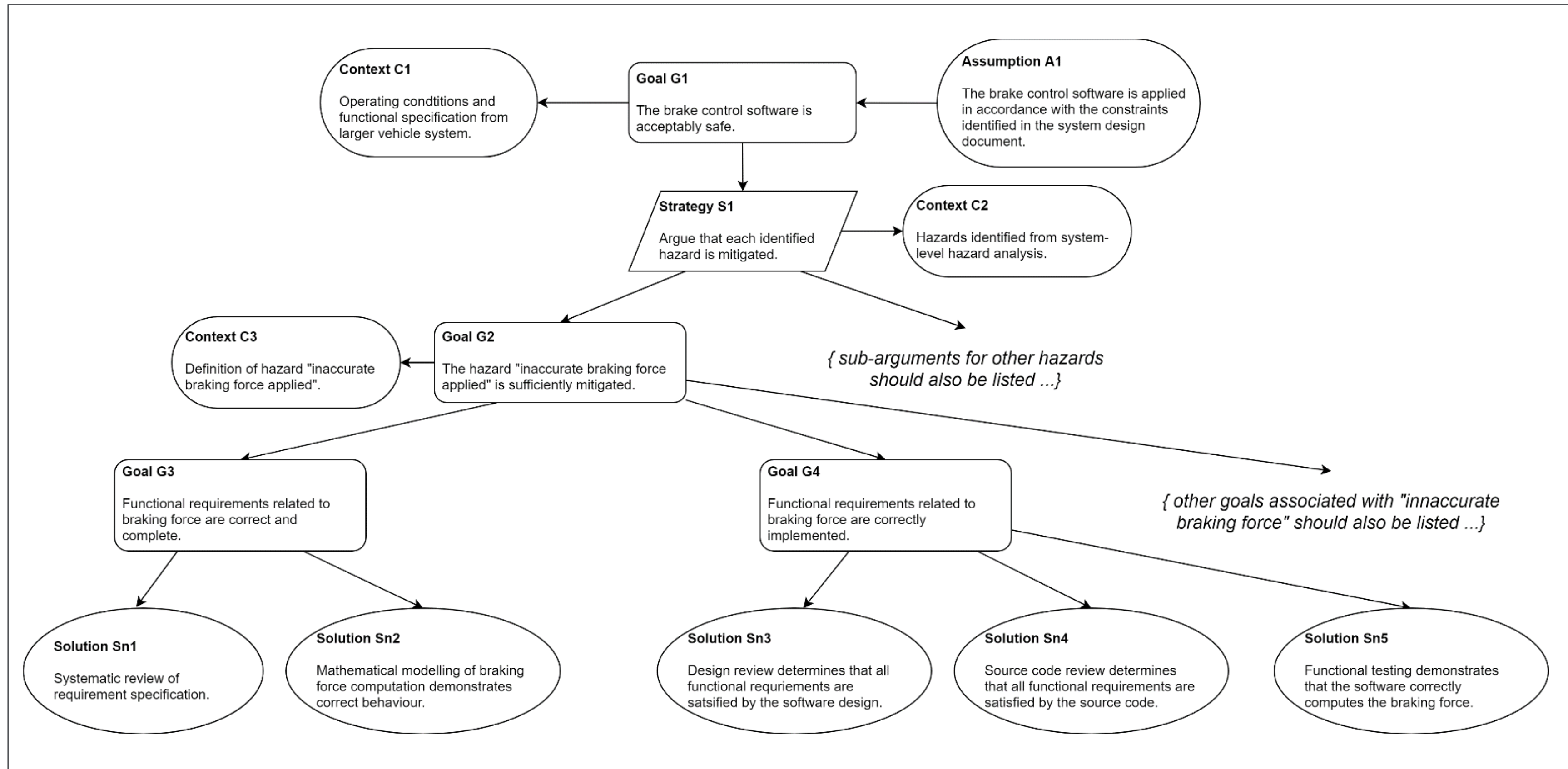

Figure 3: Sample Safety Case Argument Using Goal-Structuring Notation

3.4 SECURITY ACTIVITIES FOR SAFETY-CRITICAL SOFTWARE

Security engineering is a specialty engineering field that is focused on the design and implementation aspects of digital systems that addresses the potential for inadvertent misuse or malicious behaviour.

Therefore, it is important for Software Engineers to have a working understanding of security engineering techniques and to understand how to apply those techniques or engage specialists to do so when developing Software for Safety-Critical systems.

This section describes the following security engineering activities and practices that may be employed by Software Engineers and specialists working on Safety-Critical systems:

- Security Risk and Threat analysis
- Security controls and policies
- Security verification and validation
- Security assurance cases
- Assessment of third-party libraries

Traditionally, security-critical[1] and Safety-Critical terminologies are defined separately and vary across different industries. However, with the evolution of Internet- and network-connected critical Software, these terminologies and subject matter are often used interchangeably. However, existing standards provide limited guidance for when Safety-Critical Software and security-critical Software are combined. In the context of this document, the security of the Software is considered critical to the extent required to mitigate security-related Risk in support of the system Safety case(s).

Several security standards are available for reference (see Section 3.6 Relevant External Standards and Guidelines). Note that these standards are usually oriented towards enterprise systems rather than industrial systems (for example, manufacturing, process control, transportation).

When undertaking work with Safety-Critical Software, Software Engineers should consider guidance from relevant security standards, so they can make informed decisions about the application of appropriate security engineering techniques. Security standards provide foundational guidance for building Safety-Critical Software, but the guidance should be applied with care to avoid affecting the reliability of the Software.

Software Engineers should also employ security engineering practices to reduce the likelihood that malicious agents could contribute to the occurrence of a Safety Hazard. This section describes a collection of these practices.

The following list of activities and practices is not comprehensive, but represents the minimum standard of practice that Software Engineers should follow when developing Safety-Critical Software.

