ISSN 2754-1118
Vol.1 No.1



The Safety-Critical Systems Club

SAFETY-CRITICAL SYSTEMS eJOURNAL

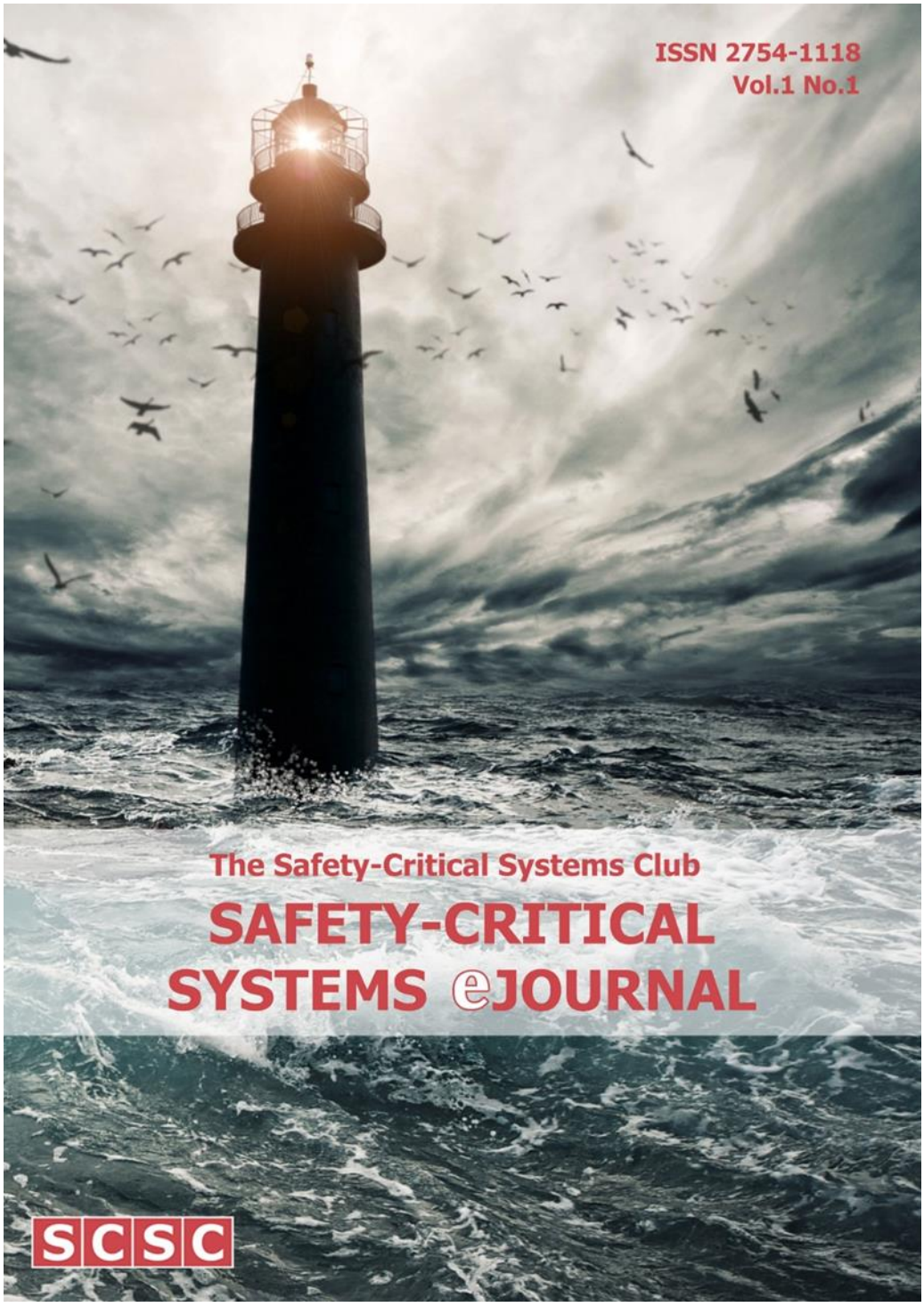
SCSC

scsc.uk/SCSC-174

About the Cover

The lighthouse, depicted here by Alex King as an island of stability in a stormy sea, represents a safety-critical system, which is intended to provide stable, trusted outcomes in a chaotic environment.

Lighthouses provide help to seafarers and have done so for thousands of years. Some warn of the presence of rocks or of other long-term hazards to navigation. Others are navigational aids and, with other cues, may be used to find a safe passage to harbour. Many perform both rôles; the lighthouse is thus itself (part of) a safety-critical system.



© Text: John Spriggs, Cover Art: Alex King 2022.

Welcome to the First Issue of a New Journal!

Welcome to the launch issue of the Safety-Critical Systems eJournal, which is published by the Safety Critical Systems Club, a UK Community Interest Company. Our mission is, "To publish high-quality, peer-reviewed articles on the subject of systems safety". By systems, we mean not only the platforms, but also the people and their procedures that make up the whole. Systems safety addresses those systems, their components, and the services they are used to provide. We do not have a narrow view of system safety, however; our scope is wide, and includes safety-related topics such as resilience, security, public health, and environmental impact.

This is not *purely* an academic journal presenting research results, we also intend to report upon practical aspects of systems safety; what works and what does not. The aim is to reflect how industrial practise is developing, for example how new standards are to be interpreted, or what new analysis techniques are being trialled. We are also interested in the past, how tools and techniques were justified for deployment, or significant lessons learned (and acted upon); and in the present, with topical or industry review articles, for example.

There will be two issues per volume, published in January and July of each year. In addition to the on-line presentation, each Volume of the journal will be made available in printed form both for libraries and for those of us who want a more-permanent record. The print volume will be published each December.

If you would like to submit a paper for a future issue, please see "Information For Authors" in the right-hand pane of the journal home page.

In this issue we have three papers:

- Rob Ashmore & James Sharp (UK) propose a set of generic assurance topics applicable to all types of programmable content for all types of platform, giving some novel examples;
- Dewi Daniels & Nick Tudor (UK) claim that many software reliability models do not provide results in which sufficient confidence can be placed, proposing an alternative approach; and
- Bruce Hunter (Australia) discusses how cyber-attacks, such as a ransomware attack on a business system, may also affect critical infrastructure, whether intentionally, or unintentionally; he considers cybersecurity threat mitigation, minimising the impact of attacks on safety systems.

My thanks go to the authors for contributing these papers, and also to the peer-reviewers (four per paper) for suggesting improvements. Apologies also to those reviewers who made some recommendations that were not taken up.

You may find some of this material controversial, or you may think that it does not go far enough. Subsequent issues of this journal will have provision for readers' letters to the Editor responding to individual papers.

For more resources addressing system safety, see the Club's web-site https://scsc.uk and please support us by becoming a member.

John Spriggs, Editor

January 2022

© John Spriggs 2022.

Generic Assurance Topics for Any Type of Programmable Content

Rob Ashmore and James Sharp

Dstl, Portsdown West, UK

Abstract

An ever-growing range of technologies can be used to implement programmable content. Examples include highly complex System-On-Chip designs, which use traditional electronics, as well as approaches based on quantum technologies and biological systems. Standards and guidance are required to support the safe use of these, and other, emerging programmable technologies. However, technology-specific guidance is challenging to produce, especially in a timely manner. To help bridge this gap, we propose a set of generic assurance topics, which are applicable to all types of programmable content, introducing considerations based on the assurance of both a program, and the associated substrate. The topics are initially introduced by considering multi-core processors. Their application to alternate technologies is illustrated by considering electronic hardware tailored for machine learning, quantum computing and computation using a bio-based substrate.

1 Introduction

1.1 Motivation

Safety-critical systems are making ever-increasing use of programmable content. This progress has been enabled by safety standards and guidance material. For example, RTCA/DO-178C (RTCA 2011a) addresses software, RTCA/DO-254 (RTCA 2000) addresses Complex Electronic Hardware (CEH) and both RTCA/DO-200 (RTCA 2015) and SCSC-127F (SCSC 2021) address data. Collectively, these examples cover three main aspects of programmability, specifically, software, hardware, and data. Whilst these examples are widely applicable, there is a growing recognition that technology-specific details are important. This is apparent, for example, in RTCA/DO-178C's supplements, e.g. RTCA/DO-331 (RTCA 2011b), which covers model-based development, and documents related to Multi-Core Processors (MCPs), e.g. CAST-32A (CAST 2016).

Programmable content technologies are developing rapidly. The associated need for specific, detailed information inevitably challenges the safety community. We could wait until the technology is well understood and key aspects have been codified as good practice, but this would mean standards significantly lag behind technology development. Another way of addressing the challenge would be to publish good practice rapidly, frequently updating it as new information becomes available. Neither of these approaches

© Crown copyright (2022), Dstl. This material is licensed under the terms of the Open Government Licence except where otherwise stated. To view this licence, visit http://www.nationalarchives.gov.uk/doc/open-government-licence/version/3 or write to the Information Policy Team, The National Archives, Kew, London TW9 4DU, or email: psi@nationalarchives.gov.uk.

is desirable; standards should not lag technology, nor should they change too often (Johnson 2016).

Consequently, we frame a generic set of topics that an applicant would be expected to discuss with a regulator. The details of those discussions would be specific to the technologies involved in a particular application. Such discussions may be similar to those associated with Certification Review Items (CRIs), which are used in the aviation domain. Equivalently, the topics we identify may be an appropriate structure for CRIs related to programmable content.

Over time, as experience is gained, information may be collected and summarised into a more-traditional form of codified good practice. Although feasible, we recognise this approach has limitations. In particular, both the applicant and regulator need a sufficiently detailed understanding of the relevant technologies: this is not easy to achieve.

1.2 Related Work

The use of a generic set of topics is not a new concept. For example, there are already the "4+1 Software Safety Assurance Principles" (Hawkins et al. 2013). In addition, the Federal Aviation Administration (FAA) and the National Aeronautics and Space Administration (NASA) have described a set of overarching properties (Holloway 2019).

Our work differs from the software safety assurance principles in that it explicitly considers the substrate on which the software is running. In addition, it differs from the overarching properties in that it only considers programmable content, rather than a whole aircraft or an entire system. Consequently, our scope is broader than that of the software safety assurance principles and narrower than that of the overarching properties. Crucially, this choice of scope allows us to direct an appropriate amount of attention to novel types of programmable content.

2 Structure of Our Approach

2.1 Concepts

A program instantiates user-observable behaviour, where a "user" might be, for example, a human or another system component. Although the implementation may be complicated, we suggest that a program's intended behaviour is sufficiently constrained to allow it to be captured in a set of requirements that are meaningful to a user. This is often because a program has been developed to satisfy the needs of a particular user (or group of users). For clarity, we note that our concept of a program encapsulates both software and data.

A substrate provides the physical environment in which the program executes. The same substrate is expected to be able to support many different programs (not necessarily executing simultaneously). Equivalently, from the view of any individual program, the substrate is over-specified; a single program will only use a portion of the substrate's capabilities. In addition, both the physical nature of the substrate and the typically complex interaction between substrate and program result in substrate-level requirements being more detailed and more numerous than user-level program requirements. Consequently, we suggest that, typically, it is not possible for all aspects of a substrate's behaviour to be captured by requirements.

Without a substrate, a program is merely theoretical. Although some claims might be made about a program's behaviour, e.g. number of steps associated with a worst-case execution, these inevitably make assumptions about the substrate. Consequently, safety assurance of a program cannot be completed in isolation. Conversely, without a program, a substrate delivers no user-observable functionality, so safety assurance of a substrate in isolation is, at best, incomplete. Hence, safety assurance arguments related to programmable content are about a program (or a collection of programs) executing on a given substrate. This observation reflects current practice, e.g. RTCA/DO-178C's consideration of the target computer (RTCA 2011a).

To simplify presentation, we often refer to a single program, with a single developer. In almost all cases, there will be multiple programs and multiple developers. For example, in a traditional implementation, the substrate would be some form of processor, whereas the Operating System (OS) and applications would be programs.

The preceding discussion may suggest a clear and obvious separation between program and substrate. As the example of processor microcode illustrates, however, this is not always the case. Fundamentally, for us, the program contains things that, by design, would be expected to vary between programs and hence are under developer control. By that interpretation, microcode would be part of the substrate. Regardless of where the dividing line is drawn for any specific application of a particular technology, it is important that nothing is left unconsidered. This is another reason why, as noted earlier, assurance is about a program executing on a substrate.

2.2 Program Level Assurance

We are concerned with providing assurance that programmable content behaves as expected by the user, which may be a human or another system component. Our focus is on showing that the programmable content meets the expectations of the user, rather than demonstrating that the user's expectations will result in a safe system. Nevertheless, assurance of programmable content behaviour should be a useful claim, within a wider assurance argument, that a system is acceptably safe. As noted previously, a program's intended behaviour is captured in a set of requirements. Consequently, assurance needs to protect against all the following situations (Figure 1):

1. Where there is a difference in understanding of the requirements between the user and the program developer: colloquially, "**requirements are misunderstood**".
2. Where the program's behaviour is a strict subset of the requirements: "**some expected behaviour is not present**".
3. Where the program's behaviour is a strict superset of the requirements: "**some unexpected behaviour is present**".

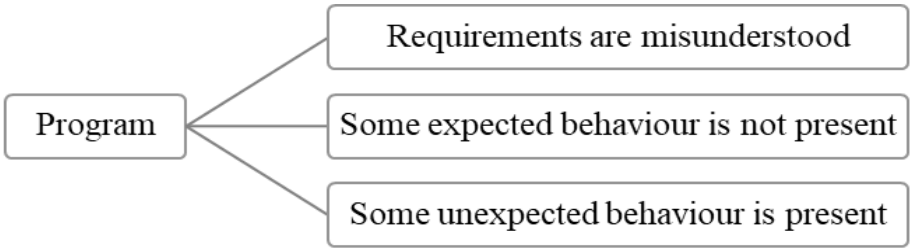


Figure 1 ~ Program Level Situations to Protect Against

It is helpful to reflect these behaviour-related outcomes in the more familiar context of traditional software.

In this case, protection against the first situation (**requirements are misunderstood**) is achieved by ensuring that requirements have particular characteristics, for example, accurate, complete, consistent, unambiguous and verifiable. Adopting a Requirements-Based Testing (RBT) approach, where the program and the tests are independently derived from the requirements, also helps protect against ambiguity in requirements.

RBT also protects against the second situation (**some expected behaviour is not present**). If the program's behaviour is verified against each requirement, then all expected behaviour should be present. This, of course, assumes that the verification methods are sound: control of the test environment and accuracy of tests cases are important considerations here.

Protection against the third situation (**some unexpected behaviour is present**) typically involves measuring code coverage using structural metrics, e.g. branch coverage, as part of an RBT endeavour. This approach is based on an implicit assumption that code structure (rather than, say, data) is the main way that different behaviours are instantiated. This approach also assumes that large-scale behaviour, as expressed in user-level requirements, can be decomposed to low-level behaviour, where coverage is typically measured. Robustness testing, e.g. checking for arithmetic overflow and exceeded frame times, also helps protect against some unexpected behaviour being present.

Formal methods (RTCA 2011c) provide an alternative approach to protecting against our three undesirable outcomes. In this approach, requirements are written in a formal language, guaranteeing an accurate, unambiguous, consistent (but not necessarily complete) description of intended behaviour. This description is refined, often through several stages, to formally demonstrate that the program instantiates the requirements. There have been several successful applications of formal methods; recent examples include a compiler (Leroy et al. 2016) and a microkernel (Klein et al. 2009).

Our three behaviour-related outcomes represent a slightly different way of considering program-level assurance. Although they cover similar themes, e.g. satisfaction of requirements, our outcomes provide an alternative perspective to the "4+1 Software Safety Assurance Principles" (Hawkins et al. 2013). Paradoxically, the widespread success of these principles, which summarise most current approaches to program-level safety, means they are not well-suited for our purpose. We believe they are, in many people's minds, wedded tightly to the development of traditional software, including the explicit, traceable, hierarchical decomposition of requirements. Consequently, they are not ideally suited for considering new types of programmable content (Ashmore and Lennon 2017). That said, our separation of program level and substrate level assurance means that the 4+1 Principles could be used for the former, with our topic areas being adopted for the latter. This might be an attractive approach when the program is largely traditional, with the novelty being in the substrate level.

In addition to the three behavioural outcomes noted above, there is also a need to consider protection against malicious intent. Different technologies may provide protection against certain types of malicious activity: the memory protection offered by the Rust programming language (Balasubramanian et al. 2017) is one example; the CHERI (Capability Hardware Enhanced RISC [Reduced Instruction Set Computer] Instructions) are another (Watson 2019). Most program developments are, however, subject to the same two vulnerabilities. Firstly, the insider threat, which may be partially protected against by review activities. Secondly, vulnerabilities inserted via the tool chain, e.g. (Thompson 2007), (Goodin 2017), and (Peisert et al. 2021), which may be partially (but only partially)

protected against by obtaining professional-quality tools from reputable supply chains. Given the common nature of these vulnerabilities, for reasons of brevity, malicious activity at the program level is not discussed in detail in this paper. Nevertheless, it remains a very important issue.

2.3 Substrate Level Assurance

We adopt an expectation-based approach for substrate-level assurance. In particular, we are concerned with demonstrating that the substrate behaves as expected by the program developer.

As discussed previously, we contend that substrate-level behaviour cannot be meaningfully encapsulated (for a program developer) in a set of requirements. Inevitably, due to the *breadth* and *depth* that these requirements would have to cover, expectations only ever capture a subset.

The *breadth* issue arises because the substrate supports many different types of program. Consequently, an individual program will only use a fraction of the substrate's capability. A program developer will focus on those aspects of the substrate that appear relevant to their specific development. This understandable focusing inevitably produces a partial picture.

The *depth* issue arises because the substrate spans from a physical implementation to the level of abstraction used by the program developer: for traditional processors, the substrate spans from transistor-level properties of electrons to the Instruction Set Architecture (ISA). Across this scope, many aspects of physics can affect behaviour. For traditional electronics, single event upsets are an example that may arise naturally (Taber and Normand 1993), as are manufacturing and age-related issues (Dixit et al. 2021). There are also malicious attacks that exploit predictable outcomes of changes in the substrate's electronic environment, e.g. (Mutlu and Kim 2019) and (Murdock et al. 2020). Encapsulating a complete set of these effects in requirements seems, to us, an insurmountable challenge.

Typically, some form of on-target testing addresses common behaviour. Hence, the question of unexpected behaviour reduces to identification of potential edge cases. Ideally, these should be identified by a process that is rigorous, repeatable, and auditable. This is often achieved through the provision of some form of structure: the guidewords used in a hazard and operability study (HAZOP) are one example (Crawley and Tyler 2015). We propose the structure illustrated in Figure 2.

At the top level, this structure distinguishes between two classes. The first class covers cases where the program developer expects the substrate to behave in a manner that could be feasible but is not exhibited at the relevant time: colloquially, we term this a "**could be, but isn't**" (CBBI) behaviour. The second class covers cases where the developer expects a behaviour that "**could never be**" (CNB) delivered by the substrate. This distinction can be important because the former class (CBBI) can typically be controlled through configuration, whereas the latter class (CNB) cannot.

Note that the notion of expectation used here includes cases where the program developer expects something to happen, as well as cases where the program developer expects something not to happen. As such, it encapsulates situations where expected behaviour is absent, as well as situations where unexpected behaviour is present.