3.4.1 SECURITY RISK AND THREAT ANALYSIS

In security engineering literature, the term "threat" is used rather than "Hazard." However, in these guidelines, these terms are considered the same and are used interchangeably. When assessing the security of Safety-Critical Software, Software Engineers should consider how security vulnerabilities or a malicious agent might contribute to the occurrence of a Hazard.

A security Risk analysis provides Engineering Professionals with an understanding of the Risk associated with Threats. For Safety-Critical Software, a security Risk analysis should be conducted that mirrors the Safety Risk analysis described in Section 3.3 Safety Engineering for Safety-Critical Software.

Briefly, a security Risk analysis should determine the likelihood that a Threat will occur, and ascertain the severity of that Threat. In this context, the goal of a security Risk analysis would be to understand any

[1] Literature definitions of "security-critical software" often refer to data associated with the Software being sensitive to loss or theft for reasons of privacy, property, and economic loss. While this may be a concurrent concern for some Safety-Critical Software, such concerns are not a focus of this document.

additional Safety Risk associated with the security vulnerabilities and the potential impact of malicious agents. The resulting level of Risk should be used to motivate additional security measures.

Once the level of security Risk has been established, a Threat analysis may be used to identify vulnerabilities. Many techniques for analyzing Threats exist, and are similar to the techniques discussed in Section 3.3.1 Hazard Analysis. The primary difference in a security context is the existence of a malicious agent.

Common techniques for conducting Threat analyses include the following:

- Threat modelling: A systematic approach to identifying and enumerating Threats to a system. Threats can then be addressed and mitigated by adding security requirements. See Shostack (2014) for additional details.
- Attack tree analysis (ATA): A systematic top-down method of identifying vulnerabilities and malicious actions that constitute Threats. This approach is similar to the fault tree analysis (FTA) discussed in Section 3.3.1.2 Techniques for Hazard Analysis.
- System theoretic process and analysis for security (STPA-Sec): A security adaptation of a Safety-focused STPA Hazard analysis method. See Young (2014) for additional details.

During a Threat analysis, the following aspects of information security should be considered:

- Confidentiality: Information regarding the system's design and operation is controlled and distributed on a need-to-know basis.
- Integrity: The data used by the Software to complete Safety-Critical functions is accurate and trustworthy.
- Availability: The system is capable of completing its specified Safety-Critical functions.

The output of a Threat analysis should be a set of system vulnerabilities that are not already mitigated by the system design.

3.4.2 SECURITY CONTROLS AND POLICIES

Once vulnerabilities have been identified, Software Engineers should design security controls and policies to prevent malicious agents from exploiting the vulnerabilities. In cases where prevention is not possible, a combination of controls and policies aimed at detection and recovery should be employed.

Controls and policies might include the following:

- Designs that limit access to critical system controls (such as network isolation)
- Data backup procedures (such as nightly backup to offsite locations)
- Data integrity measures (such as checksums)
- Data protection measures (such as encryption)
- Software measures (such as validation of user inputs)
- Network security measures (such as firewalls)
- Access control and authentication mechanisms (such as passwords or 2FA)
- System monitoring mechanisms (such as SIEM or automated monitoring systems)
- Physical access controls (such as locked doors or air-gapped systems)
- Operational policies (such as user access management policies or documented system recovery policies)
- Human resources policies (such as termination procedures with respect to access to information)
- Security training policies (such as policies surrounding the frequency and content of training)
- Secure Software development life cycle policies (such as secure development practices or reduction of security defects in finished code)
- Patching policies (such as how systems and Software will be updated to remediate security defects)
- Vulnerability management policies (such as how Software security defects will be identified and addressed)

The designed security controls should be expressed as requirements, to ensure they are fully incorporated into the Software engineering process as discussed in Section 3.2 Software Engineering Processes and Life Cycle.

3.4.2.1 Control of Remotely Accessible Systems

Software Engineers should consider the security implications of functions that permit remote access or control. Systems permitting remote access have an increased attack surface that may expose a large number of vulnerabilities to a malicious agent. In highly critical systems, the Risk associated with remote access functionality might be intolerable.

In some cases, it might be possible to isolate the highly critical functions and/or components and provide additional security controls. For example, a remote access system for a chemical processing facility might include network isolation via a virtual private network (VPN), where network control policies dictate that users have “read-only” rights when accessing the network remotely, and must be physically on-site for “write” operations to be allowed on the processing Software and/or hardware.

Alternatively, the security of communications can be increased by tightly controlling access to systems by applying restrictions on communication origins and timing, authentication methods, permissible commands, and automated behavioural analysis.

3.4.3 SECURITY VERIFICATION AND VALIDATION

Software Engineers should engage in validation activities to establish the adequacy of the implemented controls and policies. This might include actions to:

- conduct audits for compliance to existing security standards and guidelines;
- perform independent adversarial system testing;
- use testing tools to identify common security defects;
- confirm the traceability of requirements, controls, and policies through the design and implementation of the Safety-Critical Software; and
- verify Software security requirements.

3.4.4 SECURITY ASSURANCE CASES

Security assurance cases present an evidence-based argument for the security of a system. Security assurance cases may be viewed as an extension of Safety cases; accordingly, the guidance in Section 3.3.4 Safety Cases also applies in the context of security assurance cases.

Where applicable, Software Engineers should create a security assurance case for Safety-Critical Software. Usually, the Safety case and the security assurance case can be combined to form one argument.

3.4.5 ASSESSMENT OF THIRD-PARTY LIBRARIES

Third-party libraries should be assessed to determine if they contain security vulnerabilities, before being chosen for inclusion in a system.

Software Engineers should assess third-party libraries for the following:

- Source (for example, location from which the library was downloaded)
- Existence of prior independent testing or assessment of security controls
- Conformance to secure Software engineering standards
- Additional referenced or included libraries (for example, libraries that are referenced or included within the library being assessed)

3.5 OBSERVATION OF DEFICIENCIES

When providing Software engineering services, Engineering Professionals may become aware of a significant deficiency in other aspects of the Software or the larger project that involves the practice of professional engineering.