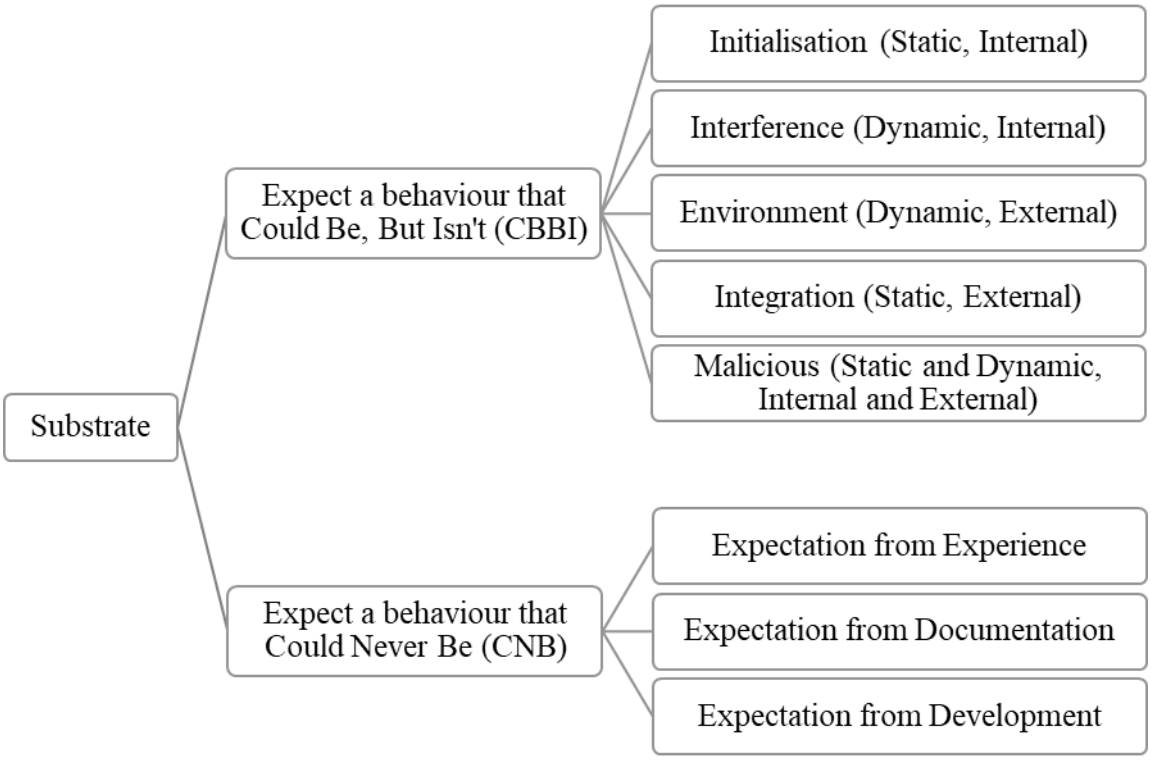


Figure 2 ~ Substrate Level Situations to Protect Against

Unexpected **CBBI** behaviours occur due to some influence on the substrate's behaviour. That influence may be **static,** i.e. remain fixed for the duration of program execution, or it may be **dynamic**. In addition, that influence may be **internal** to the substrate, or it may be **external**. The combination of these properties gives us four ways of identifying unexpected behaviour. These are illustrated in Table 1, with examples from the context of an MCP. These examples are provided solely to help illustrate our approach. As noted previously (in subsection 1.1), there are technology-specific standards for MCP, which are better suited to that specific technology than our more generic approach. Note that, as shown in Table 1, the "static, external" combination is addressed by (system) integration.

Table 1 ~ Examples of CBBI Protections in an MCP Context

	Static	**Dynamic**
Internal	*Initialisation:* Ensuring the MCP configuration settings are as intended.	*Interference:* Mitigating the potential effects of shared resource use.
External	*Integration:* Introduction of a "safety net" (or "safety monitor") external to the MCP.	*Environment:* Protecting against single event upsets, for example, due to cosmic rays.

In addition to the four combinations shown in Table 1, we explicitly consider a fifth item, specifically, **malicious** activity. Examples from the perspective of an MCP are shown in Table 2. This demonstrates that malicious activity can be associated with all four of the previously identified combinations, which may suggest that a specific malicious item is unnecessary. We have considerable sympathy with that view, especially as we strongly favour greater integration between, historically separate, safety and security activities. However, on balance, we believe the explicit security-related focus provided by the

malicious item is beneficial. Note that we explicitly cover malicious activity for the substrate because it differs significantly between substrates. (At the level of detail considered in this paper, there is much more commonality in the types of malicious activity that are associated with the program level.)

Table 2 ~ Examples of Malicious Activity for an MCP

	Static	**Dynamic**
Internal	*Malicious:* Hardware-based spyware that operates continuously.	*Malicious:* Hardware-based Trojan that is triggered by specific conditions.
External	*Malicious:* Undervolting a processor, to break secure enclave protection.	*Malicious:* Generation of external electro-magnetic fields to induce faults.

For **CNB** behaviours, we seek to understand how the program developer may come to expect impossible ("**could never be**") substrate behaviour. We identify three potential sources for the mistaken expectation. Working from a more general perspective to a more specific one, these are:

- Expectations based on previous **experience**. For example, a developer used to working with single core processors may assume exclusive use (within a given time period) of peripherals. This might not be true for an MCP-based implementation.
- Expectations based on substrate **documentation**. These expectations could be mistaken, for example, because of errata or the presence of undocumented features (Domas 2017).
- Expectations based on **development** processes. Differences between the host, i.e. development, and target, i.e. operational, systems are a common source of mistaken expectations.

2.4 Outline Comparisons with Other Standards

We stress that our approach is not meant to replace existing, or emerging, technology-specific standards (or standards-like documents). Where technology-specific information is available, this should provide a more detailed description of relevant assurance topics (and how these may be addressed) than is possible for any generic approach. Technology-specific standards can describe both requirements and associated acceptable means of compliance. In contrast, our generic approach can only outline requirements, in the form of topic areas to be discussed.

As suggested above, our approach is intended to be used when no technology-specific standards are available. Since they satisfy different aims, a detailed comparison of our approach with existing technology-specific standards is not appropriate. Nevertheless, a top-level, outline comparison is considered beneficial, for two reasons. Firstly, it provides an additional description of our approach and how it may be used. Secondly, it provides confidence that the identified topic areas are both necessary and sufficient.

Consequently, our approach has been compared with CAST-32A (CAST 2016), which considers MCP, and the computation-level framework of SCSC-153A (SCSC 2020), which considers artificial intelligence implemented using machine learning techniques.

Details of these outline comparisons are in Appendix A (Table 6 and Table 7, respectively).

The mapping to CAST-32A illustrates that all but one of CAST-32A's objectives can be mapped to our generic topic areas. The exception is that CAST-32A explicitly asks for an accomplishment summary, whereas communication mechanisms are not explicitly detailed in our approach. This mapping also shows that each of our areas maps to at least one CAST-32A objective.

The mapping to the SCSC-153A computation level shows that all of SCSC-153A's objectives can be mapped to our generic topic areas. It also shows that each topic area maps to at least one objective. It is noteworthy that our substrate topic areas, of which there are eight, correspond to only two of the nineteen SCSC-153A objectives. This reflects the greater prominence given to the substrate in our work, and is deemed appropriate, since this is an area of significant technological development.

The simple nature of these mappings means it would be inappropriate to read too much into their results. Nevertheless, they provide some confidence that the topic areas identified in our work are both necessary and sufficient for discussing assurance-related issues associated with new programmable content technologies. Further confidence is provided by the examples discussed in the following section.

3 Example Application

3.1 Example Programmable Content Technologies

To illustrate our topic areas, we apply them to three program/substrate combinations:

- An algorithm developed using Machine Learning (ML), running on a large-scale, complex System-On-Chip (SOC);
- An algorithm running on a quantum computer; and
- An algorithm running on a bio-based substrate.

The first example represents a technology that is available commercially now. The second and third technologies represent items that are being actively researched and where the associated substrates have very different properties to silicon-based electronics. We suggest that this range of examples (together with the previous MCP-related commentary) illustrates the generic nature of our approach. We note, however, that many other types of substrate could have been considered, e.g. neuromorphic processors (Davies et al. 2018), and memory-based compute (Bearden et al. 2020). The large number of potential examples is a key motivation for this paper.

The three examples are considered in the following subsections. Each subsection begins with a brief overview of the program and substrate. This is followed by a discussion of some of the key assurance points. Note that these discussions are not intended to be complete. Their purpose is not to act as a ready-made argument that can be deployed for any use of the associated substrate. Instead, the discussions are intended to demonstrate the way our generic assurance topics highlight key assurance aspects.

3.2 An Algorithm Developed using ML, Running on a SOC

Overview: This example uses Commercial Off-The-Shelf (COTS) hardware in the context of an autonomous vehicle. More specifically, the substrate is an NVIDIA Jetson AGX Xavier SOC (NVIDIA Corporation 2021). This includes: a 512-core Graphical Processing Unit (GPU); an 8-core Central Processing Unit (CPU); two Deep Learning Accelerator (DLA) engines; a Very Long Instruction Word (VLIW) vision processor; Double Data Rate (DDR) 4 memory; and embedded Multi-Media Card (eMMC) storage.

We consider a program implemented as a Deep Neural Network (DNN), which performs object recognition on camera images. This program was developed using a combination of tools, including: the Jetson AGX Xavier Developer Kit; TensorRT; cuDNN; and TensorFlow. Training data was obtained from a fleet of camera-equipped vehicles. The program also includes the Board Support Package (BSP) and the OS, which in this case is Linux for Tegra (L4T).

Our considerations for this example draw heavily on (Ashmore and Sharp 2020), (Ashmore et al. 2021) and (SCSC 2020).

Program Level: For the purposes of this paper, it is assumed that the BSP and OS are largely traditional, so they are not considered further. In practice the BSP and OS are often an important part of the overall COTS package. As such, their influence on the behaviour of the substrate may require a thorough analysis.

ML is typically used to solve open problems, which do not have a complete set of accurate, consistent, and verifiable requirements. Indeed, if such a set of requirements is available then traditional software development techniques may be preferred (Salay and Czarnecki 2018).

When ML is used, protection against **requirements are misunderstood** may be achieved, at least in part, by:

- Ensuring training data is relevant to the object recognition task: for example, a training data set comprising German language road signs would not be relevant for an algorithm deployed on UK roads.
- Ensuring model behaviour is interpretable, including behaviour on a single input (local interpretability) and behaviour on classes of input (global interpretability).
- Ensuring independent verification activities provide suitable coverage of the inputs likely to be received during operational use.

Protection against **some expected behaviour is not present** may be achieved, at least in part, by:

- Ensuring measures of model performance adequately capture required behaviour. For example, some misclassifications may be more important than others: misclassifying a pedestrian as a road marking is likely to be worse than mistaking a 50 mph speed limit sign for a 30 mph limit. Consequently, a model that had similar misclassification rates, regardless of the classes involved, would not be demonstrating the expected behaviour.
- Ensuring test environments are sufficiently representative of the real world. This can be challenging, especially for open problems, where it is difficult to know which parts of reality need to be sufficiently represented.

Protection against **some unexpected behaviour is present** may be achieved, at least in part, by:

- Ensuring training data is suitably complete. For example, covering a sufficiently wide collection of classes, and sub-classes (Paterson and Calinescu 2019), of object types, observed in a sufficiently wide range of meteorological and illumination conditions.
- Ensuring testing on the target hardware covers, either exhaustively or via appropriate sampling, the training, test, and verification sets. Also, ensuring that this testing covers robustness issues, e.g. arithmetic overflow, numerical accuracy.
- Ensuring independent verification activities provide suitable coverage of inputs that may be received following failures elsewhere in the system, e.g. camera issues.

Substrate Level: The substrate is a COTS item. This means the programmable content developer is likely to, at best, have limited influence over its design features. It also means that the developer is likely to have only limited information about the SOC's implementation-level features. (A similar situation prevails for many modern processors, including MCPs, where detailed implementation-level feature information is unlikely to be available.)

From an **initialisation (CBBI - static, internal)** perspective, documenting and justifying the desired SOC configuration is important. This includes, but is not limited to, control of debug features and microcode updates.

From an **interference (CBBI - dynamic, internal)** perspective, runtime monitoring of the SOC's configuration is important (especially for software-configurable items). The potential effects of other software running alongside the program under test may also be important.

From an **environment (CBBI - dynamic, external)** perspective there may be a need to protect against the effect of active sensors on the autonomous vehicle, or other vehicles. For example, the combined effect of multiple, active sensors in a cluttered urban environment may need to be considered.

From an **integration (CBBI - static, external)** perspective, (system) integration activities providing for multiple processing channels, running on separate substrates, is an important consideration. For example, the object recognition channel (the subject of the current discussion) may be combined with a much simpler object detection channel. When multiple channels are used, the degree of independence needs to be carefully considered.

From a **malicious** perspective, the COTS nature of the substrate makes it extremely difficult to protect against the introduction of a hardware-based Trojan. Thinking about the standard cyber security triad of Confidentiality, Integrity and Availability (CIA), availability should be detectable and manageable by traditional safety measures, similar to a hardware failure of the SOC. Multiple processing channels should be used to detect loss of integrity. Confidentiality is very difficult to protect at the substrate-level. System-level architectural designs, which, from a confidentiality perspective, treat the SOC as an "untrusted box" may be an appropriate way of mitigating this risk.

From an **experience (CNB)** perspective, a developer may be unfamiliar with the combination of features available on the SOC. In particular, the combination of CPU, GPU and DLA may mean that the developer's program is executed on an unexpected part of the substrate. This risk may be exacerbated if different SOC workloads result in different execution patterns.

From a **documentation (CNB)** perspective, the possibility of undocumented features and document errata should be considered. The latter may be protected against by careful monitoring of information provided by the substrate manufacturer. Where specific features of the substrate are of particular importance, a "trust but verify" approach may be appropriate.

From a **development (CNB)** perspective, the SOC chosen for this example may be sufficiently powerful to support both development and operational use, thus removing host-target differences. Alternatively, these differences may be important, if development is conducted on a GPU, but operational inference is conducted on a DLA. One way of mitigating this concern would be to conduct all post-training evaluation on the intended target hardware.

3.3 An Algorithm Running on a Quantum Computer

Overview: We begin by noting that, if it is successfully commercialised (Dyakonov 2019), quantum computing is most likely to be applied as a co-processing technology, i.e. alongside traditional computers.

To illustrate the utility of quantum computing, it is helpful to consider different categories of problem (or application). For presentational reasons, we are deliberately imprecise: a more precise description is provided in (Gheorghiu et al. 2019). As shown in Table 3, we identify three separate classes, depending on whether a problem can easily be solved, and whether a solution can easily be checked, using traditional computers.

Table 3 ~ Computation Classes Relevant to Quantum Computation

Using Traditional Computers ...	**Class 1**	**Class 2**	**Class 3**
... Easily Solved?	Yes	No	No
... Easily Checked?	Yes	Yes	No

For problems in Class 1, there is no need to invoke the additional complexity of a quantum computer. For problems in Class 2, provided the implications of a failed check can be handled safely, a significant amount of the assurance burden may be borne by using traditional computing to check a solution and, as such, assurance of the quantum portion may be less of a concern. Hence, problems in Class 1 and Class 2 are excluded from the current discussion.

To the best of our knowledge, the existence of Class 3 problems has not been proven. However, there are problems that credibly can be claimed as being in this class. One example requires calculation of a path that traverses a network (or graph) of a particular form (Childs et al. 2003).

A key aspect of a quantum computer is the notion of a "quantum bit", or qubit. Unlike a traditional bit, which holds a single value (either 0 or 1), a qubit holds a superposition of both 0 and 1; equivalently, a qubit holds a probability distribution over the space *{0, 1}*. Entanglement can be used to link qubits, so that the superposition of n qubits describes a probability distribution over a space containing 2n items. When a measurement is taken, the superposition collapses to a single item. The likelihood of receiving a particular measurement matches the associated probability encoded in the superposition.

Over recent years, there has been significant progress on quantum computing. For example, there is an open-source system, which includes a compiler, simulator and emulator (Steiger et al. 2018). This system includes a backend that links to cloud-based access to quantum computers, provided by IBM (IBM 2021). Other ways of interacting with these quantum computers are also available (IBM n.d.).

A simplified, but indicative, quantum computer system stack, based on (Fu et al. 2016), is illustrated in Table 4. This distinguishes program-level considerations and substrate-level considerations.

Table 4 ~ Simplified Quantum Computer System Stack

Stack	**Program / Substrate**
Quantum Algorithm	Program
Programming Paradigm / Language	
Compiler	
Quantum Instruction Set Architecture	Substrate
Quantum Execution / Error Correction	
Quantum Chip	

Given our current level of understanding, this example focuses on issues to be discussed, rather than on potential solutions to these issues.

Program Level: Our chosen example involves calculating a traversal path for graphs of a certain form. When considering **requirements are misunderstood**, it could be whether the program should calculate any path, or whether the shortest path should be calculated. As highlighted previously, quantum computing is inherently stochastic. Hence, requirements relating to the probability of obtaining the desired result (including whether results from sequential runs are statistically independent) may be misunderstood.

In terms of **some expected behaviour is not present**, test results may be influenced by inappropriate control of the test environment. This may be a particular concern if multiple tests are executed sequentially.

The program being considered is relatively simple in intent and, as such, offers little opportunity for cases where **some unexpected behaviour is present**. One possible example is different classes of network may be much easier to analyse. In some circumstances, significant changes in compute timing, for different classes of input, may be additional and unwanted behaviour.

Substrate Level: Quantum computation is an immature domain. Consequently, a specific example substrate is not discussed. Instead, we consider the quantum substrate in more general terms than that of the ML SOC.

From the perspective of **initialisation (CBBI - static, internal)**, we are concerned with the way that initial qubit values are established, as well as the way that qubit entanglements are created.

In terms of **interference (CBBI - dynamic, internal)**, decoherence and noise are significant challenges to the physical implementation of a quantum computer. These effects make it difficult to maintain specific values in, and entanglement between, qubits. Quantum Error Correction (QEC) methods are used to help mitigate these effects, at the expense of using a larger number of qubits.

With regards to **environment (CBBI - dynamic, external)**, current quantum computers require tight control of the environment, in particular to reduce the amount of environmental noise.

From the perspective of **integration (CBBI - static, external)**, the way that information is passed from a traditional computing resource to the quantum computer is of interest. Additionally, the number of times a quantum algorithm is repeated is another factor. Further, most QEC approaches rely on integration between a control mechanism implemented on a traditional computer and a quantum computer.

Malicious intervention into quantum cryptographic protocols is well-studied (Padmavathi et al. 2016). The potential vulnerabilities of quantum computers appear to be less well investigated. One obvious topic is an effective denial of service attack, caused by increasing the noise in the environment.

Quantum computer programs are based on a very different paradigm to traditional programs. For example, all steps in a quantum program need to be reversible, qubits cannot be copied, and no loops are permitted. (If necessary, loops can be achieved by repeated interaction between traditional and quantum computers, with the former handling the loop construct.) There are also cases where it is appropriate to use an undefined qubit in a calculation. These differences mean a developer may have unrealistic expectations based on previous **experience (CNB)**.

Inaccurate **documentation (CNB)** may create unrealistic expectations. One significant area involves understanding gate-level reliability, which depends on the QEC scheme employed, as well as the quantum computer's tolerance to external noise.

Unrealistic expectations may arise as part of **development (CNB)** due to inaccuracies in simulators and emulators used to debug quantum algorithms. Some of these, especially those relating to the final, pre-measurement superposition distribution, may be difficult to identify.

3.4 An Algorithm Running on a Bio-Based Substrate

Overview: This example considers molecular-based techniques, which exploit Chemical Reaction Networks (CRNs), implemented via Domain Strand Displacement (DSD) (Badelt et al. 2017). A brief introduction to CRNs is available in (Ashmore 2020).

A chemical reaction, in which reactants produce products at a given rate, is expressed in the general form:

$$Reactants \xrightarrow{Rate} Products \qquad \text{(Equation 1)}$$

Both *reactants* and *products* are more generally referred to as *species*. A CRN is a collection of related reactions.

In order to practically implement a CRN, fuel species are typically needed, e.g. to catalyse reactions, and waste species are often produced. Hence, a more complete description is:

$$Reactants + Fuel \xrightarrow{Rate} Products + Waste \qquad \text{(Equation 2)}$$

A number of implementations of CRNs have been demonstrated, including neural networks and solutions to the 3-SAT satisfiability problem (Winfree 2019).