In such instances, the Engineering Professional must act in a way that is consistent with the intent of the Association's Code of Ethics, Bylaw 14(a). Principle 9 of the Code of Ethics requires Engineering Professionals to "report to their Association or other appropriate agencies any hazardous, illegal, or unethical professional decisions or practices by members, licensees, or others".

Accordingly, an Engineering Professional who observes a significant deficiency in any aspect of the project should report it to the client or to the client's representative, and if the client or their representative does not respond appropriately, the observing Engineering Professional must inform the appropriate regulatory authorities of the significant deficiency.

In addition to the reporting obligation discussed above, if the client does not choose to proceed with appropriate actions to mitigate the significant deficiency, then it is recommended that the Engineering Professional express their concerns in writing, and note that he or she cannot take responsibility for their aspects of the project. It is recommended that in such a communication, the Engineering Professional notes that all Engineering Professionals are obligated to design in accordance with good engineering practice, including the practices outlined in these guidelines.

3.6 RELEVANT EXTERNAL STANDARDS AND GUIDELINES

Table 2: List of Relevant External Standards and Guidelines in this section contains references to relevant standards that might apply to Software Engineers involved in the development of Safety-Critical Software.

Importantly, Software Engineers must identify the standards and guidelines that apply to the projects that they support. Applicability may be established by consulting one or more of the following sources:

- Any federal, provincial, local, or other legally applicable legislation, regulations, or rules
- Contracts or other applicable documents agreed to with clients
- Standard industry practice

Industry standards evolve over time, so the following collection of top resources is not meant to be comprehensive nor is it a checklist for compliance with these guidelines. Rather, it represents a selection of common standards that reflect current practices in specific industries.

See also Section 6.0 References and Related Documents.

Table 2: List of Relevant External Standards and Guidelines

TITLE	DESCRIPTION	REFERENCE[a]
BEST PRACTICES		
Center for Internet Security (CIS) Critical Security Controls for Effective Cyber Defense	• Contains practices aimed at improving security for critical computer systems. • Contains actionable guidance that Engineering Professionals and organizations may consult to improve security of connected systems.	CIS 2019a
CIS Benchmarks: Secure System Configurations	• Secure benchmark system configurations for common commercial off-the-shelf Software and operating systems. • These benchmarks can be followed to ensure that critical computer systems are deployed in a secure manner.	CIS 2019b
Open Web Application Security Project (OWASP) Secure Software Development Life Cycle	• A standard approach that can be applied to the Software development life cycle (SDLC) of both online and offline applications, to reduce the occurrence of security defects that reach production codebases. • The secure SDLC covers all stages of the Software development process.	OWASP Foundation 2019
FRAMEWORKS		
Building Security In Maturity Model (BSIMM)	• Based on real-world practices of companies that include security in their Software development practices. • Comprises 4 domains that include 12 practices and a total of 119 security-related activities that are designed to increase the maturity of secure Software development practices at an organization.	Synopsis 2020
National Institute for Standards and Technology (NIST) Cybersecurity Framework	• Consists of standards and guidelines that focus on improving the security profile of critical infrastructure. • Developed in the United States of America and may require adaptation to a Canadian context. • However, it is widely recognized internationally as a high-quality framework for managing cybersecurity Risk.	NIST 2019
REGULATIONS		
WorkSafeBC Occupational Health and Safety Regulation (OHSR), Sections 19.36 – 19.40 Control Systems	• Deal with the design of control systems and require that qualified persons design the control system. • Although the design of control systems is not solely within the scope of Software engineering, if the control system being designed is for a Safety-Critical system, and if there is Software in that system that controls Safety-Critical functions, work related to creating that control-system Software must be done by a Software Engineer.	WorkSafeBC 2019

TITLE	DESCRIPTION	REFERENCE[a]
STANDARDS		
DO-178C Software Considerations in Airborne Systems and Equipment Certification	• Provides recommendations regarding the Software engineering for Software used in aircraft. • Also has a number of appendices that cover specific topics such as Tool Qualification (DO-330), Model Based Software Engineering (DO-331), Object-Oriented Programming (DO-332), and Formal Methods (DO-333).	RTCA 2011a
DO-278A Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems	• This related companion to DO-178C is focused on Software engineering for ground-based systems used in the aviation industry (for example, air traffic control).	RTCA 2011b
IEC 61508 Functional Safety of Electrical/Electronic/Programmab le Electronic Safety-related Systems	• Recommends generic systems, hardware, and Software engineering activities and is not tailored for a specific domain. • Widely adopted in industry. Many of the activities recommended in these guidelines are consistent with recommendations in IEC 61508.	IEC 2010
ISO 26262 Road Vehicles – Functional Safety	• A refinement of IEC 61508 for the automotive industry, and provides recommendations regarding system, hardware, and Software engineering processes for automotive systems.	ISO 2018
ISO 62304 Medical Device Software – Software Life Cycle Processes	• Provides guidance on Software engineering activities for the development of medical devices.	ISO 2006
ISO/IEC 27001 Information Technology – Security Techniques – Information Security Management Systems — Requirements	• Provides high-level guidance for the management of information security. • Covers a broad range of information security topics; accordingly, only some clauses are specific to Safety-Critical Software.	ISO/IEC 2013
ISO/IEC/IEEE 29148 Systems and Software Engineering – Life Cycle Processes – Requirements Engineering	• Specifies the required processes implemented in the engineering activities that result in requirements for systems and Software products (including services) throughout the life cycle.	ISO/IEC/IEEE 2018
UL 1998 Standard for Software In Programmable Components	• Provides requirements related to non-networked embedded Software residing in programmable components performing Safety-related functions, whose failure is capable of resulting in a Risk of fire, electric shock, or injury to persons.	UL 2013

NOTE

[a] Full references are listed in Section 6.0 References and Related Documents.