For the purposes of our current discussion, we consider Equation 1 to be at the program level and Equation 2 to be at the substrate level; that level also includes, for example, the physical vessel in which the reactions occur.

Within this section, we assume the CRN is implemented in a bulk, well-mixed system rather than, for example, using a surface-based implementation (which, from our perspective, would be a different type of substrate). Where we need a specific example, we consider a combined CRN watchdog-oscillator system, where the watchdog can reset a failed oscillator (Ellis et al. 2019). However, much of the following focuses on generic CRN-related issues, rather than specific details of this example. This discussion is based on (Ellis et al. 2019), (Lutz et al. 2012) and (Johnson et al. 2019).

Program Level: In terms of **requirements are misunderstood**, CRNs are inherently stochastic, which needs to be appropriately reflected in the requirements. Requirements may also include implicit (and, consequently, misunderstood) assumptions regarding the fate, or future utility, of intermediate products. CRN behaviour is memoryless, stochastic, and asynchronous, so CRNs can be represented as Continuous Time Markov Chains (CTMCs). They are thus amenable to model checking, using tools like PRISM (Kwiatkowska et al. 2011), which can help detect ambiguous requirements.

Model checking can also provide a way of protecting against **some expected behaviour is not present**. CRNs also consume physical resources. In the case of our specific example, this may limit the number of times a broken oscillator may be restarted. Conversely, a user might expect unlimited restarts.

In CRNs, the concentration of a species, i.e. the amount present in the bulk mixture, can be viewed as analogous to a real-valued variable. More strictly, since concentrations cannot be negative, two species are needed to represent a real-valued variable: one for positive values and one for negative values. Including a set of reactions to cover negative values, when a variable can only take positive values, would be an example of **some unexpected behaviour is present** at the program level. Another example of unexpected behaviour could be including reactions to "tidy up" intermediary species, when not specified to do so in the requirements.

Substrate Level: In terms of **initialisation (CBBI - static, internal)**, key issues include the provision of sufficient quantities of reactant and fuel species. In addition, vessels used to contain the CRN implementation should be demonstrably free from contamination.

Concerning **interference (CBBI - dynamic, internal)**, in a bulk mixture all possible reactions occur simultaneously. This can make it challenging to schedule discrete steps within a program. Steps may be approximated with very different reaction rates (earlier steps having faster rates). Additionally, the products of a first step may be the reactants of a second step: this will ensure some of the first step completes before the second step begins; it will not ensure that the first step fully completes before the second step begins. Another approach involves using separate physical containers for each step: this obviously complicates the physical implementation (and integration). Another interference issue is unintended reactions, especially reactions that involve waste species.

From the perspective of the **environment (CBBI - dynamic, external)**, chemical reaction rates are sensitive to environmental conditions, like temperature and pressure. If these are not as intended, then the actual reaction rates may be very different to those anticipated. Additionally, the change in reaction rate may not be uniform across all reactions in a CRN, making sequencing of program steps even more challenging.

In terms of **integration (CBBI - static, external)**, a bio-based approach needs a mechanism for communicating the results of the computation. This is often achieved via molecules with different levels of luminescence. Using this type of approach to "read" the result of the computation is part of integration. For ongoing, continuous programs, e.g. monitors, integration may also involve ensuring there is a sufficient flow of fuel and reactants; removal of waste may also be a factor.

To the best of our knowledge, there has been little work on **malicious** interference with CRNs. Consequently, we merely highlight two theoretical possibilities. Firstly, we note that additional species could be inserted so that the program's behaviour changes after a certain number of oscillations (or, perhaps, a certain number of oscillator resets). Secondly, we note that the concentration of waste species could provide an adversary with valuable information, potentially undermining confidentiality.

From an **experience (CNB)** perspective, most programmers used to traditional electronics may be challenged by the notion that all possible reactions can occur simultaneously. The significant time taken for processing to complete (typically measured in hours) may be another relevant factor.

With regards to **documentation (CNB)**, the mapping from an abstract CRN, e.g. Equation 1, to an implementation CRN, e.g. Equation 2, means that, in essence, each substrate is program-specific. In traditional terms, the substrate is more like an Application-Specific Integrated Circuit (ASIC) than a general-purpose CPU. Hence, documentation relates to the way the relevant species are created and their resulting properties, e.g. amount of species fragments, or other unintended items.

In terms of **development (CNB)**, there are two main ways of analysing bulk mixtures. CTMCs account for stochastic behaviour and are amenable to model checking. Ordinary Differential Equations (ODEs) represent expected, average behaviour. This combination of methods may help protect against incorrect expectations. In addition, bisimulation approaches can provide confidence that an implementation CRN accurately reflects the intent of an abstract CRN (Johnson et al. 2019).

4 Summary

Table 5, overleaf, provides a very brief summary of the main points raised in the preceding examples. Depending on the nature of the example, these points may be mechanisms for providing assurance evidence, or issues that should be addressed by assurance evidence. For completeness, this table also includes a line on malicious activity at the program level.

The table demonstrates the potential applicability of our generic assurance topics to a wide range of programmable content. Whilst this is encouraging, more work is needed before these topics, broken down into program and application substrate, can become a formal recommendation. Completion of a number of specific worked examples, with sufficient detail to provide compelling assurance arguments, are an important part of that work.

Finally, motivated by the creation of specialised SOCs for ML tasks, and the broadening of computation away from traditional transistor-based architectures, we have highlighted the need for a set of generic assurance topics. These should enable a consistent approach to assuring elements within the ever-broadening domain of programmable content.

Table 5 ~ Summary of Generic Assurance Topics and Chosen Examples

		Developed Using ML, Running on a SOC	**Algorithm Running on a Quantum Computer**	**Algorithm Running on a Bio-Based Substrate**
Program	**Requirements are misunderstood**	Relevant training data; interpretable behaviour; independent verification	Requirements relating to probability of correct result; independence of sequential runs	Reflect stochastic nature; intermediate products; probabilistic model checking
	Some expected behaviour is not present	Adequate measures of model performance; sufficiently representative test environment	Control of the test environment, e.g. sequential tests	Probabilistic model checking; consumption of physical resources
	Some unnecessary behaviour is present	Sufficient testing (including robustness); independent verification	Significant, input-dependent changes in timing	Inclusion of species for negative values; tidy up of intermediate species
	Malicious activity	Insider threat; tool-based vulnerabilities	Insider threat; tool-based vulnerabilities	Insider threat; tool-based vulnerabilities

		Developed Using ML, Running on a SOC	Algorithm Running on a Quantum Computer	Algorithm Running on a Bio-Based Substrate
Substrate	**Initialisation (CBBI - static, internal)**	Document and justify SOC configuration	Initial qubit values and entanglements	Sufficient quantities of species; vessels free from contamination
	Interference (CBBI - dynamic, internal)	Runtime monitoring; effects of co-hosted software	Decoherence; noise; QEC	Sequencing steps; unintended reactions
	Environment (CBBI - dynamic, external)	Active sensors on own, and other vehicles	Noise	Effects on reaction rates
	Integration (CBBI - static, external)	Multiple processing channels	Passing information from traditional to quantum; number of re-runs	Reading the results; input flow of species; removal of waste
	Malicious activity	Hardware-based Trojans; multiple processing channels; untrusted "closed box" architecture	Denial of service by increasing environment noise	Insertion of additional species; concentration of waste species
	Experience (CNB)	Combination of processing features	Reversible; no qubit copying; no loops	Simultaneous reactions; long processing time
	Documentation (CNB)	Undocumented features; document errata; trust but verify	Gate-level reliability; QEC scheme	Way species are created
	Development (CNB)	Host-target differences; train on target	Inaccuracies in simulators and emulators	CTMCs; ODEs; bisimulation

Disclaimers

This document is an overview of UK MOD sponsored research and is released for informational purposes only. The contents of this document should not be interpreted as representing the views of the UK MOD, nor should it be assumed that they reflect any current or future UK MOD policy. The information contained in this document cannot supersede any statutory or contractual requirements or liabilities and is offered without prejudice or commitment.

Acknowledgments

The authors gratefully acknowledge the comments provided by the anonymous reviewers. These led to a significantly improved paper.

References

Ashmore, R. (2020). Urban Maths - Best of Both Worlds. *Mathematics Today, June 2020.* Retrieved from: https://ima.org.uk/14267/urban-maths-the-best-of-both-worlds/. Accessed 14th December 2021.

Ashmore, R., & Lennon, E. (2017). Progress Towards the Assurance of Non-traditional Software. In Parsons, M., & Kelly, T. (Eds.) *Developments in System Safety Engineering, Proceedings of the 25th Safety-critical Systems Symposium, February 2017*. Independently Published.

Ashmore, R., & Sharp, J. (2020). Assurance Argument Elements for Off-the-shelf, Complex Computational Hardware. In Casimiro, A., Ortmeier, F., Bitsch, F., & Ferreira, P. (Eds.). *Computer Safety, Reliability, and Security: 39th International Conference, SAFECOMP 2020, Lisbon, Portugal, September 16–18, 2020, Proceedings* (pp. 260-269). Springer.

Ashmore, R., Calinescu, R., & Paterson, C. (2021). Assuring the Machine Learning lifecycle: Desiderata, methods, and challenges. *ACM Computing Surveys (CSUR), 54(5),* 1-39. https://doi.org/10.1145/3453444

Badelt, S., Shin, S. W., Johnson, R. F., Dong, Q., Thachuk, C., & Winfree, E. (2017). A General-Purpose CRN-to-DSD Compiler with Formal Verification, Optimization, and Simulation Capabilities. In Brijder, R., & Qian, L. (Eds). *DNA Computing and Molecular Programming: 23rd International Conference, DNA 23, Austin, TX, USA, September 24–28, 2017* (pp. 232-248). Springer, Cham, Switzerland. https://doi.org/10.1007/978-3-319-66799-7_15

Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamarić, Z., & Ryzhyk, L. (2017). System Programming in Rust: Beyond Safety. In *Proceedings of the 16th Workshop on Hot Topics in Operating Systems, May 2017* (pp. 156-161). https://doi.org/10.1145/3102980.3103006

Bearden, S.R., Pei, Y.R., & Di Ventra, M. (2020) Efficient solution of Boolean satisfiability problems with digital memcomputing. *Scientific Reports, 10(1),* 1-8. Retrieved from: https://www.nature.com/articles/s41598-020-76666-2 Accessed 20th January 2022.

CAST, the Certification Authorities Software Team (2016). *Multi-core Processors.* Position Paper CAST-32A, November 2016

Childs, A.M., Cleve, R., Deotto, E., Farhi, E., Gutmann, S., & Spielman, D. A. (2003). Exponential algorithmic speedup by a quantum walk. In *Proceedings of the thirty-fifth annual ACM symposium on Theory of computing, June 2003* (pp. 59-68) https://doi.org/10.1145/780542.780552

Crawley, F., & Tyler, B. (2015). *HAZOP: Guide to best practice*. Elsevier, 3rd edition (21 April 2015)

Davies, M., Srinivasa, N., Lin, T. H., Chinya, G., Cao, Y., Choday, S.H., Dimou, G., Joshi, P., Imam, N., Jain, S., & Liao, Y. (2018). Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro, 38(1), January 2018,* 82-99 https://doi.org/10.1109/MM.2018.112130359

Dixit, H.D., Pendharkar, S., Beadon, M., Mason, C., Chakravarthy, T., Muthiah, B., & Sankar, S. (2021) *Silent Data Corruptions at Scale.* arXiv. Retrieved from: https://arxiv.org/pdf/2102.11245v1.pdf Accessed 20th January 2022.

Domas, C. (2017*). Breaking the x86 ISA.* Black Hat, USA. Retrieved from: https://www.blackhat.com/docs/us-17/thursday/us-17-Domas-Breaking-The-x86-Instruction-Set-wp.pdf Accessed 20th January 2022.

Dyakonov, M. (2019). When will useful quantum computers be constructed? Not in the foreseeable future, this physicist argues. Here's why: The case against: Quantum computing. *IEEE Spectrum, 56(3), March 2019,* (pp 24–29) https://doi.org/10.1109/MSPEC.2019.8651931

Ellis, S.J., Klinge, T.H., Lathrop, J.I., Lutz, J.H., Lutz, R.R., Miner, A.S., & Potter H.D. (2019). Runtime Fault Detection in Programmed Molecular Systems. *ACM Transactions on Software Engineering and Methodology (TOSEM), 28(2) April 2019, Article No.: 6,* (pp 1–20) https://doi.org/10.1145/3295740

Fu, X., Riesebos, L., Lao, L., Almudever, C.G., Sebastiano, F., Versluis, R., Charbon, E., Bertels, K. (2016). A heterogeneous quantum computer architecture. In *Proceedings of the ACM International Conference on Computing Frontiers, May 2016,* (pp 323–330) https://doi.org/10.1145/2903150.2906827

Gheorghiu, A., Kapourniotis, T., Kashefi, E. (2019). Verification of quantum computation: An overview of existing approaches. *Theory of Computing Systems, 63(4), May 2019* (pp.715-808) https://doi.org/10.1007/s00224-018-9872-3

Goodin, D. (2017). Apple scrambles after 40 malicious "XcodeGhost" apps haunt App Store. *Ars Technica.* Retrieved from: https://arstechnica.com/information-technology/2015/09/apple-scrambles-after-40-malicious-xcodeghost-apps-haunt-app-store/ Accessed 20th January2022.

Hawkins, R., Habli, I., & Kelly, T. (2013). The principles of software safety assurance. In *31st International System Safety Conference, 2013*. Retrieved from: https://www-users.cs.york.ac.uk/rhawkins/papers/HawkinsISSC13.pdf Accessed 20th January 2022.

Holloway, C. M. (2019). *Understanding the overarching properties.* NASA Technical Memorandum NASA/TM-2019-220292. Retrieved from: https://ntrs.nasa.gov/api/citations/20190029284/downloads/NASA-TM-2019-220292Replacement.pdf Accessed 20th January 2022.

IBM. (2021) *IBM Quantum Experience.* Retrieved from: http://research.ibm.com/quantum/, Accessed 14th December 2021.

IBM. (n.d.) *Qiskit: Open-Source Quantum Development*. Retrieved from: https://qiskit.org/. Accessed 14th December 2021.

Johnson, C. (2016). Role of Regulators in Safeguarding the Interface between Autonomous Systems and the General Public. In Hewett, J. (Ed.). *Proceedings of the 34th International System Safety Conference, Orlando, USA 8-12 August 2016* Retrieved from: http://www.dcs.gla.ac.uk/~johnson/papers/ISSC16/regulator.pdf Accessed 20th January 2022.

Johnson, R., Dong, Q., & Winfree, E. (2019). Verifying chemical reaction network implementations: a bisimulation approach. *Theoretical Computer Science, 765,* (pp.3-46). Retrieved from: https://authors.library.caltech.edu/96464/1/1-s2.0-S0304397518300136-main.pdf Accessed 20th January 2022.

Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., & Sewell, T. (2009). seL4: Formal verification of an OS kernel. In *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles, October 2009,* (pp. 207-220) https://doi.org/10.1145/1629575.1629596

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In Gopalakrishnan, G., & Qadeer, S. *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*. (pp. 585-591). Springer. Berlin, Heidelberg, Germany.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. (2016). CompCert-a formally verified optimizing compiler. In *Proceeding of the 8th European Congress on Embedded Real Time Software and Systems, ERTS 2016, Toulouse, France* Retrieved from: https://www.researchgate.net/publication/293814383_CompCert_-_A_Formally_Verified_Optimizing_Compiler Accessed 20th January 2022.

Lutz, R. R., Lutz, J. H., Lathrop, J. I., Klinge, T. H., Mathur, D., Stull, D. M., Bergquist, T. G., & Henderson, E. R. (2012). Requirements analysis for a product family of DNA nanodevices. In *2012 20th IEEE International Requirements Engineering Conference (RE)* (pp. 211-220). https://doi.ieeecomputersociety.org/10.1109/RE.2012.6345806

Murdock, K., Oswald, D., Garcia, F.D., Van Bulck, J., Gruss, D., & Piessens, F. (2020). Plundervolt: Software-based fault injection attacks against Intel SGX. In *2020 IEEE Symposium on Security and Privacy (SP)* (pp. 1466-1482). https://doi.ieeecomputersociety.org/10.1109/SP40000.2020.00057

Mutlu, O., & Kim, J. S. (2019). RowHammer: A retrospective. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 39(8), August 2020.* 1555-1571 https://doi.org/10.1109/TCAD.2019.2915318

NVIDIA Corporation. (2021). *Jetson AGX Xavier Developer Kit.* Retrieved from: https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit. Accessed 14th December 2021.

Padmavathi, V., Vardhan, B. V., & Krishna, A. V. N. (2016). Quantum cryptography and quantum key distribution protocols: a survey. In *2016 IEEE 6th International Conference on Advanced Computing (IACC)* (pp. 556-562). https://doi.org/10.1109/IACC.2016.109

Paterson, C., & Calinescu, R. (2019). *Detection and mitigation of rare subclasses in neural network classifiers.* arXiv preprint arXiv:1911.12780, 2019. Retrieved from: https://www.researchgate.net/profile/Colin-Paterson-4/publication/337671290_Detection_and_Mitigation_of_Rare_Subclasses_in_Neural_Network_Classifiers/links/5df770984585159aa4809742/Detection-and-Mitigation-of-Rare-Subclasses-in-Neural-Network-Classifiers.pdf Accessed 20th January 2022.

Peisert, S., Schneier, B., Okhravi, H., Massacci, F., Benzel, T., Landwehr, C., Mannan, M., Mirkovic, J., Prakash, A., & Michael, J. B. (2021). Perspectives on the SolarWinds Incident. *IEEE Security & Privacy, 19(02),* (pp. 7-13). Retrieved from: https://www.computer.org/csdl/magazine/sp/2021/02/09382367/1saZVPHhZew Accessed 20th January 2022.

RTCA. (2000). *Design Assurance Guidance for Electronic Hardware*. RTCA/DO-254, RTCA, Inc. Also available as EUROCAE Document ED-80.

RTCA. (2011a). *Software Considerations in Airborne Systems and Equipment Certification*. RTCA/DO-178C, RTCA, Inc. Also available as EUROCAE Document ED-12C.

RTCA. (2011b). *Model-Based Development Supplement*. RTCA/DO-331, RTCA, Inc. Also available as EUROCAE Document ED-218.

RTCA. (2011c). *Formal Methods Supplement to DO-178C and DO-278A*. RTCA/DO-333, RTCA, Inc. Also available as EUROCAE Document ED-216.

RTCA. (2015). *Standards for Processing Aeronautical Data*. RTCA/DO-200, RTCA, Inc. Also available as EUROCAE Document ED-76.

Salay, R., & Czarnecki, K. (2018). *Using machine learning safely in automotive software: An assessment and adaption of software process requirements in ISO 26262*. arXiv preprint arXiv:1808.01614, 2018 Retrieved from: https://arxiv.org/ftp/arxiv/papers/1808/1808.01614.pdf Accessed 20th January 2022.

SCSC, Safety Critical Systems Club C.I.C. (2020). *Safety assurance objectives for autonomous systems*, SCSC-153A. Retrieved from: https://scsc.uk/r153A:1 Accessed 20th January 2022.

SCSC, Safety Critical Systems Club C.I.C. (2021). *Data Safety Guidance*. SCSC-127F. Retrieved from: https://scsc.uk/r127F:1 Accessed 20th January 2022.

Steiger, D. S., Häner, T., & Troyer, M. (2018). ProjectQ: an open source software framework for quantum computing. *Quantum*, *2*, p.49 https://doi.org/10.22331/q-2018-01-31-49

Taber, A., & Normand, E. (1993). Single event upset in avionics. *IEEE Transactions on Nuclear Science, vol. 40, no. 2, pp. 120-126, April 1993,* https://doi.org/10.1109/23.212327

Thompson, K. (2007). Reflections on trusting trust. In *ACM Turing Award Lectures* (p. 1983), January 2007. Association for Computing Machinery. New York, USA https://doi.org/10.1145/1283920.1283940

Watson, R. N. M., Moore, S. W., Sewell, P., & Neumann, P. G. (2019). *An introduction to CHERI*. University of Cambridge, Computer Laboratory, Technical Report Number 941, UCAM-CL-TR-941, ISSN 1476-2986

Winfree, E. (2019). Chemical reaction networks and stochastic local search. In *International Conference on DNA Computing and Molecular Programming* (pp. 1-20). Springer. Cham Switzerland.