4.0 QUALITY MANAGEMENT IN PROFESSIONAL PRACTICE

4.1 QUALITY MANAGEMENT REQUIREMENTS

Engineering Professionals must adhere to the applicable quality management requirements during all phases of the work, in accordance with the Association's Bylaws. It is also important to be aware of whether additional quality management requirements exist from authorities having jurisdiction or through service contracts.

To meet the intent of the quality management requirements, Engineering Professionals must establish and maintain documented quality management processes for the following activities:

- The application of relevant professional practice guidelines
- Authentication of professional documents by the application of the professional seal
- Direct supervision of delegated professional engineering activities
- Retention of complete project documentation
- Regular, documented checks using a written quality control process
- Documented field reviews of Engineering designs/recommendations during implementation or construction

4.1.1 PROFESSIONAL PRACTICE GUIDELINES

Engineering Professionals are required to comply with the intent of any applicable professional practice guidelines related to the engineering work they undertake. One of the three objectives of the Association, as stated in the *Act* is "to establish, maintain, and enforce standards for the qualifications and practice of its members and licensees". Practice guidelines are one means by which the Association fulfills this obligation.

These professional practice guidelines establish the standard of practice for the development of Safety-Critical Software. Software Engineers and other Engineering Professionals who carry out these activities are required to meet the intent of these guidelines.

4.1.2 USE OF SEAL

In accordance with the *Act*, s.20(9), Engineering Professionals are required to seal all professional engineering documents they prepare or deliver in their professional capacity to others who will rely on the information contained in the documents. This applies to documents that Engineering Professionals have personally prepared and those that others have prepared under their direct supervision.

Failure to seal these engineering or geoscience documents is a breach of the *Act*.

4.1.2.1 Sealing Software Engineering Documents

The Association's *Quality Management Guidelines – Use of Seal* outline sealing requirements for general engineering practices (Engineers and Geoscientists BC 2017). Therefore, this section aims to provide an interpretation of those guidelines that is specific to Software engineering and aligns with the Association's standard of practice for the use of seal in Safety-Critical Software engineering projects. Note that the recommendations in those generic guidelines still apply to a Software Engineer's general engineering practice; however, Software Engineers must be familiar with both the following discipline-specific interpretation and the generic guidelines, and must use their professional judgment when determining sealing procedures for a specific project or context.

Where a Software Engineer performs Safety-Critical Software engineering work in British Columbia, the Software Engineer is required to seal all relevant engineering documents and artifacts according to the *Quality Management Guidelines – Use of Seal* (Engineers and Geoscientists BC 2017).

If a client refuses to accept the sealing of certain project deliverables, the Software Engineer is not required to seal the deliverables submitted to the client but must keep a record of the client's refusal. In all cases, the Software Engineer must keep a sealed version of relevant engineering documents and artifacts for his or her own records.

4.1.2.2 Sealing Software Engineering Work Within an Organization

Software Engineers often work within organizations as employees, and have varying roles that combine technical expertise with management responsibilities.

In such cases, the Software Engineer should advocate within their organization for a professional engineering practice policy. This policy should outline the roles and responsibilities of a Software Engineer with respect to sealing Safety-Critical Software engineering documents.

It may be necessary for each project to have its own project-specific clarifications in project-specific plans. These documents should impose procedures for sealing deliverables that are consistent with the guidance in *Quality Management Guidelines – Use of Seal* (Engineers and Geoscientists BC 2017) and with the obligation at section 20(9) of the *Act* that professional engineers seal all professional engineering documents that they prepare or deliver in their professional capacity. If a professional engineering practice policy has not been finalized, Software Engineers must use their professional judgment to assess whether a particular deliverable must be sealed, and are responsible for storing sealed versions of relevant deliverables that they have created. (See also Section 4.1.4 Retention of Project Documentation).

4.1.2.3 Sealing Source Code Artifacts

Software Engineers might produce Source Code (or similar) artifacts as part of their engineering work. This is often referred to as a "build," meaning a compilation of Source Code into binary files suitable for integration into a target application environment or larger Software project. However, in some circumstances, Software Engineers may be delivering plaintext Source Code files for compilation, integration, and testing by other parties.

In cases where it is reasonable to apply a physical or digital seal directly to the Source Code (for example, when the number of lines of code is small), the Software Engineer should do so and store the resulting sealed document accordingly.

However, in most cases it is not reasonable to apply a physical or digital seal directly to the Source Code if the volume of code is large or widely distributed, or if there are further dependencies on third-party items such as shared libraries or compiler versions. In such cases, it is usual practice for the Software Engineer to produce and seal a declaration document that precisely indicates the items for which the Software Engineer is providing assurance.

A declaration document should include the following information, at a minimum:

- Clear identification of and reference to the archival location of the provided artifact.
- For compiled artifacts, a version reference to the systematic version control system for the Source Code that constitutes the artifact.
- Binary identification information that uniquely identifies binary and/or source files (for example, an MD5 checksum for each file).
- A qualification statement regarding the intended use of the provided artifact. Examples of such statements include the following:
  - For unit test
  - For integration with larger project
  - For deployment to production environment
  - For testing prior to deployment
- A declaration statement similar to the following:
  - "The seal and signature of the Engineering Professional on this document provides assurance that the essential phases of Software development outlined in the *Professional Practice Guidelines – Development of Safety-Critical Software* have been followed in the development of the Software."

Additionally, the declaration document may refer to or describe the following, as appropriate to the nature and intended use of the article being delivered:

- For compiled items, details related to the build environment such as the following:
  - Compiler identification, including compiler version
  - Build target environment
  - Third-party libraries required, and their version numbers
  - Any other required files, such as makefiles or binary reference tables
- References to applicable test results
- Lists of resolved and/or known issues
- References to applicable user, operation, or maintenance manuals

4.1.2.4 Sealing Evolving Software Engineering Artifacts

It is common practice in Software engineering to produce many revisions of Software engineering artifacts (such as Source Code). These artifacts might be released to other departments in an organization (such as the testing department) or even to the client in draft form.

Provided the revision is clearly marked as a "draft" (or a similar term), it is not mandatory to seal these artifacts. Once a "final" or "release" version of the artifact has been identified, the Software Engineer should seal the artifact according to the guidance above.

4.1.2.5 Employing Digital Sealing Technology

Software Engineers may seal electronic documents using an electronic version of their seal in conjunction with digital certificate technology, from a provider such as Notarius, Inc.

For more information, refer to *Quality Management Guidelines – Use of Seal* (Engineers and Geoscientists BC 2017).

4.1.3 DIRECT SUPERVISION

In accordance with the *Act*, s.1(1) and 20(9), Engineering Professionals are required to directly supervise any engineering work they delegate. When working under the direct supervision of an Engineering Professional, unlicensed persons or non-members may assist in performing engineering work, but they may not assume responsibility for it. Engineering Professionals who are limited licensees may only directly supervise work within the scope of their license.

With regard to direct supervision, the Engineering Professional having overall responsibility should consider:

- the complexity of the project and the nature of the Risks;

- which aspects of the work should be delegated;
- the training and experience of individuals to whom work is delegated; and
- the amount of instruction, supervision, and review required.

These guidelines recognize that Software engineering projects are complex and often involve large artifacts (such as Source Code repositories with millions of lines of Source Code) developed by large teams of individuals spread across many jurisdictions, particularly with respect to use of open-source items or shared libraries. As such, it is not always reasonable for one Software Engineer to take responsibility for all aspects of a project.

Accordingly, Software Engineers may create a declaration document that includes statements indicating the scope of their work (and those they directly supervise) for which they are taking responsibility. The Software Engineer should seal the declaration in accordance with the recommendations discussed above in Section 4.1.2 Use of Seal.

A version of the following statement may be used:

> "The seal and signature of the undersigned on this document provides assurance that established quality management processes, policies, and Software development activities have been followed by the undersigned and those directly supervised by the undersigned. The undersigned does not warrant or guarantee with respect to latent defects in third-party components of the described design or deliverable not discovered during the development process, but does, by sealing and signing, provide assurance that the Software product substantially complies in all material respects with the intent of the *Professional Practice Guidelines – Development of Safety-Critical Software*."

The following actions or practices may indicate that a supervising Engineering Professional has not adequately directly supervised his or her subordinates:

- Being regularly and for significant periods of time absent from the principal office premises from which professional services are rendered
- Being regularly and for significant periods of time out of communication with subordinates under the supervising Engineering Professional's supervision
- Failing to personally inspect or review the work of subordinates where necessary and appropriate
- Conducting a limited, cursory, or perfunctory review of plans or projects in lieu of appropriate detailed review
- Failing to be personally available on a reasonable basis or with adequate advance notice for consultation with subordinates where circumstances require personal availability

For more information, refer to *Quality Management Guidelines – Direct Supervision* (Engineers and Geoscientists BC 2018a).

4.1.4 RETENTION OF PROJECT DOCUMENTATION

In accordance with Bylaw 14(b)(1), Engineering Professionals are required to establish and maintain documented quality management processes that include retaining complete project documentation for a minimum of ten (10) years after the completion of a project or ten (10) years after engineering documentation is no longer in use.

These obligations apply to Engineering Professionals in all sectors. Project documentation in this context includes documentation related to any ongoing engineering work, which may not have a discrete start and end, and may occur in any sector.

Many Engineering Professionals are employed by organizations, which ultimately own the project documentation. Engineering Professionals are considered compliant with this quality management requirement when a complete set of project documentation is retained by the organizations that

employ them using means and methods that are consistent with the Association's Bylaws and guidelines.

For more information, refer to *Quality Management Guidelines – Retention of Project Documentation* (Engineers and Geoscientists BC 2018b).

4.1.5 DOCUMENTED CHECKS OF ENGINEERING AND GEOSCIENCE WORK

In accordance with Bylaw 14(b)(2), Engineering Professionals are required to perform a documented quality checking process of engineering work, appropriate to the Risk associated with that work.

Regardless of sector, Engineering Professionals must meet this quality management requirement. In this context, 'checking' means all professional deliverables must undergo a documented quality checking process before being finalized and delivered. This process would normally involve an internal check by another Engineering Professional within the same organization. Where an appropriate internal checker is not available, an external checker (i.e., one outside the organization) must be engaged. Where an internal or external check has been carried out, this must be documented.

Engineering Professionals are responsible for ensuring that the checks being performed are appropriate to the level of Risk. Considerations for the level of checking should include the type of document and the complexity of the subject matter and underlying conditions; quality and reliability of background information, field data, and elements at Risk; and the Engineering Professional's training and experience.

The same principles apply to the checking of Source Code, input files, and other artifacts of the Software development process.

For more information, refer to *Quality Management Guidelines – Documented Checks of Engineering and Geoscience Work* (Engineers and Geoscientists BC 2018c).

4.1.6 DOCUMENTED FIELD REVIEWS DURING IMPLEMENTATION OR CONSTRUCTION

In accordance with Bylaw 14(b)(3), field reviews are reviews conducted at the site of the construction or implementation of the engineering work. They are carried out by an Engineering Professional or a subordinate acting under the Engineering Professional's direct supervision (see **Section 4.1.3 Direct Supervision**).

Field reviews enable the Engineering Professional to ascertain whether the construction or implementation of the work substantially complies in all material respects with the engineering concepts or intent reflected in the engineering documents prepared for the work.