Appendix A. Outline Comparison Mappings

Table 6, below, provides an outline mapping between the objectives listed in CAST-32A and the generic topic areas identified in this paper. Table 7 provides a similar mapping between the objectives in the computation-level framework of SCSC-153A and the topic areas of this paper.

Table 6 ~ Outline Mapping from CAST-32A Objectives to Generic Topic Areas

CAST-32A Objective	**Program**			**Substrate**								**Count**
	Requirements are misunderstood	Some expected be-haviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
MCP_Planning_1				X	X		X			X	X	**5**
MCP_Resource_ Usage_1				X	X							**2**
MCP_Resource_ Usage_2					X	X		X				**3**
MCP_Planning_2				X	X		X			X		**4**
MCP_Resource_ Usage_3					X					X		**2**
MCP_Resource_ Usage_4					X		X					**2**
MCP_Software_1	X	X	X		X							**4**
MCP_Software_2	X	X	X		X				X	X		**6**
MCP_Error_ Handling_1							X	X				**2**
MCP_Accomplishment_ Summary_1												**0**
Count	**2**	**2**	**2**	**3**	**8**	**1**	**4**	**2**	**1**	**4**	**1**	**-**

Table 7 ~ Outline Mapping from SCSC-153A Computation-Level Objectives to Generic Topic Areas

SCSC-153A Computation-Level Objective	Program			Substrate								Count
	Requirements are misunderstood	Some expected behaviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
COM 1-1: Data is acquired and controlled appropriately.	X											**1**
COM1-2: Pre-processing methods do not introduce errors.		X	X									**2**
COM1-3: Data captures the required algorithm behaviour.		X										**1**
COM1-4: Adverse effects arising from distribution shift are protected against.			X									**1**
COM2-1: Functional requirements imposed on the algorithm are defined and satisfied.	X	X	X									**3**
COM2-2: Non-functional requirements imposed on the algorithm are defined and satisfied.	X	X	X									**3**
COM2-3: Algorithm performance is measured objectively.		X										**1**
COM2-4: Performance boundaries are established and complied with.		X										**1**
COM2-5: The algorithm is verified with an appropriate level of coverage.		X	X									**2**
COM2-6: The test environment is appropriate.		X										**1**
COM2-7: Each algorithm variant is tested appropriately.		X	X									**2**

SCSC-153A Computation-Level Objective	Program			Substrate								Count
	Requirements are misunderstood	Some expected behaviour is not present	Some unexpected behaviour is present	CBBI Initialisation (Static, Internal)	CBBI Interference (Dynamic, Internal)	CBBI Environment (Dynamic, External)	CBBI Integration (Static, External)	CBBI Malicious	CNB Expectation from Experience	CNB Expectation from Documentation	CNB Expectation from Development	
COM3-1: An appropriate algorithm type is used.		X	X									**2**
COM3-2: Typical errors are identified and protected against.			X									**1**
COM3-3: The algorithm's behaviour is explainable.	X	X	X									**3**
COM3-4: Post-incident analysis is supported.		X	X									**2**
COM4-1: The software is developed and maintained using appropriate standards.	X	X	X									**3**
COM4-2: Software misbehaviour does not result in incorrect outputs from the algorithm.			X									**1**
COM5-1: Appropriate computational hardware standards are employed.				X	X	X	X	X	X	X	X	**8**
COM5-2: Hardware misbehaviour does not result in incorrect outputs from the algorithm.					X	X		X				**3**
Count	**5**	**13**	**12**	**1**	**2**	**2**	**1**	**2**	**1**	**1**	**1**	**-**

Software Reliability and the Misuse of Statistics

Dewi Daniels[1], and Nicholas (Nick) Tudor[2]

1. Software Safety Limited, Trowbridge, UK
2. D-RisQ Ltd, Malvern, UK

Abstract

Many papers have been written on software reliability. The claim is made that failures of software-based systems occur randomly and that statistical techniques used to predict random hardware failure rates can also be used to predict software failure rates. This claim has not been challenged in any academic papers, though it is treated with suspicion by many practising engineers. As a result, the applicability of these statistical techniques has been accepted in some standards, such as IEC 61508, but has been rejected in others, such as RTCA/DO-178C. It is more important than ever to understand whether this claim is true. There is strong lobbying from industry to allow software not developed to any standard to be used for safety critical applications provided it has sufficient product service history. The European Union Aviation Safety Agency (EASA) is promoting dissimilar software in the belief that using two or more independent software teams will deliver ultra-high levels of software reliability. Software defects are different from random hardware failures and need to be treated differently. This paper argues that the techniques used for statistical evaluation of software make unwarranted assumptions about software and lead to overly optimistic predictions of "software failure rates". This paper concludes that many software reliability models do not provide results in which confidence can be placed. Instead, this paper proposes an alternative way forward that does provide evidence that software is safe for its intended use before it enters service.

1 Introduction

Many papers have been written on software reliability, including (Littlewood and Verrall 1973) and (Ladkin and Littlewood 2016). The claim is made that the same statistical techniques used to predict random hardware failure rates can also be used to predict software failure rates. While (Ladkin and Littlewood 2016) admits that "*It is true that software fails systematically, in that, if a program fails in certain circumstances, it will always fail when those circumstances are exactly repeated*", the authors go on to claim that, "*there is uncertainty about when a program will receive an input that will cause it to fai*l". They therefore conclude that software failures form a stochastic (random) process and that probabilistic models can be applied to software failures. A view is that as software is a component of a system and all components contribute to the reliability of the system, that software therefore has to have a failure rate. This reflects the demands of some reliability engineers using techniques such as Failure Mode and Effects Criticality Analysis (FMECA) and therefore also demands to have failure mechanisms for software.

The applicability of these statistical techniques is accepted by some industry standards, such as IEC 61508-3 (IEC 2010a), but rejected by others, such as RTCA/DO-178C (RTCA 2011).

© Software Safety Limited and D-RisQ Ltd 2022.

There is pressure from industry to allow commercial-off-the-shelf (COTS) and open source software to be accepted for use in safety critical systems based on product service history. For example, RTCA SC-240/EUROCAE WG-117 has been directed by RTCA Program Management Committee (PMC) and the EUROCAE Council (EC) to develop guidance on the integration of COTS, open source, and service history into software for airborne systems (RTCA 2021).

Furthermore, the European Union Aviation Safety Agency (EASA) is advocating dissimilar software development in the belief that the use of multiple, independent software teams will prevent common mode software failures (EASA 2014).

2 Hardware Reliability Modelling

The mechanical domain successfully uses statistical models. However, that world is markedly different from the one occupied by software. There are no similar failure mechanisms for software: there is no corrosion, fatigue, or fungus, let alone vagaries in manufacturing, which engineers account for by building in safety factors. It has been shown over many years that the use of modelling and statistics in techniques such as FMECA give well understood and predictable outcomes. These are models built up from knowledge of individual component failure mechanisms, their likelihood, and contribution to the overall system reliability. Looking for the contribution from the software domain, some mechanical engineers wish to have a number for reliability of the software component. Unfortunately, a response that it is either a '1' or a '0', does not tend to satisfy, so seeking a statistical approach to determining the contribution of software to overall system reliability is understandable perhaps, if one thinks of software as a component of a system, like all other components; unfortunately, it is not like other components, and needs to be treated differently, as we shall now discuss.

It should be noted that complex electronic hardware (CEH), on the other hand, can fail in very similar ways to software. Indeed, writing in VHDL (VHSIC [Very High-Speed Integrated Circuits] Hardware Description Language) is very similar in many ways to software programming. The issues raised in this paper could therefore be applied equally to CEH.

3 Software Reliability Modelling

3.1 Software Reliability in the Standards

3.1.1 Software Reliability in RTCA/DO-178C

RTCA/DO-178C (RTCA 2011) warns, "*Development of software to a software level does not imply the assignment of a failure rate for that software. Thus, software reliability rates based on software levels cannot be used by the system safety assessment process in the same way as hardware failure rates*". Concerning software reliability models, it states, "*Many methods for predicting software reliability based on developmental metrics have been published, for example, software structure, defect detection rate, etc. This document does not provide guidance for those types of methods, because at the time of writing, currently available methods did not provide results in which confidence can be placed*".

3.1.2 Software Reliability in IEC 61508

IEC 61508-3 (IEC 2010a) recommends probabilistic testing (highly recommended for software aspects of system safety validation at SIL 4). IEC 61508-7 (IEC 2010b) recognises that it is very difficult to demonstrate ultra-high levels of reliability using these techniques. (Littlewood and Strigini 1993) acknowledge that, where ultra-high dependability is required for software-based systems, it is not possible to confirm that a sufficiently high, numerically expressed dependability has been achieved.

Annex D of (IEC 2010b) describes a probabilistic approach to determining software integrity for pre-developed software. It is noted that Annex D is informative rather than normative. Table 1 below shows the number of failure-free demands experienced or hours of failure-free operation needed to qualify for a particular safety integrity level according to Annex D.

Table 1 ~ Necessary History from IEC 61508-7 Annex D

SIL	Low demand mode of operation	Number of treated demands		High demand or continuous mode of operation	Hours of operation in total	
	(Probability of failure to perform its design function on demand)	$1-\alpha = 0.99$	$1-\alpha = 0.95$	(Probability of a dangerous failure per hour)	$1-\alpha = 0.99$	$1-\alpha = 0.95$
4	$\geq 10^{-5}$ to $< 10^{-4}$	4.6×10^5	3×10^5	$\geq 10^{-9}$ to $< 10^{-8}$	4.6×10^9	3×10^9
3	$\geq 10^{-4}$ to $< 10^{-3}$	4.6×10^4	3×10^4	$\geq 10^{-8}$ to $< 10^{-7}$	4.6×10^8	3×10^8
2	$\geq 10^{-3}$ to $< 10^{-2}$	4.6×10^3	3×10^3	$\geq 10^{-7}$ to $< 10^{-6}$	4.6×10^7	3×10^7
1	$\geq 10^{-2}$ to $< 10^{-1}$	4.6×10^2	3×10^2	$\geq 10^{-6}$ to $< 10^{-5}$	4.6×10^5	3×10^6

NOTE 1 $1-\alpha$ represents the confidence level.

NOTE 2 See [IEC 61508-7] D.2.1 and D.2.3 for prerequisites and details of how this table is derived

3.2 Applicability of Models

While models can be useful in various circumstances, users have to be careful to ensure that the chosen model is applicable and that the limitations of the model are understood when drawing conclusions. (Mandelbrot and Hudson 2004) predicted the stock market crash that occurred in 2008. (Mandelbrot and Hudson 2004) claimed that the mathematical models used were flawed and that it was mistaken to assume that the normal distribution was a useful model for tracking price changes in the stock markets. Most economists responded that independence and normality are just assumptions that help simplify the mathematics. However, the inappropriate application of the normal distribution underestimated the probability that many borrowers would default on their subprime mortgages at the same time.

One must be careful of the "tail" in the models. If the application of the model underestimates the probability of an extreme event, then the consequences of decisions made to defend against such an event could be far reaching. (Mandelbrot and Hudson 2004) claimed that stock market prices follow a power law rather than the normal distribution. Power laws have fatter tails than the normal distribution. A statistical model based on the normal distribution would have underestimated the probability of unlikely events such as stock market crashes.

There are many examples of incorrect application of modelling to problems. Indeed, a quote generally attributed to George Box is, "All models are wrong, but some are useful", which tries to explain that models are not the real artefact but may nevertheless be useful. The scientist, engineer, forecaster must therefore ensure that the assumptions behind the use of the model are the right assumptions to be making and which must be understood before valid conclusions can be made. Without this understanding, a false premise can lead to almost any conclusion one would like, or indeed, not like.

3.3 Assumptions

3.3.1 Preamble

Many software reliability models assume that software execution is a Bernoulli process. It is therefore assumed that:

1. Executing a software program results in one of two outcomes, which we term Success or Failure; and
2. The probability of Success is the same every time the software is executed.

The first of these assumptions seems a reasonable assumption. The software either produces the correct output or it does not. We accept it is not always easy to determine whether an output is correct or not, but we will assume for now that we can do this. The second of these assumptions is an unwarranted assumption in most cases, as we explore in the following examples.

3.3.2 Example 1: State

Most software programs contain state. This could be in the form of a variable stored in Random Access Memory (RAM), a record stored on a Solid-State Drive (SSD), a Hard Disk Drive (HDD), in the Cloud, or even as the contents of a memory cache. State essentially introduces bias into the system. A fair coin or die does not have "memory". Software state is worse than a biased coin because the bias could change from one trial to the next. Suppose a software program increments an unsigned 16-bit counter. When the counter reaches 65,535, it wraps around to zero, causing the program to produce the wrong output. The probability of success is not the same every time the software is executed. Assuming there are no other defects, the probability of success the first 65,535 times the software program is executed is one; the probability of success the $65{,}536^{th}$ time the software program is executed is zero. If we had executed the software program 65,535 times and observed it produce the correct output each time, we would have a high degree of confidence that the software program is correct, yet it will fail the very next time it is invoked.

The presence of state means that whether invoking a software program results in success or failure depends not just on the inputs presented at that time, but on previous invocations

of the software. If you are worrying about cache contents, then it also depends on other software that is running on the same hardware (even if we're using time slicing!). If the software halts the processor or enters an infinite loop, then all subsequent invocations will fail until the software is reset. More subtly, a previous invocation of the software could have resulted in the state changing in such a way that a future invocation of the software will fail. This means that the probability of success is not the same every time the software is executed.

3.3.3 Example 2: Environment

The mathematical models assume that the probability of success is the same every time the software is executed. How can the probability of success be the same when the software is dependent on its environment, and that environment is continually changing, often in subtle ways? Proponents of software reliability modelling may argue that it is simply a matter of ensuring that the software is exercised in its operational environment. However, it is very difficult to identify all the dependencies of the software on its environment. Failure to do so can lead to misplaced confidence in the software's reliability.

Suppose a software program does not handle 29 February correctly in a leap year. Assuming there were no other defects, had we started running the program on 1 March 2016, then by 28 February 2020 we would have observed 35,040 hours of failure free operation. We would have had a high degree of confidence that the software is correct, yet it would have failed the very next day, 29 February 2020. Software reliability models attempt to predict future software behaviour based on observations of past software behaviour. There is a saying, attributed to Niels Bohr but which is apparently an old Danish proverb, that, "it is difficult to predict, especially the future".

The Ariane 5 accident showed that even a small change in operational profile (from Ariane 4 to Ariane 5) can have an unexpected and catastrophic impact on software behaviour (O'Halloran 2005).

3.3.4 Example 3: Defect Distribution

How are defects distributed by component in software systems? (Hopkins and Hatton 2019) analysed the software defects in a numerical library written in Fortran. They found very strong evidence of defect clustering with typically 80% of all components in the library exhibiting no defect. Furthermore, software errors are more likely to occur at parameter limits and boundaries, which is why software testing techniques such as boundary value analysis are so effective.

Defect clustering is in fact so ubiquitous that it appeared 4th in the "top 10 defect reduction list" compiled by (Boehm and Basili 2001). It follows that if a defect is observed in the implementation of a software feature, it is likely there will be further defects in features implemented by the same software component. Defect clustering turns out to be the result of combining the assumption of uniform distribution of defects across lines of code (or more accurately tokens) with the asymptotic power-law distribution of component lengths which holds for all software systems (Hatton 2014). Suppose then we have a software application that contains ten software components, and that we are told that this software application contains ten defects. The most likely scenario given our prior knowledge is that typically 8 of the 10 components are defect free and the remaining 2 contain all 10 defects. Of course, we don't know which but, as soon as we encounter a defect in a component, we know it's likely to have more defects and we should change our strategy

accordingly. Accommodating this in software temporal failure models is by no means obvious, and has not to our knowledge been accomplished.

3.3.5 Example 4: Non-Operational Modes and Easter Eggs

Some software programs have different modes of operation. As well as the normal operational mode, there may be non-operational modes, such as a bootstrap mode, a software update mode, or a debug mode. Should the software inadvertently enter one of these non-operational modes, then it will stop working as expected.

As described in (Ladkin and Littlewood 2016), another variation is that some software contains what are known as Easter Eggs. These are features that are intended to surprise and delight the user. For example, both Microsoft® Excel 97 and Google Earth contain a flight simulator. The Excel 97 flight simulator is launched by creating a new worksheet, pressing F5, typing "L97:X97", pressing Enter, pressing Tab once, holding down Ctrl + Shift and left clicking the Chart Wizard toolbar icon. The Google Earth flight simulator is launched by typing Ctrl + Alt + A. If an Easter Egg is activated inadvertently, then again, the software will stop working as expected.

Easter Eggs is a term also used in the cyber security context. They are “surprises” that can be triggered by environmental conditions decided upon by a malicious actor. Should those conditions be met, the Easter Egg will execute. These conditions may not have been thought of by developers as unintentional conditions that trigger the surprise and, of course, would have been missed by test. Consequently, the software will work normally, possibly for many years, until the right conditions are met, and this evades statistical modelling.

3.3.6 Example 5: Duration of the Experiment

From a statistical point of view, it does not matter whether we run one copy of the software for a million hours or a million copies of the software for an hour each; both experiments result in one million hours of operation. In the real world, these two experiments are not the same. Running a million copies of the software for an hour each does not give us confidence that the software will run for two hours, let alone for days, months or years. Why do we expect the environment’s behaviour for one hour to be the same as the environment’s continuous behaviour for a million hours?

3.4 Limitations

There are some severe limitations that apply even if we accept that the assumptions are valid for a particular software application:

1. A very large number of hours of operation is required to produce a statistically significant result (Butler and Finelli 1993, Kalra and Paddock 2016).
2. Furthermore, the mathematical models used require that we observe no failures. If we observe a failure, we need to start the experiment again. Running a software program for 10,000 hours and observing no failures is not the same as running a software program for 10,000 hours, observing 10 failures and fixing the defects that caused those 10 failures. In both cases, there are no known defects at the end of the experiment. However, in the first case, we would have a high degree of confidence that if we ran the software for another 10,000 hours, we would still see no defects. In the second case, if we ran the software for

another 10,000 hours, we would expect to see further defects (they would just not be the same defects).

3. No software changes can take place during the experiment. If the software is changed in any way, the experiment must be restarted unless justification can be provided as to why the changes made do not invalidate the data collected up to that point. Indeed, this is the approach used in hardware reliability experiments such as those used for airframe stress tests.
4. The software reliability claim is only valid for the software version that was tested, running on the same hardware that was used in the test. If a new software version is released, the experiment must be repeated.
5. The operational profile must be identical.
6. There needs to be an effective system for observing, recording, and documenting faults.

3.5 Dissimilar Software

There is a belief that dissimilar software development will defend against common mode or common cause development errors in safety critical systems. In an experiment reported in (Knight and Leveson 1986), the programs were individually extremely reliable but that the number of tests in which more than one program failed was substantially more than expected. (Knight and Leveson 1986) showed that there were common mode errors introduced into software regardless of independence of people, hence diversity didn't achieve the desired outcome. These were common mode errors in the software implementation. Dissimilar software development would not have been able to protect against common mode errors in the software requirements, since the same software requirements were given to all the participants in the experiment.

In a later paper, (Knight and Leveson 1990) stated, "*Until N-version programming has been shown to achieve ultra-high reliability and/or has been shown to achieve higher reliability than alternative ways of building software, the claims that it does so should be considered unproven hypotheses. Until these hypotheses are shown to hold for controlled experiments, depending on N-version programming in real systems to achieve ultra-high reliability where people's lives are at risk seems to us to raise important ethical and moral questions. Attacking us or our papers will not change this*".