In the context of these guidelines, the requirement for field review can be interpreted to apply to both verification of Software and its ongoing maintenance. Refer to the following sections for descriptions of such activities:

- Section 3.2.1.5 Verification of Software
- Section 3.2.1.6 Maintenance of Software
- Section 3.4.3 Security Verification and Validation

For more information, refer to *Quality Management Guidelines – Documented Field Reviews during Implementation or Construction* (Engineers and Geoscientists BC 2018d).

5.0 PROFESSIONAL REGISTRATION & EDUCATION, TRAINING, AND EXPERIENCE

5.1 PROFESSIONAL REGISTRATION

It is the responsibility of Engineering Professionals to determine whether they are qualified by training and/or experience to undertake and accept responsibility for carrying out Safety-Critical Software development (Code of Ethics Principle 2).

As described in these guidelines, the creation of Safety-critical Software requires the systematic application of engineering principles, and its operation has the potential to cause personal harm, injury, illness, death, or damage to the environment. Safety-critical Software development constitutes the practice of professional engineering under the *Act*.

In the broader Software development community, the specification, design, implementation, verification, deployment, or maintenance of Software is not limited to Engineering Professionals. Many types of Software exist that do not require a systematic, disciplined, quantifiable approach to their development, or have minimal Risk associated with their application, and as a result do not require the oversight of an Engineering Professional.

Such work may be conducted by individuals other than Software Engineers. These guidelines are not intended to apply directly to projects that are not Safety-Critical Software projects.

5.2 EDUCATION, TRAINING, AND EXPERIENCE

The creation of Safety-Critical Software requires certain levels of education, training, and experience in many overlapping areas of engineering. The Engineering Professional who takes responsibility for Safety-Critical Software must adhere to the Association's Code of Ethics (to undertake and accept responsibility for professional assignments only when qualified by training or experience) and, therefore, must evaluate his or her qualifications and must possess the appropriate education, training, and experience to provide the services. The level of education, training, and experience required of the Engineering Professional should be adequate for the complexity of the project.

Engineering Professionals, as registered professionals, have met minimum education, experience, and character requirements for admission to the profession. However, the educational and experience requirements for professional registration do not necessarily constitute an appropriate combination of education and experience for the creation of Safety-Critical Software.

This section describes a set of indicators that Engineering Professionals can use to determine whether they have an appropriate combination of education and experience. Note that these indicators are not an exhaustive list of education and experience types that are relevant to Safety-Critical Software

engineering. Satisfying one or more of these indicators does not automatically imply competence in Safety-Critical Software engineering.

5.2.1 EDUCATIONAL INDICATORS

Certain indicators show that an Engineering Professional has received education that might qualify him or her to participate professionally in the creation of Safety-Critical Software. Educational indicators are subdivided into formal education (such as university or engineering school) and informal education (such as continuing professional development).

Formal educational indicators include that the Engineering Professional has obtained or completed one or more of the following:

- An undergraduate-level degree in Software engineering or a related engineering field from an accredited engineering program
- An undergraduate-level or graduate-level degree in computer science or mathematics from a college, university, or engineering school where the degree contains a combination of theoretical and practical educational experiences
- A non-degree training program offered by a university, college, or other educational institution that focuses on Software engineering topics

Informal educational indicators include that the Engineering Profession has participated in or undertaken one or more of the following:

- Training courses facilitated by the Engineering Professional's employer that focus on Software engineering topics
- Continuing professional development courses or sessions offered by professional organizations (such as Engineers and Geoscientists BC) that focus on Software engineering topics
- Conferences or industry events which focus on Software engineering topics
- A rigorous self-study program involving a structured approach that contains materials from text books and technical papers on Software engineering topics

5.2.2 EXPERIENCE INDICATORS

Certain indicators show that an Engineering Professional has an appropriate combination of experience that might qualify him or her to participate professionally in the creation of Safety-Critical Software.

Experience indicators include that the Engineering Professional has completed one or more of the following:

- For an extended duration (greater than one year) and/or as an Engineering-in-Training (EIT), participated in the creation of Safety-Critical Software under the direct supervision of a professional engineer with an appropriate combination of education and experience
- By participating in past projects working alongside Software Engineers, developed a sufficient knowledge of Software engineering principles
- Participated in academic or industry working groups that focus on Software engineering topics
- Obtained substantial experience creating production-grade Software that, although it is not Safety-Critical Software, shares properties of Safety-Critical Software in its application and development process (such as Software deployed in high-reliability telecommunications networks)

5.2.3 EXAMPLES OF EDUCATION AND EXPERIENCE

Following are fictional descriptions of individuals who, through a combination education and experience may or may not be qualified to contribute professionally to the creation of Safety-Critical Software.

5.2.3.1 Formally Trained and Mentored

Susan is a Software Engineer who obtained a Bachelor of Engineering with a specialization in Software engineering five years ago. Since graduating, Susan has worked as a Software developer on a mix of Safety-Critical and non–Safety-Critical projects at an automotive company under the supervision of several experienced Software Engineers. This time also

counted towards her Engineer-in-Training (EIT) experience. Within the last year, Susan has registered as an Engineering Professional and continues to undertake continuing professional development focused on Software engineering topics.

Based on her education and industrial experience, Susan is likely qualified to undertake Safety-Critical Software engineering work.

5.2.3.2 Experience in Adjacent Discipline

Kaleb is a licensed engineer who practices electrical engineering and specializes in the design of electronics. Due to the nature of his work, Kaleb has worked closely with Software Engineers and has begun to participate in Software engineering tasks, especially at the interface between the Software and the hardware he is working on. Additionally, Kaleb has attended several continuing professional development seminars on techniques for Software development and management.

Based on a combination of formal and informal education and extensive work experience, Kaleb is likely qualified to undertake Safety-Critical Software engineering work. However, Kaleb should be aware of his limitations, particularly when the scope of his work extends too far beyond the hardware-Software interfaces that he is experienced with.

5.2.3.3 Non–Safety-Critical Software Development Experience

Heather obtained a Bachelor of Computer Science six years ago and has since taken several jobs at a large Software company that makes a number of Internet Software applications. While Heather is a very skilled Software developer, none of Heather's experience has been in development of Safety-Critical Software or Software of a similar level of criticality.

At present, Heather is not qualified to (independently) undertake Safety-Critical Software engineering work.

Heather could work towards becoming qualified by: 1) successfully registering as an Engineering Professional; and 2) working under the direct supervision of a Software Engineer on one or more Safety-Critical Software projects for an extended period of time.

5.2.3.4 Experience in an Unrelated Discipline

Wilson has been an Engineering Professional for the last 20 years and specializes in the configuration and installation of heating, ventilation, and air conditioning (HVAC) systems. Wilson has been assigned as a subject matter expert to a new project that incorporates Safety-Critical Software. He has written Source Code for a number of control algorithms. The last time Wilson wrote Source Code was during a college programming course.

Wilson is not likely qualified to contribute in this manner to his company's Safety-Critical Software without direct supervision from a Software Engineer. Possible choices for Wilson include the following:

- Wilson could limit his involvement in the project to his original subject matter expert role, where he designs the control system(s) and provides the designs to a qualified Software Engineer for further Software design and implementation. In this case, Wilson should participate in the Software engineering process (for example, through review, meetings) to ensure that the intent of his design(s) are fully implemented and verified appropriately.
- Wilson (or his company) could engage a qualified Software Engineer to directly supervise the Software engineering work related to Wilson's control system design. In this case, Wilson may continue to write the Source Code, but should defer to the Software Engineer regarding matters relating to Software engineering.
- Wilson could undertake a combination of formal and informal Software engineering education and also work under the direct supervision of a Software Engineer on a Safety-Critical Software project for an extended period of time (perhaps in the same company).

6.0 REFERENCES AND RELATED DOCUMENTS

Documents cited in these guidelines appear in Section 6.1: Regulations, Section 6.2: References, and Section 6.3: Codes and Standards.

Related documents that may be of interest to users of these guidelines but are not formally cited elsewhere in this document appear in Section 6.4: Related Documents.

6.1 REGULATIONS

Engineers and Geoscientists Act [RSBC 1996], Chapter 116.

Workers Compensation Act [RSBC 1996], Chapter 492.

Workers Compensation Act, Occupational Health and Safety Regulation, B.C. Reg. 296/97.

6.2 REFERENCES

Center for Internet Security (CIS). 2019a. CIS Controls. [accessed: 2019 Jun 14]. https://www.cisecurity.org/controls/.

CIS. 2019b. CIS Benchmarks. [accessed: 2019 Nov 26]. https://www.cisecurity.org/cis-benchmarks/.

Engineers and Geoscientists BC. 2018a. Quality Management Guidelines: Direct Supervision. Version 1.3. Burnaby, BC: Engineers and Geoscientists BC. [accessed: 2019 Jun 14]. https://www.egbc.ca/Practice-Resources/Quality-Management-Guidelines.

Engineers and Geoscientists BC. 2018b. Quality Management Guidelines: Retention of Project Documentation. Version 1.3. Burnaby, BC: Engineers and Geoscientists BC. [accessed: 2019 Jun 14]. https://www.egbc.ca/Practice-Resources/Quality-Management-Guidelines.

Engineers and Geoscientists BC. 2018c. Quality Management Guidelines: Documented Checks of Engineering and Geoscience Work. Version 1.3. Burnaby, BC: Engineers and Geoscientists BC. [accessed: 2019 Jun 14]. https://www.egbc.ca/Practice-Resources/Quality-Management-Guidelines.

Engineers and Geoscientists BC. 2018d. Quality Management Guidelines: Documented Field Reviews During Implementation or Construction. Version 1.3. Burnaby, BC: Engineers and Geoscientists BC. [accessed: 2019 Jun 14]. https://www.egbc.ca/Practice-Resources/Quality-Management-Guidelines.

Engineers and Geoscientists BC. 2017. Quality Management Guidelines: Use of Seal. Version 2.0. Burnaby, BC: Engineers and Geoscientists BC. [accessed: 2019 Jun 14]. https://www.egbc.ca/Practice-Resources/Quality-Management-Guidelines.

Engineers Canada. 2016. White Paper on Professional Practice in Software Engineering. Ottawa, ON: Engineers Canada. [accessed: 2019 Jun 14]. https://engineerscanada.ca/publications/white-paper-on-professional-practice-in-software-engineering.

Ericson II CA. 2015. Hazard Analysis Techniques for System Safety. 2nd ed. Hoboken, NJ: Wiley.

Leveson NG, Thomas JP. 2018. STPA [System Theoretic Process Analysis] Handbook. Cambridge, MA: Partnership for Systems Approaches to Safety and Security. [accessed: 2019 Jun 14]. http://psas.scripts.mit.edu/home/get_file.php?name=STPA_handbook.pdf.

National Institute of Standards and Technology (NIST). 2019. Cybersecurity Framework. [accessed: 2019 Jun 14]. https://www.nist.gov/cyberframework.

OWASP Foundation. 2019. Open Web Application Security Project (OWASP). [website]. [accessed: 2019 Nov 26]. https://owasp.org/.

Shostack A. 2014. Threat Modeling: Designing for Security. Indianapolis, IN: John Wiley & Sons, Inc.

Synopsis Software Integrity Group (Synopsis). 2020. Building Security In Maturity Model (BSIMM) 10. [accessed: 2019 Nov 26]. https://www.bsimm.com/.

Young W, Leveson NG. 2014. An Integrated Approach to Safety and Security Based on Systems Theory. Communications of the Association for Computing Machinery, vol. 57, no. 2, pp. 31–35.

6.3 CODES AND STANDARDS

International Electrotechnical Commission (IEC). 2010. IEC 61508:2010 Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems. Geneva, Switzerland: IEC.