On the other hand, (Hatton 1997) concluded, "*The balance of data tends to suggest that N version techniques are preferable in software development when the cost of failure is high due to our inability to make one really good version whatever techniques we currently use. Even though the advantage is much less than that for N independent channels, the difference is still substantial and in this case gave a factor of 5–9 improvement on average for a majority voted 3 version system compared with a single version typical of the current state of the art*". However, this claim was based only on the data from the Knight and Leveson experiment and on the state of the art at the time. More work like (Hatton 1997) is needed in order to widen the samples, and to account for modern development and verification practices.

EASA is promoting dissimilar software. There is a difference of opinion between the FAA and EASA, so dissimilar software is currently the topic of a harmonization effort between the FAA and EASA. EASA have been keen to emphasise that they do not mandate dissimilar software development, but that they do require independence. It is worth examining in this paper, the basis for EASA's position.

At a meeting at which one of the authors was present, EASA stated that for flight controls, EASA will not accept full reliance on Development Assurance and Quality Assurance as

sole mitigation of a common mode leading to a total loss of flight controls. When asked, the EASA presenter stated this EASA position is documented in the EASA Generic Certification Review Item (CRI) on "Consideration of Common Mode Failures and Errors in Flight Control Functions" (EASA 2014). This Generic CRI states that the EASA position is that Development Assurance alone is not necessarily sufficient to establish an acceptable level of safety for Flight Control functions, as described specifically in CAST 24 (CAST 2006), and that mitigation means or techniques should be provided to protect against Common Mode Failures/Errors, including software development errors. (EASA 2014) does not specifically address dissimilar software development. (EASA 2014) is a draft CRI that has never been formally issued.

(CAST 2006) does discuss design diversity, dissimilar software, and N-version design. (CAST 2006) cites three papers on design diversity and N-version design, which are (Littlewood 1996), (Littlewood et al. 2000), and (Littlewood et al. 2001). (CAST 2006) has now been withdrawn. It is worth examining here, therefore, what these papers say:

1. (Littlewood 1996) concludes, "*In real applications of this work, it will be necessary to estimate what has actually been achieved, rather than to rely upon the more general results presented here. It is here that hardware engineers have a considerable advantage over their software counterparts. As long as there is sufficient failure data, from previous use of the different component types, that 'covers' the environments within which the new system will operate, the distributions that are needed to compute system reliability will be estimable. This contrasts with the software diversity situation, where estimability is severely constrained by the practical limitations to the number of versions that can be developed*".
2. (Littlewood et al. 2000) claims that, "*The idea that diversity may be a more cost effective way to deliver high diversity is alive and recently it was spelled out by Hatton*" (Hatton 1997). (Littlewood et al. 2000) concludes, "*it appears that Hatton's suggestion that design diversity is always going to be more cost effective than developing a single version software is not trustworthy. In order for us to be certain that diversity will bring more than it takes we need to measure the dependence, which is currently an open question*". We should note that Hatton did not actually suggest that design diversity is "always" going to be more cost effective. Hatton's claim was considerably more circumspect than claimed in (Littlewood et al. 2000) and, as noted earlier, it is recommended that more work like the (Hatton 1997) study is conducted.
3. (Littlewood et al. 2001) presents a theoretical model of hardware redundancy. This model assumes that the presence or absence of each fault (which is claimed to be a random event) is independent of the presence of any other fault, and of the presence in another version. They admit this is the novel assumption that allows them to predict distributions of the probability of failure per demand (pfd) of versions and systems. There is no basis for this assumption, even for hardware, other than that it is a convenience that makes the mathematics tractable. The assumption is certainly not valid for software, as is acknowledged as (Littlewood et al. 2001) concludes, "*However, we do not hide that many of these results are only useful for better understanding these complex, counter-intuitive problems: they do not lead to simple, general recipes for design and assessment. Such understanding is necessary before it is possible to begin engineering diverse fault-tolerant systems with dependability assurance founded on formal models*" and "*Difficulties remain in using some of this theory — most particularly in populating the models with estimates of their key parameters when dealing with real systems*".

In summary, the EASA position is based upon an unissued CRI, which itself was based upon CAST-24 (now withdrawn), which cites the three papers discussed above, which do not appear to support the EASA position.

3.6 Are the Models Correct? Are they Useful?

The proponents of software reliability claim that the advantage of assuming that software execution is a Bernoulli process is that the pertinent mathematics is simple, clear, and well understood. This may be so, but it does not follow that a Bernoulli process is an accurate (or even useful) model of software failures.

The use of a Bernoulli process to model software failures is fundamentally flawed. Treating software execution as a Bernoulli process assumes that the probability of success is the same every time the software is executed, which we have shown to be an unwarranted assumption.

This assumption *may* be warranted in specific circumstances. For example, the assumption might be warranted for a software interlock implemented in predicate logic with no state. Where the assumption is unwarranted, it leads to an exaggerated confidence in probabilistic testing and product service history.

3.7 Can the Models be Fixed?

(Hatton 2012) presented a mathematical model that hypothesises that software defects follow a power law distribution. This hypothesis was validated by an experiment using 55 million lines of source code. Just as improved financial models have been constructed that use power law distributions, it may be possible to create improved models of software defects using power law distributions. However, defect density is a static property of the software, while failure rates are the result of interaction between the software and its environment over time. While it might be possible to predict the probability of defects using improved models based on Hatton's work, that is not the same as predicting the probability of system failure as a result of the use of that software. If the conditions that trigger the defect are not present then the software will work, and the system will not fail; the converse is also true.

4 The Way Forward

4.1 Aerospace

(Littlewood and Strigini 1993) concluded that, "*The main point of our paper has been to try to show that here lies the main difficulty when we want to build a system with an ultra-high dependability requirement. Even if, by the adoption of the best design practices, we were to succeed in achieving such a dependability goal, it is our view that we would not know of our achievement at least until the product proved itself after a very great deal of operational use. Nor can we expect to overcome this problem with improved mathematical and statistical techniques*". (Littlewood and Strigini 1993) went on to state, "*Is there a way out of this impasse? One approach would be to acknowledge that it is not possible to confirm that a sufficiently high, numerically expressed dependability had been achieved, and instead make a decision about whether the delivered system is 'good enough' on different grounds. This is the approach currently adopted for the certification*

of software in safety-critical avionics for civil airliners". (Littlewood and Strigini 1993) claims, "*this falls far short of providing the assurance needed*".

RTCA/DO-178B (RTCA 1992) was published in 1992. It was superseded by RTCA/DO-178C (RTCA 2011), which made minimal changes to the core document, in 2011. There are now 27,012 commercial jet airplanes in service worldwide (Boeing 2021). No hull loss accidents in passenger service have been ascribed to failure of software developed to RTCA/DO-178B to implement its requirements correctly (Daniels 2011). Software has been a contributing factor in a small number of accidents and in-flight upsets. Modern aircraft and their software are extraordinarily safe (Rushby 2012). We now have nearly 30 years of service experience that satisfying the objectives of RTCA/DO-178B/C has been sufficient to address the software considerations in aircraft certification.

4.2 Proof and Economics

We contend that our ability to make one really good version has improved in the last 30 years through the introduction of improved software development and verification techniques. In particular, the processing power of modern CPUs has made automated program proof increasingly tractable. Formal methods allow us to verify that a software artefact is correct, complete, and unambiguous.

There are plenty of anecdotes that claim that formal methods will save developers time and money; some may even say that it is 'better' than any other approach. Indeed, (Rushby 2009) and (Littlewood and Rushby 2012) wrote that formal methods may have a part to play in providing higher assurance based upon the idea introduced by (Littlewood 1998) that software may be "possibly perfect". The general proposition is that, "possible perfection provides a bridge between the verification activities used to ensure correctness of software and the probabilistic estimates required for failure at the system level". Given that use of formal methods can qualitatively achieve 'at least as good software' as any other approach, the question is what evidence is there that it does save time and money?

It has been the case for a number of years that large developers, such as Google, Facebook, Amazon, and Microsoft have been recruiting formal methods expertise with the aim of being more robust (Garavel et al 2020). For example, (Ball et al. 2004) describes how Microsoft Research developed a Static Driver Verifier using formal methods. This tool is now widely used to verify third party Windows device drivers and has greatly reduced the number of Blue Screens of Death (BSOD). Again, this is a qualitative argument, but the market appears to have its view. While there exists the possibility that better software reliability models could be developed, the only techniques available at the present time that can predict accurately the behaviour of software before it is executed are formal methods. An approach therefore might be to use formal methods wherever practicable and to back this up with testing and/or simulation where needed in order to increase confidence.

The use of formal techniques, like any software development technique, has to be based upon assumptions about the environment and there are limits to what these techniques can, or indeed should be, used for (Murray and van Oorschot 2018).

4.3 An Industrial Experiment

An experiment, run by Warwick University Manufacturing Group in 2018, on automotive software showed the benefits of the use of formal techniques. The blind trial involved benchmarks of real automotive requirements and design for 6 functions that had been seeded with 48 errors. The trial measurement process was developed by York Metrics and

time taken for each stage of the verification process was recorded. There were 3 processes used independently by 3 different people: baseline human review, tool assisted using Simulink Design Verifier (SLDV) and a formal method-based tool developed by D-RisQ Ltd called Modelworks®, and in all 3 cases the 48 errors were detected. In the case of Modelworks®, a 49th error was discovered that had not been seeded. However, the time for Modelworks® to discover these errors was between 60–80% faster across all 6 cases measured, in one case reducing from 52 hours down to 10 hours; see Figure 1. It will be noted that there are 7 sets of measurements. Experiment PP failed to complete analysis as time for the project ran out, though Modelworks® found the relevant errors. The 7th case is a repeat of TA (i.e., TA2) by a different set of 3 people in a second company applying their own processes for the benchmark. It can be seen that the results are very close (but not the same) thus demonstrating consistency in the experimental results.

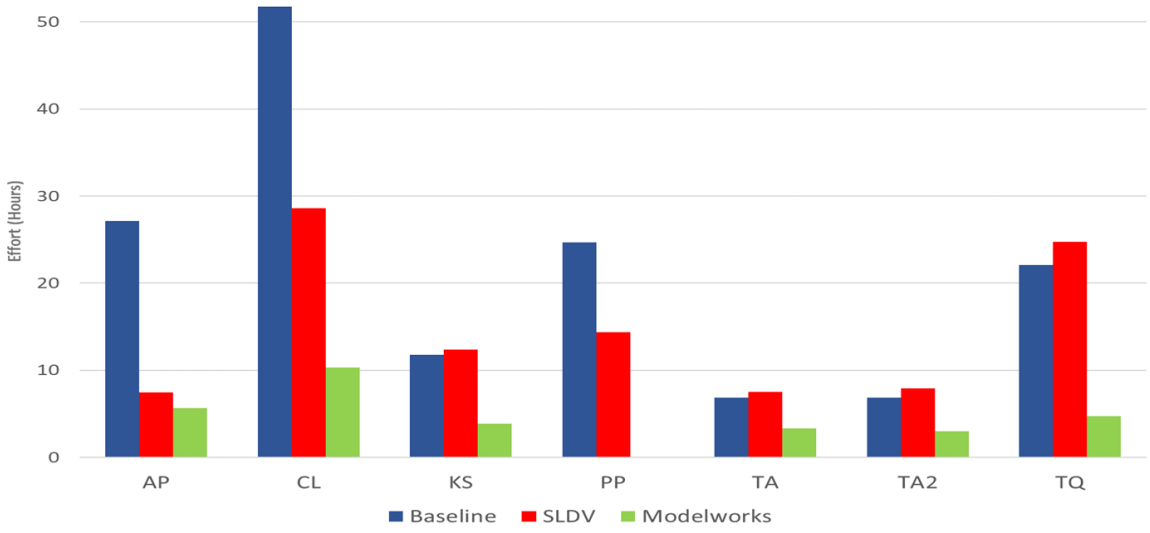


Figure 1 ~ Industrial Scale Benchmarking Study

4.4 Requirements Engineering

The weak link, for avionics software at least, remains requirements engineering. There has not been a single hull loss accident in passenger service that has been ascribed to failure of software developed to RTCA/DO-178B/C to implement its requirements correctly. There have been several accidents where software implemented the requirements correctly, but those requirements specified unsafe behaviour under some unforeseen circumstance. For example, an Airbus A320 overran the runway at Warsaw on 14 September 1993 (MCAAI 1994). A contributing factor was that deployment of the ground spoilers and engine thrust reversers was delayed because of a requirement to deploy them only when both main landing gear struts indicated Weight on Wheels. In another accident, an Airbus A320 overran the runway at Sao Paulo on 17 July 2007 (CENIPA 2009). A contributing factor was that the pilot only pulled one thrust lever into the reverse thrust position (the other thrust reverser was known to be inoperative), but a requirement stated that both thrust levers must be in the idle or reverse thrust position for either of the thrust reversers to be deployed. Finally, in the two recent Boeing 737 MAX accidents on 29 October 2018 and 10 March 2019, the Manoeuvring Characteristics Augmentation System (MCAS) software implemented its requirements correctly, but the requirements caused full nose down trim to be applied following an Angle of Attack sensor failure (Daniels 2020).

As Nancy Leveson has said, “Software-related accidents are usually caused by flawed requirements”. It therefore follows that our efforts should be focused on writing better requirements. Formal methods can help with writing better requirements by using formal requirements languages with unambiguous semantics and formal methods tools that can ensure the requirements are complete and consistent.

5 Conclusion

We have shown in this paper that software reliability models that assume that software execution is a Bernoulli process are flawed. The assumption of independence may be warranted in specific circumstances, but otherwise these models lead to an exaggerated confidence in probabilistic testing and product service history. These models cannot, in general, be used to demonstrate ultra-high reliability.

There seems to be an opportunity for the research community to investigate further the efficacy of N-version programming in mitigating the impact of software errors.

The situation is not as unsatisfactory as was stated in (Littlewood and Strigini 1993). We have now accumulated nearly 30 years of service experience that satisfying the objectives of RTCA/DO-178B/C has been sufficient to address the software considerations in aircraft certification.

The state of the art has improved considerably since RTCA/DO-178B was published in 1992. Formal methods are now used more widely, and automated program proof is now tractable and cost-effective.

Requirements continue to be the weak link. Analysis of historical aircraft accidents suggests that in those accidents where software was involved, the software implemented its requirements correctly, but the requirements specified behaviour that was unsafe and led to the accident.

We need to focus on getting the requirements right if we are to improve safety.

References

Ball, T., Cook, B., Levin, V. & Rajamani, S.K. (2004). *SLAM and Static Driver Verifier: Technology Transfer of Formal Methods inside Microsoft* (Technical Report MSR-TR-2004-08). Microsoft Research. Retrieved 22 September 2021 from https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/tr-2004-08.pdf

Boehm, B. & Basili, V. R. (2001). Software Defect Reduction Top 10 List. In *IEEE Computer. January 2001,* 135–137. Retrieved 8 November 2021 from https://www.cs.umd.edu/projects/SoftEng/ESEG/papers/82.78.pdf

Boeing. (2021). *Statistical Summary of Commercial Jet Airplane Accidents, Worldwide Operations 1959 – 2020*, Boeing. Retrieved 9 November 2021 from https://www.boeing.com/resources/boeingdotcom/company/about_bca/pdf/statsum.pdf

Butler, R.W., & Finelli, G.B. (1993). The infeasibility of quantifying the reliability of life-critical real-time software. In *IEEE Transactions on Software Engineering, 19(1)*, 3–12. Retrieved 28 September 2021 from https://shemesh.larc.nasa.gov/fm/papers/Butler-nonq-paper.pdf

CAST. (2006). *Reliance on Development Assurance Alone When Performing a Complex and Full-Time Critical Function* (Position Paper CAST-24 Rev 2). FAA Certification Authorities Software Team.

CENIPA. (2009). *Final Report A – No 67/CENIPA/2009.* Retrieved 12 November 2021 from http://sistema.cenipa.aer.mil.br/cenipa/paginas/relatorios/rf/en/3054ing_2007.pdf

Daniels, D. (2011). *Thoughts from the DO-178C committee*. Paper presented at the 6th IET International Conference on System Safety, 2011.

Daniels, D. (2020). *The Boeing 737 MAX Accidents*. Paper presented at the 28th Safety-Critical Systems Symposium, York, UK. Retrieved 9 November 2021 from https://scsc.uk/rp154.1:1

EASA. (2014). *Consideration of Common Mode Failures and Errors in Flight Control Functions* (Draft Generic CRI D-05), European Aviation Safety Agency.

Garavel, H., ter Beek. M. & van de Pol, J. (2020). *The 2020 Expert Survey on Formal Methods.* Paper presented at the 25th International Conference on Formal Methods for Industrial Critical Systems, Vienna, Austria. Retrieved 9 November 2021 from https://hal.inria.fr/hal-03082818/document

Hatton, L. (1997). *Are N average software versions better than 1 good version?* IEEE Software, SE-14(6), 71–76. Retrieved 22 September 2021 from https://www.leshatton.org/Documents/Nver_1297.pdf

Hatton, L. (2012). *Power-laws and the Conservation of Information in discrete token systems: Part 2 The role of defect.* Retrieved 17 September 2021, from https://arxiv.org/abs/1209.1652

Hatton, L. (2014). *Conservation of Information: Software's Hidden Clockwork?* IEEE Transactions on Software Engineering, 40(5), 450–460. Retrieved 8 November 2021, from https://www.leshatton.org/Documents/TSE-2013-08-0271_V1.0a.pdf

Hopkins, T.R. & Hatton, L. (2019). *Defect patterns and software metric correlations in a mature ubiquitous system.* Retrieved 17 September 2021, from https://arxiv.org/abs/1912.04014

IEC. (2010a). *Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 3: Software requirements.* International Electrotechnical Commission.

IEC. (2010b). *Functional safety of electrical/electronic/programmable electronic safety-related systems — Part 7: Overview of techniques and measures.* International Electrotechnical Commission.

Kalra, N., Paddock, S.M. (2016). *Driving to Safety: How Many Miles of Driving Would It Take to Demonstrate Autonomous Vehicle Reliability?* RAND Corporation. Retrieved 11 November 2021 from https://www.rand.org/pubs/research_reports/RR1478.html

Knight, J.C. & Leveson N.G. (1986). *An experimental evaluation of the assumption of independence in multi-version programming.* IEEE Transactions on Software Engineering, 12(1), 96–109. Retrieved 22 September 2021 from http://sunnyday.mit.edu/papers/nver-tse.pdf

Knight, J.C. & Leveson N.G. (1990). *A reply to the criticisms of the Knight & Leveson experiment.* ACM SIGSOFT Software Engineering Notes, *15*(1), 24–35. Retrieved 8 November 2021 from http://sunnyday.mit.edu/critics.pdf

Ladkin, P.B. & Littlewood, B. (2016, February). *Practical Statistical Evaluation of Critical Software.* Paper presented at the 24th Safety-Critical Systems Symposium, Brighton, UK. Retrieved 27 September 2021 from https://scsc.uk/r131/8:1

Littlewood, B. & Verrall, J.L. (1973, April). *A Bayesian reliability growth model for computer software*. Paper presented at the IEEE Symposium on Computer Software Reliability, New York, USA.

Littlewood, B. & Strigini, L. (1993). *Validation of Ultrahigh Dependability for Software-Based Systems*. Communications of the ACM (CACM), 36(11), pp. 69–80. Retrieved 9 November 2021 from https://openaccess.city.ac.uk/id/eprint/1251/1/CACMnov93.pdf

Littlewood, B. (1996). *The impact of diversity upon common mode failures*. Reliability Engineering & System Safety, 51(1), 101–113. Retrieved 22 September 2021 from https://openaccess.city.ac.uk/id/eprint/1630/

Littlewood, B. (1998) *The use of proof in diversity arguments*. IEEE Transactions on Software Engineering, 26(10), 1022–1023. https://www.researchgate.net/publication/3188113_The_Use_of_Proof_in_Diversity_Arguments_in

Littlewood, B., Popov, P. & Strigini, L. (2000). *N-version design versus one good version.* Paper presented at the International Conference on Dependable Systems & Networks, New York, USA. Retrieved 22 September 2021 from https://www.researchgate.net/publication/2585976_N-version_design_Versus_one_Good_Version

Littlewood, B., Popov, P. & Strigini, L. (2001, February). *Design Diversity: an update from research on reliability modelling.* Presented at the 9th Safety-Critical Systems Symposium, Bristol, UK. Retrieved 22 September 2021 from https://openaccess.city.ac.uk/id/eprint/261/2/SCSS2001_v_12.forDistrib.pdf

Littlewood, B. and Rushby, J. (2012). *Reasoning about the Reliability of Diverse Two-Channel Systems in Which One Channel is "Possibly Perfect".* IEEE Transactions on Software Engineering, 38(5), 1178–1194. Retrieved 9 November 2021 from https://openaccess.city.ac.uk/id/eprint/1069/1/1oo2-revised-13apr11.pdf

Mandelbrot, B.B & Hudson, R.L. (2004). *The (Mis)Behavior of Markets: A Fractal View of Risk, Ruin and Reward*, Basic Books.

MCAAI. (1994). *Report on the Accident to Airbus A320-211 Aircraft in Warsaw on 14 September 1993*, Main Commission Aircraft Accident Investigation Warsaw, March 1994. Retrieved 9 November 2021 from http://www.rvs.uni-bielefeld.de/publications/Incidents/DOCS/ComAndRep/Warsaw/warsaw-report.html

Murray, T. and van Oorschot, P. C. (2018). *BP: Formal Proofs, the Fine Print and Side Effects.* Paper presented at 2018 IEEE Cybersecurity Development (SecDev). Retrieved 9 November 2021 from https://people.scs.carleton.ca/~paulv/papers/secdev2018.pdf

O'Halloran, C. (2005). *Ariane 5: Learning from Failure.* Proceedings of the 23rd International System Safety Conference, San Diego, 2005.

RTCA. (1992). *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178B, RTCA, Inc. Also available as EUROCAE Document ED-12B.

RTCA. (2011). *Software Considerations in Airborne Systems and Equipment Certification*, RTCA/DO-178C, RTCA, Inc. Also available as EUROCAE Document ED-12C.

RTCA. (2021). *Terms of Reference, Special Committee (SC) 240, Topics on Software Advancement (Revision 1)*, RTCA Paper No. 083-21/PMC-2139, RTCA, Inc. retrieved 9 November 2021 from https://www.rtca.org/wp-content/uploads/2021/05/SC-240-TOR-Rev-1-Approved-2021-03-18.pdf

Rushby, J. (2009). *Software Verification and System Assurance*. Presented at the 7th IEEE International Conference on Software Engineering and Formal Methods (SEFM), Hanoi, Vietnam, November 2009. In SEFM Proceedings pp. 3–10. Retrieved 22 September 2021 from http://www.csl.sri.com/users/rushby/papers/sefm09.pdf

Rushby, J. (2012). *New Challenges in Certification for Aircraft Software.* Presented at the Ninth ACM International Conference on Embedded Software (EMSOFT), Taipei, Taiwan. Retrieved 22 September 2021 from http://www.csl.sri.com/users/rushby/papers/emsoft11.html

This collation page left blank intentionally.

‘til the Next Zero-Day Comes

Ransomware, Countermeasures, and the Risks They Pose to Safety

Bruce Hunter

Safety and Security Consultant, Sydney, Australia

Abstract

Cyber-attacks on critical infrastructure are not new, but their recent intensity has increased the risk of intended or unintended consequences to safety systems to become a real and present danger. Ransom use of malware attacks have mainly concentrated on business systems, by denying access to essential data, but recent attacks have affected critical infrastructure with consequential shutdown of operation-al technology including safety-related functions. Although ransomware may intentionally cause dangerous failures in the system, pervasive connectivity raises the risks of this happening. This article discusses the precursors to this danger as part of Information Technology and Operational Technology convergence, integration of business and control systems, conflicts arising out of this integration and monetarisation of vulnerability exploitation. Although using Industrial Control System examples are used, safety practitioners may use these to mitigate cybersecurity threats and minimise the impact of attacks on all safety-related systems and their recovery.

1 Out of Gas!

Usually, cyber-attacks on critical infrastructure that includes safety-related systems go relatively unnoticed, unless they have a significant impact on the public. Partly this is due to the sensitivity of companies to bad press and cautiousness in trying to prevent copy-cat attacks. An understanding of the magnitude and nature of attacks, however, can be gained from published, but anonymized, surveys of the industries concerned.

Ransomware attacks in general have become a major threat. Industry surveys have also shown that ransomware is the cyber-threat most likely to affect their organization in the next 12 months (ISACA 2021):

- 21% of businesses in general said that their organizations have experienced ransomware attacks
- 67% said that their organizations will take new precautions in light of the attack on Colonial Pipeline, an American oil distribution system from Houston, Texas
- 78% said critical infrastructure organizations should *not* pay ransom if attacked
- 84% said ransomware attacks would become more prevalent in the second half of 2021

A 2020 survey (Bakuei et al. 2021) shows, for the US alone, that 19 organisations found ransomware in their Industrial Control Systems (ICS).

© Bruce Hunter 2022.

Ransomware risk to Operational Technology (OT) including ICS differs from other forms of attack in the following ways:

- The primary objective of ransomware is monetary gain by locking up assets and information needed by the target organisation
- Impact on critical functions including safety is generally unintended and consequential when resources needed by these functions become unavailable
- Ransomware is only designed to interrupt and may contain untested side effects, which can have reliability and safety implications. In the Colonial Pipeline case, the de-encryption tool did not work properly, and recovery required other methods.

The ransomware attack on the US Colonial Pipeline company in 2021 certainly did get noticed and led to the shutdown of the gas distribution operation for a week affecting the supply of over 40% of the US East Coast supply of automotive and aviation fuel. Like most complex failures, this resulted from a series of events and decisions that made the company's systems more vulnerable, and the response to the attack less optimal.



Figure 1 ~ "Colonial Pipeline", Photograph by Orbital Joe

The key events of this attack were (Charles Carmakal, 2021):

- Hackers gained entry into the networks of Colonial Pipeline on April 29 through a forgotten virtual private network (VPN) account that had poor security (Attributed a cybercrime group using to DarkSide Ransomware-as-a-service (RAAS) toolkit)
- Just before 5am on May 7, Colonial's control room saw a ransom note on their system
- DarkSide ransomware attack locked out the business operation
- As a safety precaution, Colonial shut down its entire pipeline operation, causing critical fuel shortage to Eastern US
- After extensive checks of the OT network concluded that damage was limited to some of the Information Technology (IT) business operation, supply restarted on May 13
- Colonial did pay USD 4.4M in Bitcoin ransom, as a precaution in case recovery was not achievable, although the Federal Bureau of Investigation (FBI) was subsequently able to recover most of this ransom

The following precursors the industry faced in the last few years may have strongly influenced the reaction to this event:

- The US Department of Homeland Security (DHS) Cybersecurity & Infrastructure Security Agency (CISA), and the FBI, advised Pipeline Operators of a spear-phishing and intrusion campaign conducted by state-sponsored Chinese actors that occurred from

December 2011 to 2013, targeting U.S. oil and natural gas pipeline companies (CISA and FBI 2021a).

- A US Congressional report highlights pipeline cybersecurity issues, but US Transportation Security Administration (TSA) "*maintains that voluntary cybersecurity standards have been effective in protecting US pipelines from cyber-attacks*" (Parfomak, 2012)
- The United States Government Accountability Office conducted a review of "*TSA's efforts to assess and enhance pipeline security and cybersecurity*" and issued 10 improvement recommendations in its congressional report, GAO-19-48, but did not highlight industry cybersecurity regulation (GAO 2018).
- In 2019 alone, there were 614 reported pipeline incidents in the United States, resulting in the death of 10 people, injuries to another 35, and about $259 million in damages (Kelso, 2020).
- Despite warnings of pipeline specific ransomware threats (CISA 2020), the industry has been criticised for poor cybersecurity practices and lobbying against stronger regulation
- Industrial Control Systems (ICS) have become an attractive target to Ransomware attackers. The motives are strong (money), the risk is low (RAAS toolkits protect attacker), the target is soft (old ICS hidden vulnerabilities abound), and reward is fairly certain (ICS operations are critical). (Palmer, 2021)
- The Colonial Pipeline company supplies about 45% of east US coast fuel, making it a major risk to US transport operations
- Poor security practices would have meant uncertainty on the reliability of OT network segmentation.

These precursors could have influenced the company's response to the ransomware attack and its outcome, which did result in the shutting down the fuel supply.

Colonial's east coast pipeline is a multi-product, multi-offtake, multi-line, multi-section, and multi-storage operation that has high risks associated with operations outside the safe envelope. Out of an abundance of caution to ensure a dangerous situation did not arise, the company chose to shut down operations to prevent spread to the OT network until the business system was cleared of ransomware and put back into operation (Hoffman and Winston 2021).

The system failure resulted in the following timeline of events with the US fuel supply:

- Crisis caused by the loss of supply resulted in the Federal Motor Carrier Safety Administration (FMCSA) declaring a state of emergency in 18 states to help with the shortages (9 May 2021)
- Colonial Pipeline eventually re-established pipeline operation (13 May)
- FBI and CISA issue alerts on pipeline ransomware threat (11 and 19 May)
- TSA update to Pipeline Security Guidelines issued, replacing criticality guidelines - naturally (TSA 2021a)
- TSA issued Security Directive Pipeline-2021-01 (TSA 2021b) directing a whole range of mandatory report and assessments with significant penalties for non-compliance (27 May)
- United States Department of Justice (DOJ) gives critical infrastructure ransomware attacks equivalent priority to terrorism. (3 June)
- CISA, with the FBI, updated a joint advisory on DarkSide Ransomware: "Best Practices for Preventing Business Disruption from Ransomware Attacks" (CISA and FBI 2021b), originally published 11 May, to supplement previous advice (8July).

- On 28 July, the US president announced further steps to safeguard critical infrastructure (Biden 2021)

Consequences of the Colonial Pipeline attack have certainly focused the minds of the process sector (Moore 2021); but why the concern? Ransomware is a substantial risk to control systems due to:

- Difficulty to recover operations as systems locked by encrypted files
- High motivation of attackers due to monetary rewards or nation-state intent
- Possibility of accidental damage to safety systems as a by-product, due to unaccountability of cybercriminals compared to nation-state threat actors

Ransomware Lessons Learned for safety from this ransomware attack includes:

- Safety and security must be coordinated. This could have affected safety elements of the pipeline if essential operator controls, e.g. human-machine interfaces (HMIs), were affected (CISA 2020), or if the security response to ransomware had impact on safety.
- OT operational aspects may have continued if, independent of the business systems.
- Segregation between OT and IT is necessary but not always assured.

2 Who's in control? OT Cybersecurity

2.1 An Analogy

As I write this article in mid-2021, Sydney is in the middle of another lockdown, trying to control a COVID-19 Delta-strain outbreak despite months when there were effectively no cases in the country. Pandemics and the way we deal with them are very reminiscent of cybersecurity. The following issues are very much characteristic of pandemic and cybersecurity experiences:

- Failure consequences can be far-reaching and include morbidity
- Responses and their success are driven by
    - risk appetite of participants
    - motivation of the threat
- Protection involves:
    - separation of the vulnerable from the threat
    - continually increasing the resilience of the target
    - educating the user on responsibility and actions that may increase risk
- but Thwarted by:
    - constant evolution of the threat tactics and vulnerable entry points
    - detection reliant on known indications and subject to false positives and negatives
    - lack of persistence in assessing and addressing the risks
    - the risk that defence may actually harm the defended
    - unpopularity of preventive measures
    - cognitive bias and complacency about risk

2.2 Convergence of Technology

Control systems have evolved from mechanical, electrical, electronic, and programmable electronic systems but usually remained standalone. OT has converged with hardware and software of IT with the benefits of economies of scale, supportability of common operating systems and skillsets. A less welcome consequence of this was the inheritance of visibility and vulnerability to cyber-attack (ISACA 2016).

Adopting IT platforms and Operating Systems specifically provides:

- Economies of scale;
  - but increased vulnerabilities with complex designs, update frequency and obsolescence issues
- Commonality of software and hardware providing a wealth of functionality;
  - but inheritance of IT vulnerabilities and attack toolkits
  - and visibility to IT threat agents
- Increased connectivity to allow expanded operation and monitoring;
  - but expanded attack vectors or surfaces
- Increased functionality and capability;
  - but reduced reliability and predictability

These issues have made OT systems not only more vulnerable to cyber-attack, but also have impacted reliability and safety.

2.3 ICS Threats

Cyber-threats to systems incorporating Operational Technology (OT) have grown to a point where, in 2020, ICS Advisories were issued by US CISA at an average rate of 5 per week (https://www.cisa.gov/uscert/ics/advisories). Analysis shows that, advisories and alerts have grown to an average of 7 per week in 2021: 55% more than 2020 (Figure 2).

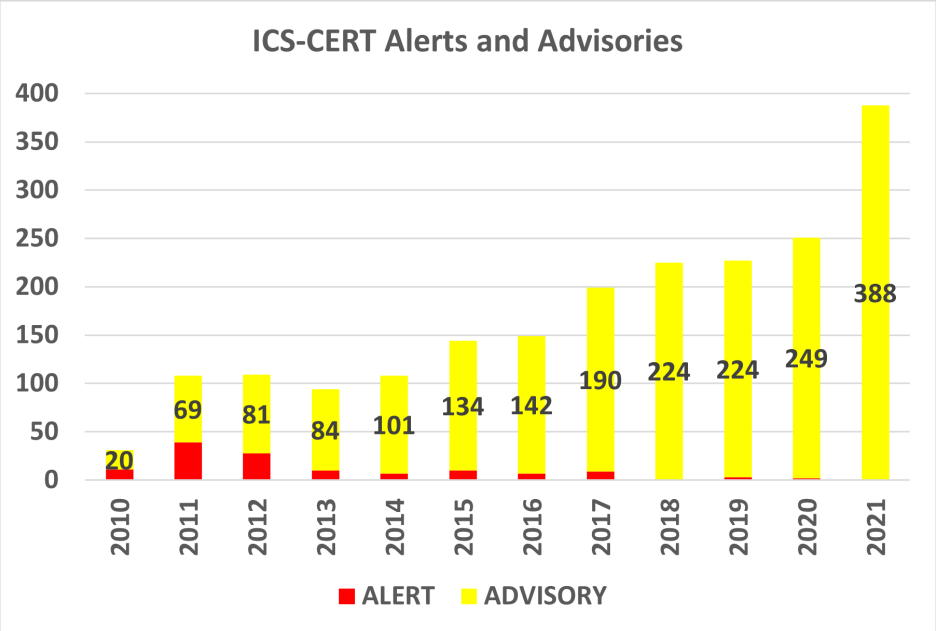


Figure 2 ~ Analysis of CISA ICS Advisories Over Time

Threat actors to ICS have evolved from script buddies and hacktivists to now include state-sponsored specialists and cybercrime using ransomware-as-a-service (RAAS) toolkits.

Significant past attacks noted by CISA of non-ransomware cases with ICS include:

- Joint CISA-FBI Cybersecurity Advisory (AA21-201A) (CISA and FBI 2021a): Gas Pipeline Intrusion Campaign. Attributed to China nation-state cyber actors. Of the targeted entities, 13 were confirmed compromises, 3 were near misses, and 7 had an unknown depth of intrusion. Impact is development by the attacker of capabilities against U.S. pipelines to physically damage pipelines or disrupt pipeline operations
- W32.DistTrack, also known as "Shamoon" (CISA 2021a): An information-stealing malware that also includes a destructive module. Attributed to Iranian nation-state cyber actors. Operational impacts of this attack include loss of intellectual property (IP) and disruption of critical systems.
- ICS Focused Malware Havex Trojan (CISA 2018): An information-stealing malware. Attributed to Russian nation-state cyber actors. Operational impacts may result from information gathered in this attack or malware installed.
- Malware campaign that has compromised numerous industrial control systems (ICSs) environments using a variant of the BlackEnergy malware (CISA 2021b): Installs malware on Internet-connected HMIs. Attributed to Russian nation-state cyber actors. Impact is other actions enabled by compromised HMI.
- Cyber-Attack against Ukrainian Critical Infrastructure including electricity grid (CISA 2021c): Attributed to Russian nation-state cyber actors. Impact was the takeover of electricity grid HMI operation, shutdown communication and backup recovery with loss of large section of the grid.
- CrashOverride Malware (CISA 2021d): Attributed to Russian nation-state cyber actors. Impact is the abuse of functionality in a targeted ICS system's legitimate control system to achieve its intended effect which has included shutdown of electricity grid using standard ICS protocols but could impact all critical infrastructure organizations.
- Safety System Targeted Malware HatMan, also known as TRITON and TRISIS (CISA 2019): Attributed to Russian Government-Owned Laboratory but possibly used by Iranian nation-state cyber-actors. Disrupted the safety-related triple redundant Emergency Shutdown (ESD) of Saudi Arabian petrochemical plant. Safety protection key switch (SIS configuration mode) on one channel was left in the program state allowing the modification of the Triconex system; the other two channels detected anomaly of exploited channel and shutdown.

2.4 Vulnerabilities and Attack Paths

The vulnerability-patch cycle with converged IT-OT, leads threat agents to go to deeper technology layers to achieve exploitation, as examples show in Figure 3.

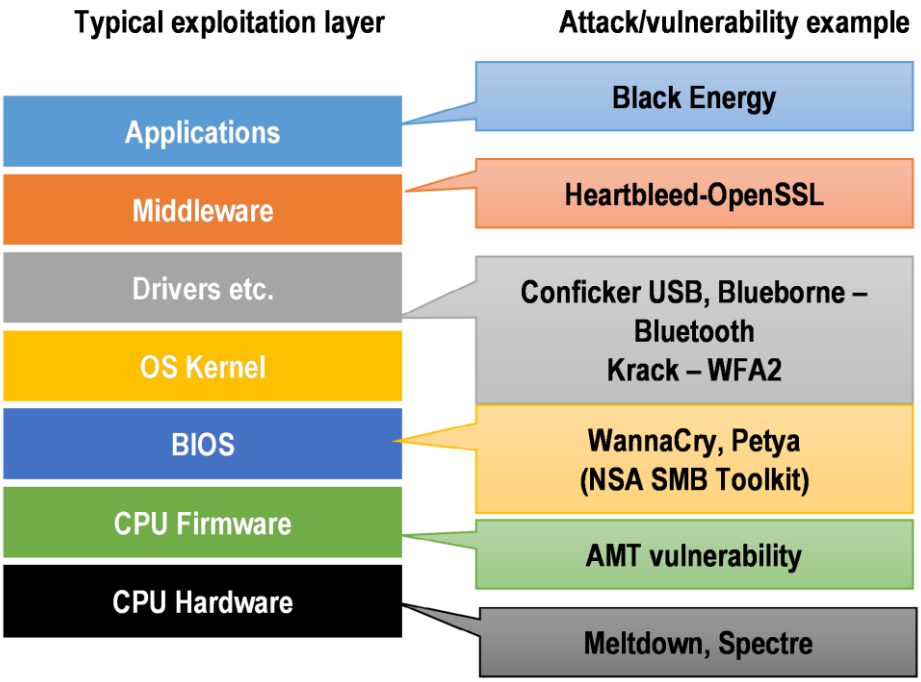


Figure 3 ~ Increasing Depth of Exploitation in Technology Layers

Attack paths and surfaces used by adversaries rely on accessibility to ICS OT networks. Understanding the risk of entry and protective measures is helped by a layered architecture as the example in Figure 4 modified from IEC 62443 (IEC n.d.), which is loosely based on Purdue (Purdue 1989).