International Organization for Standardization (ISO). 2018. ISO 26262:2018 Road Vehicles – Functional Safety. Geneva, Switzerland: ISO.

ISO. 2019. ISO 15026:2019 Systems and Software Engineering -- Systems and Software Assurance. Geneva, Switzerland: ISO.

ISO. 2006. ISO 62304:2006 Medical Device Software – Software Life Cycle Process. Geneva, Switzerland: ISO.

ISO/IEC. 2013. ISO/IEC 27001:2013 Information Technology -- Security Techniques -- Information Security Management Systems – Requirements. Geneva, Switzerland: ISO

ISO/IEC/Institute of Electrical and Electronics Engineers (IEEE). 2018. ISO/IEC/IEEE 29148:2018 Systems and Software Engineering — Life Cycle Processes — Requirements Engineering. Geneva, Switzerland: ISO.

Radio Technical Commission for Aeronautics (RTCA). 2011a. DO-178C Software Considerations in Airborne Systems and Equipment Certification. Washington, DC: RTCA.

RTCA. 2011b. DO-278A Software Integrity Assurance Considerations for Communication, Navigation, Surveillance and Air Traffic Management (CNS/ATM) Systems. Washington, DC: RTCA.

Safety Critical Systems Club (SCSC) Assurance Case Working Group (ACWG). 2018. SCSC-141B. Goal Structuring Notation Community Standard. Version 2. [accessed: 2019 Nov 26].: https://scsc.uk/scsc-141B.

WorkSafeBC. 2019. Occupational Health and Safety Regulation. B.C. Reg. 296/97. [accessed: 2019 Jun 14]. Victoria, BC: WorkSafeBC. https://www.worksafebc.com/en/law-policy/occupational-health-safety/searchable-ohs-regulation/ohs-regulation/ or https://www.canlii.org/en/bc/laws/regu/bc-reg-296-97/latest/bc-reg-296-97.html.

Underwriters Laboratories (UL). 2013. UL 1998 Standard for Software in Programmable Components. Edition 3. Northbrook, IL: UL.

6.4 RELATED DOCUMENTS

Feiler P, Goodenough J, Gurfinkel A, Weinstock CB, Wrage L. 2013. Four Pillars for Improving the Quality of Safety-Critical Software-Reliant Systems. White paper DM 288. Pittsburgh: Software Engineering Institute.

Holzmann GJ. 2006. The Power of Ten – Rules for Developing Safety Critical Code. IEEE Computer 39 (6), June 2006, pp. 93-95 DOI: 10.1109/MC.2006.212.

IEC. 2006. IEC 62304:2006 Medical Device Software – Software Life Cycle Process. Geneva, Switzerland: IEC.

IEEE. 1994. 1228-1994 - IEEE Standard for Software Safety Plans. Piscataway NJ: IEEE.

ISO. 2016. ISO 13485:2016 Medical Devices – Quality Management Systems – Requirements for Regulatory Purposes. Geneva, Switzerland: ISO.

ISO/IEC. 2014. ISO/IEC Guide 51:2014, Safety Aspects – Guidelines for Their Inclusion in Standards. Geneva, Switzerland: ISO.

ISO/IEC. 2011. ISO/IEC 25010:2011 Systems and Software Engineering -- Systems And Software Quality Requirements and Evaluation. Geneva, Switzerland: ISO.

ISO/IEC. 2012. ISO/IEC 19505-2: 2012 Information Technology – Object Management Group Unified Modeling Language (OMG UML) – Part 2: Superstructure. Geneva, Switzerland: ISO.

ISO/IEC. 2009. ISO/IEC Guide 73:2009 Risk Management – Vocabulary. Geneva, Switzerland: ISO.

ISO/IEC. 2009. ISO/IEC 15408:2009 Information Technology – Security Techniques – Evaluation Criteria for IT Security. Geneva, Switzerland: ISO.

ISO/IEC. 2004. ISO/IEC 9126:2004 Software Engineering – Product Quality. Geneva, Switzerland: ISO (superseded by ISO 25010).

Leveson NG. 1995. Safeware: System Safety and Computers. Reading, MA: Addison-Wesley.

Leveson NG. 2011. Engineering a Safer World: System Thinking Applied to Safety. Cambridge, MA: MIT Press.

7.0 APPENDIX

Appendix A: Authors and Reviewers 43

APPENDIX A: AUTHORS AND REVIEWERS

All contributors are presented in alphabetical order by last name within their respective sections.

PRIMARY AUTHORS

Simon Diemert, P.Eng., Critical Systems Labs

Dan Rankin, P.Eng., Engineers and Geoscientists BC

Jens Weber, PhD, P.Eng., University of Victoria

The primary authors would like to acknowledge Philippe Krutchen, PhD, P.Eng., who established the terms of reference and set the path for development of these guidelines.

The authors would also like to acknowledge Harshan Radhakrishnan, P.Eng., for his assistance with development of these guidelines.

REVIEW GROUP

Peter Angus, P.Eng., Schneider Electric

Pieter Botman, P.Eng., FEC, True North Systems Consulting

Sandy Buchanan, EIT, Mirai Security

Michael Henrey, EIT, Kardium Inc.

Fieran Mason-Blakley, EIT, Leverage Analytics

Martin Petruk, P.Eng., Vancouver International Airport

Kirk Richardson, EIT, Kobelt Manufacturing

Mark Sudul, P.Eng., Telus

Bastian Tenbergen, PhD, State University of New York at Oswego

Eric Zhelka, P.Eng., Emergency Management British Columbia

NOTES:

200 • 4010 REGENT STREET, BURNABY, BRITISH COLUMBIA, CANADA V5C 6N2
MAIN 604.430.8035 TOLL FREE 1.888.430.8035 EMAIL info@egbc.ca WEBSITE egbc.ca