This layered approach does provide a Defence-in-depth approach but exhibits some key vulnerable attack vectors as experience in ICS attacks:

- Entry via vulnerabilities in the web and email services of the Enterprise layer (Purdue Layer 4). Despite the use of Demilitarised Zone (DMZ) technology there are still backdoors to attacks via techniques such as email spear phishing
- Entry via Operations Control (Purdue Layer 3); DMZ again subject to backdoor entry
- Entry via Supervisory Control (Purdue Layer 2) and Controllers (Purdue Layer 1) via backdoors and malware on HMI and engineering workstation. Safety-related systems at this level have to be protected against dangerous attacks. Entry points at this level include IT technology of HMI and Engineering Workstations, again typically by malware carried on spear-phishing e-mail or remote maintenance network connection. Even if this is precluded by "air-gaps", backdoors can be gained via portable media (e.g. USB Drives) used to update system software or firmware.
- Manipulation of standard protocols in Field Device connections (Purdue Layer 1 and 0).

Network segmentation (restricting data flow) is a key technique in not only providing defence-in-depth against attack but also providing functional separation or independence for different critical elements of an ICS. Network segmentation does not necessarily mean functional separation.

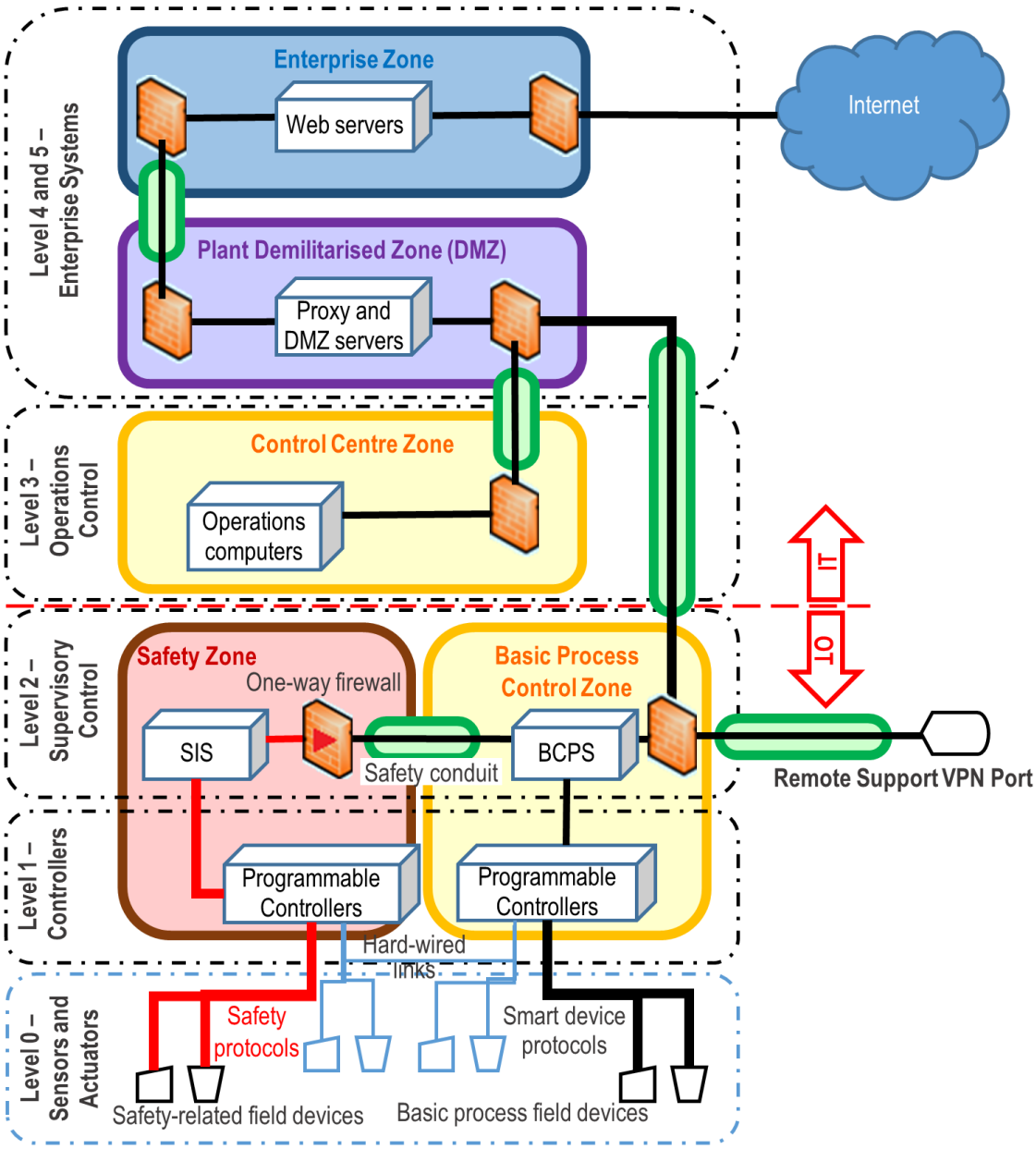


Figure 4~ Simplified Layered ICS Security Architecture (IEC 62443)

Evolution of OT architecture to Industrial Internet of Things (IIoT), Cloud Services and Nested Edge Technology is providing emerging challenges to true defence-in-depth, segmentation, and cybersecurity protection of safety-related systems. Protective segmentation becomes difficult in Software Defined Perimeters due to logical architecture as shown in Figure 5, which is extracted from Annex C of draft IIoT standard ISO/IEC FDIS 30162. Proof of dependable functional separation and interaction is difficult to prove in software-defined networks with non-industrial protocols. Standards are catching up on the model and requirements for IIoT, but issues remain in proving trustworthy segmentation of functional layers. IEC/ISO JTC1/SC41 has highlighted these issues for future Standardization work in a technical report (ISO/IEC 2020).

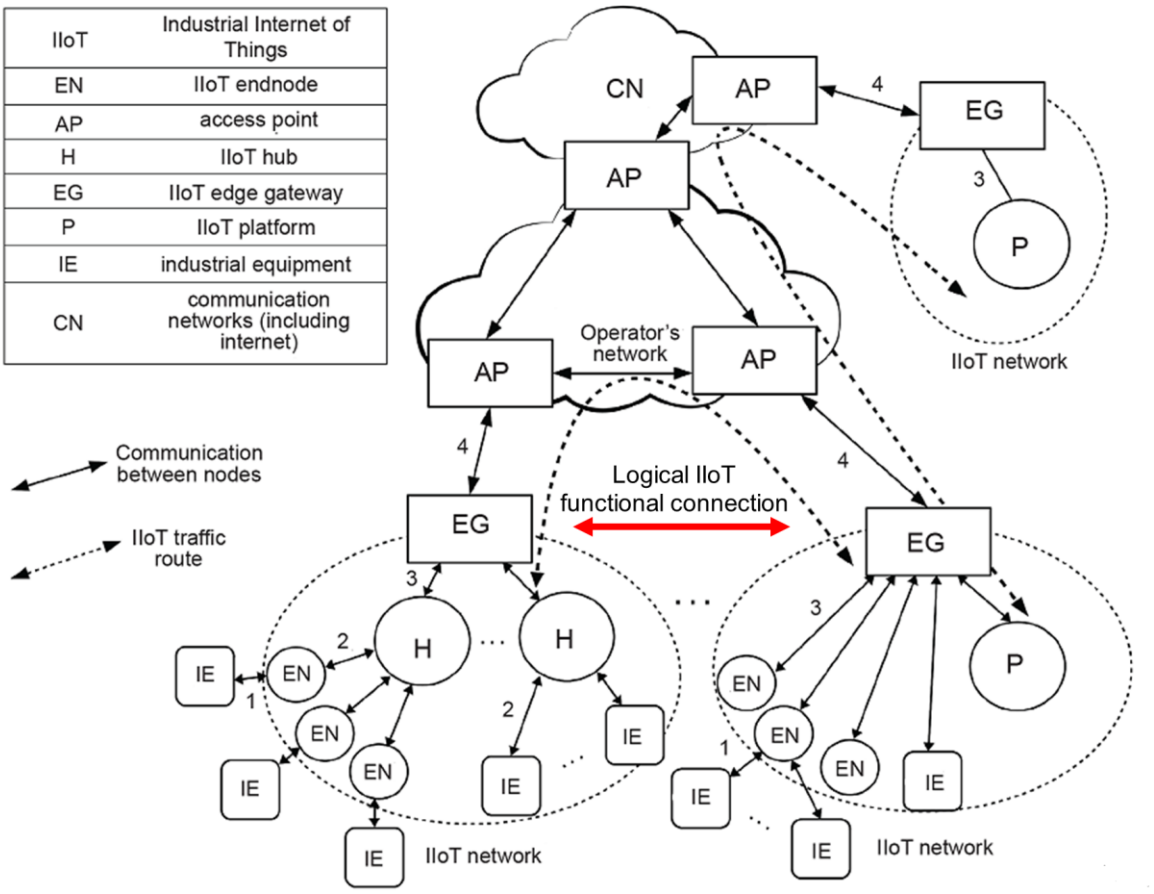


Figure 5 ~ Logical Segmentation of Network in IIoT-Cloud Architecture

2.5 Standards, Directives and Frameworks

2.5.1 General

Evolving threats to and vulnerabilities of ICS have led to a variety of standards, guides, and framework to protect these systems. There is growing guidance on securing OT in ICS (Timpson and Moradian 2018) and standards to support these.

2.5.2 Ransomware Specific Frameworks

The following specific frameworks have been established to minimise the specific risk of ransomware to ICS including pipeline systems:

- CISA FBI joint advisory on DarkSide Ransomware: Best Practices for Preventing Business Disruption from Ransomware Attacks (AA21-131A) (CISA and FBI 2021b)
- US TSA updated Pipeline Security Guidelines (TSA 2021a)
- NIST Framework with security lifecycle guidance on the application of the NIST CSF to ransomware mitigation (NIST 2021).

2.5.3 The ISO/IEC - Standard Security for Industrial Automation and Control Systems

IEC 62443 (IEC n.d) is the "go-to" standard for OT specific cybersecurity, rather than the ISO/IEC 27000 series commonly used for IT, providing requirements to protect against various levels of cyber-attack. It also supports specific protection of safety-related aspects. The published parts include requirements for developers, asset owners, operators and assessors covering the following fundamentals, as follows:

- Security Governance (SG) – having 42 requirements
- Security Development and Integration (SDI) – having 45 requirements
- Risk Management (RM) – having 74 requirements
- Asset Management (AM) – having 128 requirements
- Identification and Authentication Control (IAC) – having 75 requirements
- Use Control (UC) – having 84 requirements
- System Integrity (SI) (includes integrity of safety functions) – having 84 requirements
- Information Confidentiality (IC) – having 47 requirements
- Restricted Data Flow (RDF) – having 30 requirements
- Incident Management (IM) – having 68 requirements
- Resource Availability (RA) – having 29 requirements

2.5.4 NIST Cybersecurity Framework (CSF)

The NIST Framework for Improving Critical Infrastructure Cybersecurity (NIST 2018a), as illustrated in Figure 6, provides a model for assessing maturity of cybersecurity processes.

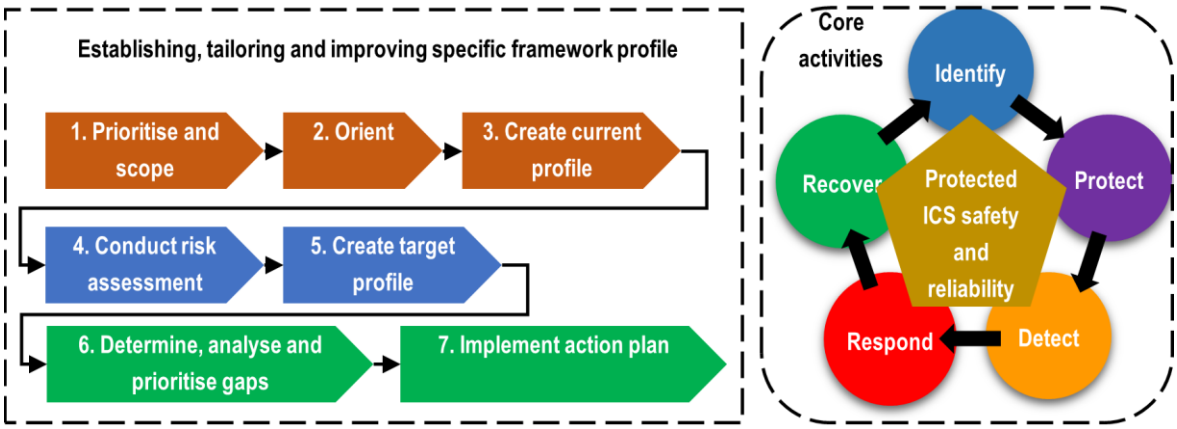


Figure 6 ~ NIST Cybersecurity Framework Model

Safety interaction with the NIST CSF as part of the kill chain is illustrated in Figure 10 of Sub-section 3.3.

Appendix A, Section A.2, proposes a tailoring of the framework core for safety-related systems. CSF profiles have been published for various critical infrastructure sectors.

2.6 Maintaining Effectiveness of ICS Cybersecurity Protection

2.6.1 Patching and Updates

The effectiveness of ICS cybersecurity countermeasures is only as good as the vulnerabilities and threats address at the time of the last update ('til the next zero-day).

Effective cybersecurity protection relies on prompt patches of system vulnerabilities. This has the following challenges with OT systems:

- Reliability of the control system can be degraded by unproven updates. Reliability growth in OT is built up over time and by correction of systematic errors. Patches may cause regression in this growth.
- Security patches may cause unexpected failure due countermeasures or malware detection patterns (Goodin 2010).

Patches and updates to OT systems require validation before they are put on live ICS (Hunter 2013). Figure 7 illustrates the cycle of new vulnerabilities, patching validation and system changes.

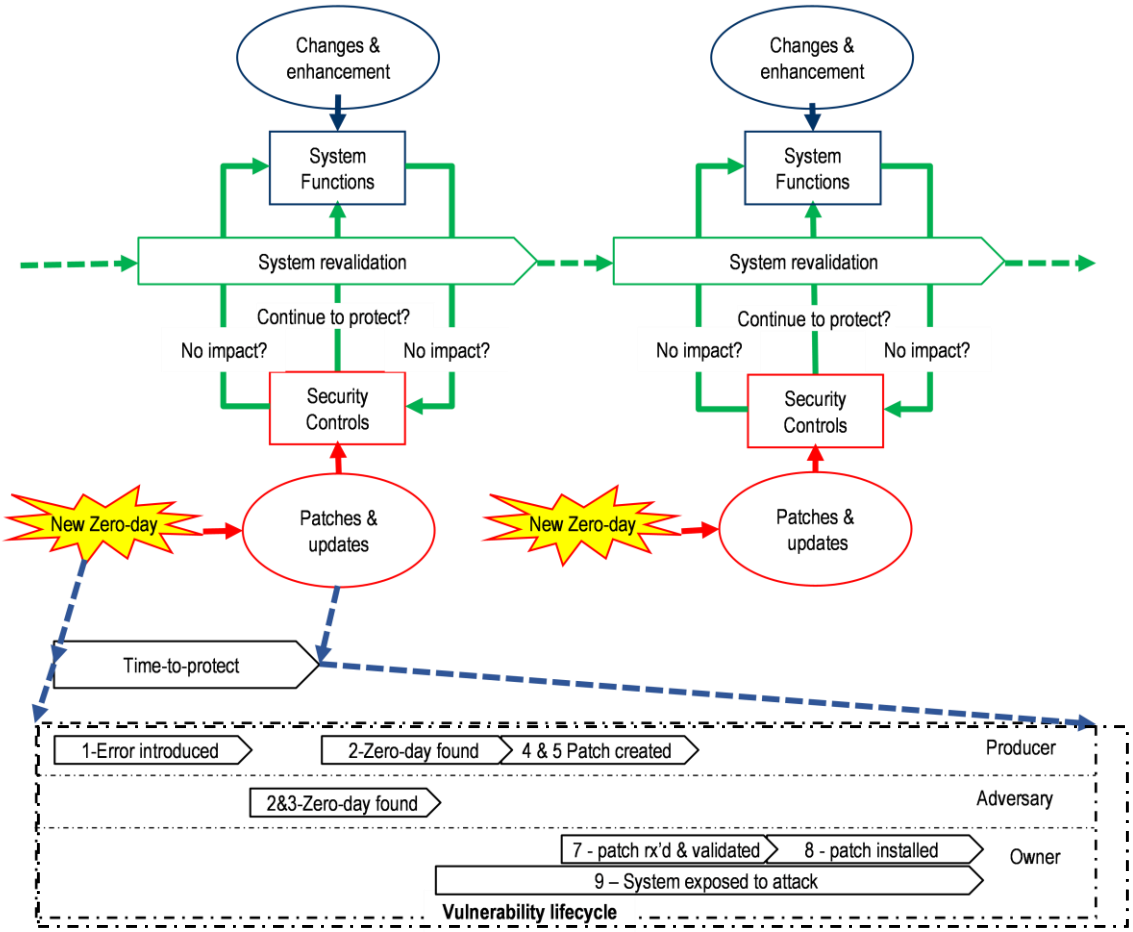


Figure 7 ~ ICS Vulnerability, Update and Validation Cycle

Vulnerability cycle includes:

1. point when software error results in vulnerability
2. point of discovery of new zero-day vulnerability

3. average time to exploitation in the field
4. time to develop a patch to address the vulnerability
5. time to communicate and act on vulnerability
6. time to stage patch and develop precautionary roll-back baseline
7. time to validate on test environment the reliability of the patch in the ICS and compatibility with Safety Functions
8. time to release into the production environment

The aim is to keep exposure time of the system, shown as 9 in Figure 7, as low as possible (hopefully before exploitation is initiated).

Patch Management in the Automation and Control System (IACS) environment; IEC 62443 (Parts 2-3) does specify methods and formats for patches in the development, notification, verification, and validation of security updates for OT systems (IEC n,d,).

2.6.2 CISA ICS Advisories and Alerts

Awareness of emerging OT product vulnerabilities, and timely patching of these, is a key aspect of cybersecurity risk reduction. This requires a balance approach to ensure that system reliability is not compromised. US CISA provides timely advisories for ICS product vulnerabilities. Monitoring of these (https://www.cisa.gov/uscert/ics/advisories) and acting on advice should be a basis of maintain cybersecurity protection of ICS.

2.6.3 Physical Architecture and Network Segmentation

Defence-in-depth, as provided by the long-accepted Purdue Enterprise Reference Architecture (Purdue 1989), is an example of a security architecture in critical industrial control system protected in “zones” and “conduits” (IEC n.d.). An example of this architecture is shown back in Figure 4 of Sub-section 2.4.

Network Segmentation has challenges in achieving separation of functions, let alone independence of these:

- We still need to communicate between IT and OT functions
  - OT functions need access to functionality in IT systems
  - IT systems need to monitor and control OT
- Network segmentation and functional separation can be bypassed by use of portable media (e.g. USB) or external maintenance links
- Firewalls separating network zones can be compromised with concerted efforts; if monitoring is only required across the gap the use of one-way firewalls or data-diodes may help in enforcing isolation.
- Security products are available that scan air-gapped systems without installing software, but these need to be set to not automatically delete files which could be critical to OT.

System Safety Standards, (IEC 2010), require that the boundary of the safety-related system be established and maintained. OT Cyber Security Standards (IEC n.d.) fulfil this by segmentation of safety related function into their own protected zones based on risk assessment, e.g. IEC 62443 Part 3.2.

Separation of functions across these safety boundaries must be maintained throughout the system lifecycle to prevent interference with process critical and safety-related functions (Hunter 2006).

2.7 Connectivity

Convergence of connectivity such as TCP/IP has increased the attack surface and vulnerabilities; however there have been previous attacks on systems with traditional OT architecture with safety and environmental impact, e.g. that on the Maroochy Shire Sewerage System (Smith 2001).

So, we know ICS systems cybersecurity's needs and practices; but what about protecting the safety-related elements?

3 Safe and Secure?

3.1 System Hierarchy

Safety Functions are rarely standalone, and are usually part of a larger system. In IACS, and their aligned systems, Safety Functions are considered, in the system hierarchy, as part of the control system essential functions as shown in Figure 8.

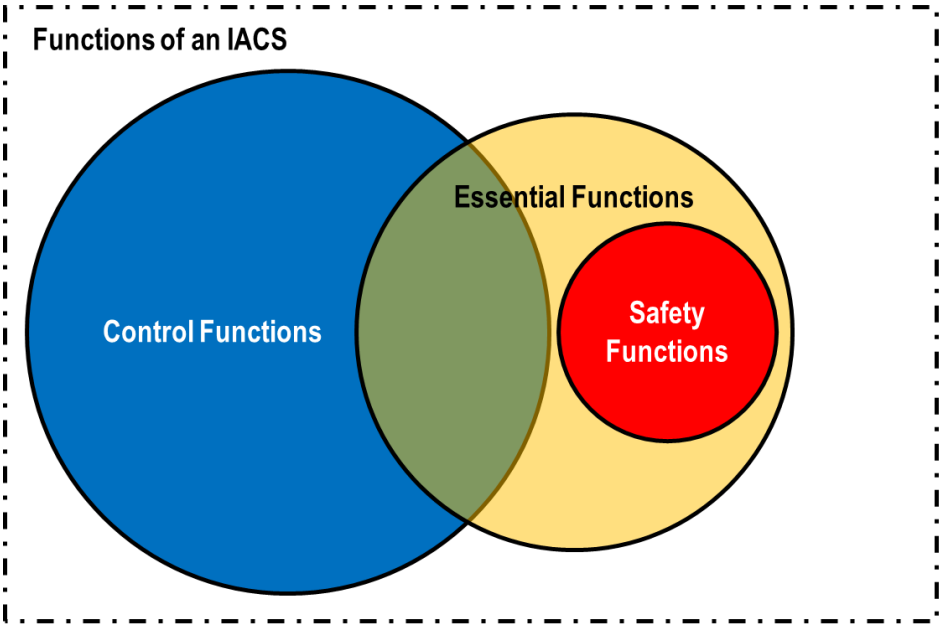


Figure 8 ~ Hierarchy of IACS Functions (IEC 62443)

This hierarchy helps to prioritise system dependability and protection from sources of unintended operation.

The following concepts, techniques and standards support ICS safety and security.

3.2 Boundaries are important after all!

As previously noted, safety standards usually call for the definition of the boundary of the safety-related elements and maintenance of and effective separation across this boundary (Hunter 2006). Setting security and safety boundaries has an impact on independence and interdependence of critical functions.

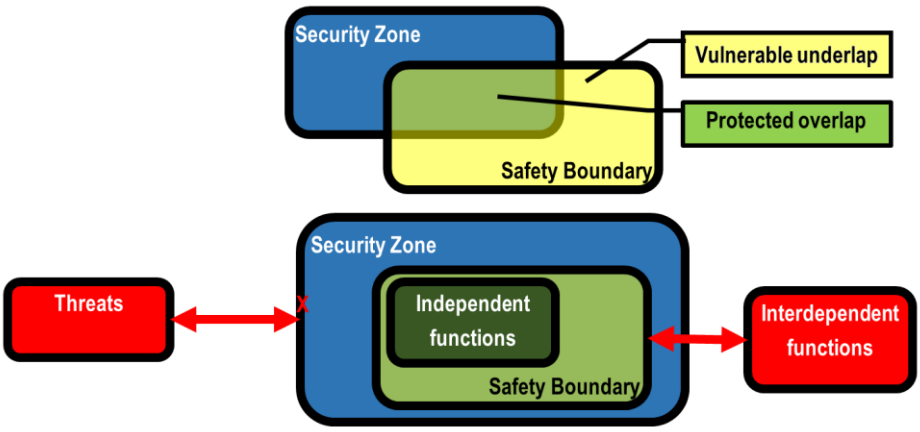


Figure 9 ~ Safety/Security Boundary Considerations

The placement of security perimeters and safety boundaries has a significant impact on the protection and reliability of safety functions:

- Overlap allows security countermeasures to work effectively and reduces the risk of incompatibility with safety functions;
- Underlap increases the risk of back-door attack vectors being exploited; and
- Reliability of safety functions may be affected by cybersecurity countermeasures inadvertently blocking interdependent functions outside the security perimeter.

3.3 Control Systems Kill Chain

Safety practitioners must face the reality that even with the best cybersecurity endeavours the possibility that Advanced Persistent Threat, "APT" will someday succeed in compromising their system. What is left is then reliance on effective safety and security response to this attack. To understand the best response to an attack, you need to know how attacks are staged.

The methods of cyber-attack have become advanced in application, and understanding the kill chain has helped to adapt mitigation to limit not only the likelihood of vulnerability exploitation but also its impact (Assante and Lee 2015). Figure 10 shows the relationship between the NIST Cybersecurity Framework (CSF) (NIST 2018a), Lockheed Martin's generic kill chain, and Safety System incident response/resiliency actions (IEC 2019a), (ISA 2017).

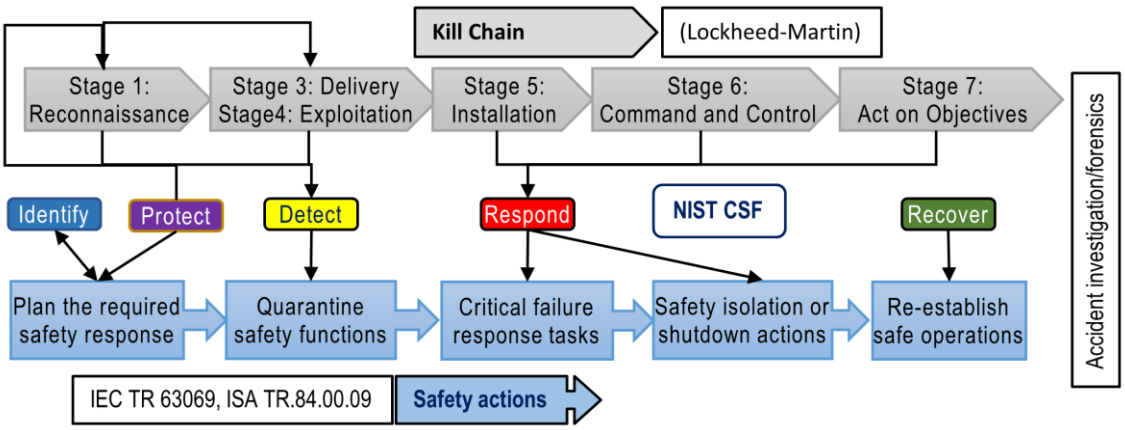


Figure 10 ~ Interaction of Cybersecurity Kill Chain with NIST CSF and Safety

Due to evolving nature of APT and despite best effects in cybersecurity, there is always a residual risk that safety-related components of a system will be compromised by a cyber-attack. Safety functions need to be resilient to this probability and ensure these do not lead to dangerous failures. Safety responses to a recognised attack include:

1. Plan the required safety response
2. Quarantine safety functions
3. Critical failure response tasks
4. Isolation or shutdown actions
5. Re-establish safe operations

Appendix A, Section A.1, provides a generic Framework Profile to address Ransomware risk of Safety Systems in OT.

3.4 Safety and Security Co-engineering Activities

Establishing safe and secure systems requires co-engineering to be undertaken across the system lifecycle (Paul et al. 2016).

IEC has published a Technical Report (IEC 2019a) to assist in protecting safety-related systems from dangerous cyber-attacks in applying safety (IEC 2010) and security (IEC n.d.) standards. It promotes the following guiding principles:

1. protection of safety implementations – to paraphrase, if it isn't secure, it isn't safe
2. protection of security implementations – to paraphrase, safety shouldn't increase security risk
3. compatibility of implementations – to paraphrase, security countermeasures shouldn't be unsafe

A fourth principle is being considered for a new edition with normative clauses:

- Guiding Principle 4: compatibility related to the higher-level system objectives
  - This could satisfy operational objectives, such as availability; and
  - Preclude down-time due to inadvertent shutdowns and fail-safe actions triggered by cyber-attacks.

Currently IEC 61508 is in development to be Edition 3. This includes more detail on handling cybersecurity risk to functional safety. This adds to clauses concerning Hazard and Risk Analysis in Edition 2 (IEC 2010).

ISA also has published a guide (ISA 2017) for the process sector to "address and provide guidance on the safety lifecycle and the cybersecurity lifecycle as they relate to the security of Safety Controls". It covers safety and security activities across the lifecycle from assessment, design, installation, operation, maintenance, modification, and decommissioning.

3.5 Unintended Countermeasure Consequences

Not all threats are external or adversarial. It is possible for inappropriate security countermeasures to impair the very functions they are protecting. This control conflict conundrum, as illustrated in Figure 11, has its counterparts in physical security such as fail-safe and fail-secure conflict of emergency exit doors (Hunter 2009) or physical security of aircraft cockpits (BEA 2016).

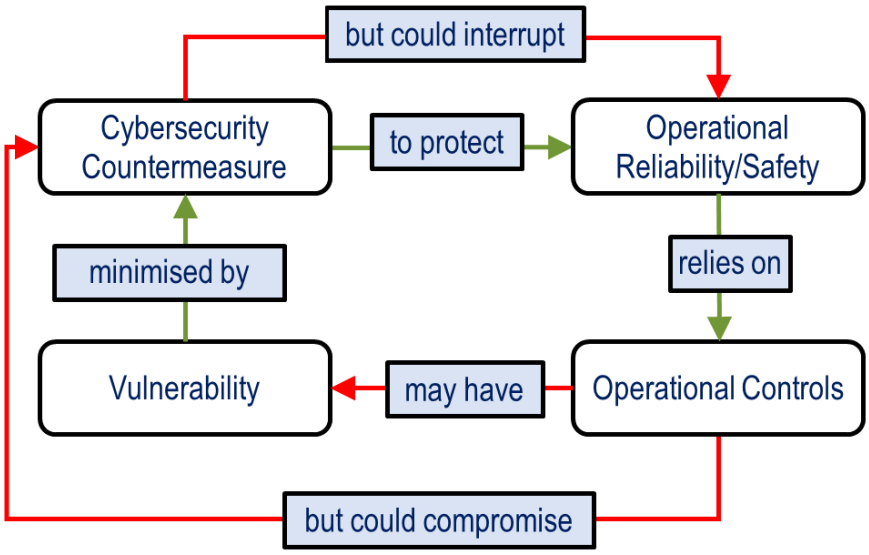


Figure 11 ~ Control Conflict Conundrum

The following safety and security standards do highlight this issue:

- NIST Cybersecurity Publications (NIST 2015), (NIST 2018b) classes these as "accidental or non-adversarial threats"
- Safety of Machinery IEC TR 63074 (IEC 2019b)
  - "*…shall not adversely affect safety integrity (e.g. increase in response time, etc.)*"
  - "*Any security countermeasure shall not adversely affect safety integrity (e.g. increase in response time, etc.)*"
- IEC TR 63069 (IEC 2019a) has a guiding principle: "*compatibility of implementations - Security implementations and safety implementations should not have adverse contradictions*"
- The ISA technical report on Cybersecurity Related to the Functional Safety Lifecycle (ISA 2017) classes these as "*security compromises* [which may] *occur during normal maintenance or other field activities where the cybersecurity compromise is unintentional or accidental*"

Appendix A, Section A.1, proposes common conflicts between cybersecurity and safety.

4 Mind the gap! - Maintaining Safety Independence

4.1 Risks to Functional Separation

Much reliance is placed on air-gaps, but even with no network connection between safety and non-safety related systems these can be bypassed. Loss of functional separation is almost entropic. The risk to this separation includes:

- Bypasses by human action including use of removable media, e.g. USB, that allow malware to infiltrate across the gap by breaking network segmentation and functional separation controls
- Unapproved or forgotten network connections including remote maintenance ports

- Untested changes to network configuration including:
  - message filtering;
  - port assignments;
  - access rules; and
  - authentication.

4.2 Establishing Effective Functional Safety Separation

Effective separation of safety systems is not as easy as it sounds. Strictly, the integrity of the separation mechanisms should have the same integrity as the safety system itself (Hunter 2006). This would mean:

- Network segmentation would require firewalls with a "SIL", i.e. Safety Integrity Level (IEC 2010)
- Human introduced bypasses, such as USB drives across air-gaps, would entail Human Reliability Analysis
- Hard-wired links across safety boundary may provide a level of isolation but require substantiation of meeting requirements for independence of safety, e.g. IEC 61508 Part 3, 7.4.2.9 (IEC 2010).

Pragmatically, separation will not be perfect or lasting; the fall-back position is to safely handle cybersecurity intrusions by a safety-driven incident response.

From a safety perspective, there are key points in the lifecycle to establish and maintain separation from non-safety parts of the system, as proposed in Table 1.

Table 1 ~ Lifecycle Consideration of Safety Boundaries and Functional Separation

IEC 61508 Safety Lifecycle	Functional Safety Separation Activities	Related Cybersecurity Activities
Phase 4. Overall Safety Requirements	Determine safety boundaries	Determine security perimeters and zones
Phase 5. Overall Safety Requirement Allocation	Determine separation requirements	Cybersecurity architecture segmentation requirements
Phase 9. System Safety Requirements Specification	Specify trans-boundary information allowed and prohibited	Establish network firewall rules and air-gap needs
Phase 10. Safety-related Systems Realisation	Establishment of separation measures	Verify security requirements support safety requirements.
Phase 13. Overall Safety Validation	Proof of separation of non-safety systems or influences	Validate effectiveness of security architecture and separation of safety zones.
Phase 14. Overall Operation, Maintenance and Repair	Monitoring for compromised separation	Conduct safe ongoing dependency, visibility, and penetration testing

IEC 61508 Safety Lifecycle	Functional Safety Separation Activities	Related Cybersecurity Activities
Phase 15. Overall Modification and Retrofit	Re-evaluating safety boundaries and separation	Validate system modifications have not reduced the effectiveness of security architecture and separation of safety zones. Prevent unauthenticated SIS configuration changes.

5 Summing up

Safety related systems are a growing target of cyber-attacks, including ransomware. Designers, installers, operators, and maintainers of these system should:

- Understand that safety functions in control systems are subject to increasing cyber threats;
- Ensure safety-related systems are effectively protected by well-maintained cybersecurity measures that are compatible with the system's safety functions;
- Safety and security must be coordinated and cooperative:
  - Safety practitioners, try talking with your cybersecurity counterparts
  - Cybersecurity practitioners, try talking with your system safety counterparts
  - You will be amazed how useful these conversations are…
- Segregation between OT and IT is not assured. Air-gapping is not certain (see RSA 2FA lesson (Greenberg 2021)). Functional separation must be proven and actively maintained for ongoing integrity of safety;
- The challenges of Software Defined Perimeters, such as used in IIoT and Factory 4.0, may increase cyber-attack surface for OT and safety-related systems. They may also decrease true reliability and provable safety integrity.

Disclaimers

Opinions expressed in this paper are based purely on the public references listed. They do not infer culpability or carelessness on any party.

Acknowledgments

The author acknowledges the dedication of safety and security professionals who protect us from dangerous outcome of failures in technology in our lives. The author appreciates the help of Gurinder Pal Singh, Ian H. Gibson, and anonymous peer-reviewers, who have offered advice on this article.

Figure 1 is a photograph taken in 2005 by Orbital Joe, from his album, "Around Baltimore and the State": https://www.flickr.com/photos/orbitaljoe/albums/154745. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 2.0 Generic (CC BY-NC-ND 2.0) International License. To view a copy of this license, visit https://creativecommons.org/licenses/by-nc-nd/2.0/.

References

Assante, M. J., & Lee, R. M. (2015). *The Industrial Control System Cyber Kill Chain.* SANS White Paper, October 2015. Retrieved from SANS Institute: https://sansorg.egnyte.com/dl/HHa9fCekmc Accessed 19th January 2022

Bakuei, M., Flores, R., Remorin, L., & Yarochkin, F. (2021). *2020 Report on Threats Affecting ICS Endpoints*. Retrieved from Trend Micro Research: https://documents.trendmicro.com/assets/white_papers/wp-2020-report-on-threats-affecting-critical-industrial-endpoints.pdf Accessed 18th January 2022

BEA. (2016). *Final Report: Accident on 24 March 2015 at Prads-Haute-Bléone (Alpes-de-Haute-Provence, France) to the Airbus A320-211 registered D-AIPX operated by Germanwings.* Bureau d'Enquêtes et d'Analyses pour la sécurité de l'aviation civile, France. Final Report BEA2015-0125, March 2016. Retrieved from: https://www.bea.aero/uploads/tx_elydbrapports/BEA2015-0125.en-LR.pdf Accessed 20th January 2022

Biden, J. R. (2021). *National Security Memorandum on Improving Cybersecurity for Critical Infrastructure Control Systems.* July 2021. Retrieved from The White House: https://www.whitehouse.gov/briefing-room/statements-releases/2021/07/28/national-security-memorandum-on-improving-cybersecurity-for-critical-infrastructure-control-systems/ Accessed 30th July 2021

Carmakal C. (2021). Prepared Statement to the US House Committee on Homeland Security, June 9, 2021. Retrieved from https://homeland.house.gov/imo/media/doc/2021-06-09-HRG-Testimony%20Carmakal.pdf Accessed 18th January 2022

CISA. (2018). *ICS Focused Malware.* US Cybersecurity and Infrastructure Security Agency, ICS Alert (ICS-ALERT-14-176-02A), updated August 2018. Retrieved from: https://www.cisa.gov/uscert/ics/alerts/ICS-ALERT-14-176-02A Accessed 19th January 2022

CISA. (2019). *HatMan—Safety System Targeted Malware.* US Cybersecurity and Infrastructure Security Agency, Malware Analysis Report MAR-17-352-01, updated February 2019. Retrieved from: https://www.cisa.gov/uscert/sites/default/files/documents/MAR-17-352-01%20HatMan%20-%20Safety%20System%20Targeted%20Malware%20%28Update%20B%29.pdf Accessed 19th January 2022

CISA. (2020). *Ransomware Impacting Pipeline Operations*. US Cybersecurity and Infrastructure Security Agency, Alert (AA20-049A), updated October 2021. Retrieved from https://www.cisa.gov/uscert/ncas/alerts/aa20-049a Accessed 18th January 2022

CISA. (2021a). *Shamoon/DistTrack Malware.* US Cybersecurity and Infrastructure Security Agency, ICS Joint Security Awareness Report (JSAR-12-241-01B), updated July 2021. Retrieved from: https://www.cisa.gov/uscert/ics/jsar/JSAR-12-241-01B Accessed 19th January 2022

CISA. (2021b). *Ongoing Sophisticated Malware Campaign Compromising ICS.* US Cybersecurity and Infrastructure Security Agency, ICS Alert (ICS-ALERT-14-281-01E), updated July 2021. Retrieved from: https://www.cisa.gov/uscert/ics/alerts/ICS-ALERT-14-281-01B Accessed 19th January 2022

CISA. (2021c). *Cyber-Attack Against Ukrainian Critical Infrastructure*. US Cybersecurity and Infrastructure Security Agency, ICS Alert (IR-ALERT-H-16-056-01), updated July 2021. Retrieved from: https://www.cisa.gov/uscert/ics/alerts/IR-ALERT-H-16-056-01 Accessed 19th January 2022

CISA. (2021d). *CrashOverride Malware.* US Cybersecurity and Infrastructure Security Agency, Alert (TA17-163A), updated July 2021. Retrieved from: https://www.cisa.gov/uscert/ncas/alerts/TA17-163A Accessed 19th January 2022

CISA & FBI. (2021a). *Joint Cybersecurity Advisory: Chinese Gas Pipeline Intrusion Campaign, 2011 to 2013.* US Federal Bureau of Investigation and Cybersecurity and Infrastructure Security Agency, Alert (AA21-201A), updated July 2021. Retrieved from https://www.cisa.gov/uscert/ncas/alerts/aa21-201a Accessed 18th January 2022

CISA & FBI. (2021b*). DarkSide Ransomware: Best Practices for Preventing Business Disruption from Ransomware Attacks*. US Federal Bureau of Investigation and Cybersecurity and Infrastructure Security Agency, Alert (AA21-131A), updated July 2021, Retrieved from: https://us-cert.cisa.gov/ncas/alerts/aa21-131a Accessed 30th July 2021

GAO. (2018). *Critical Infrastructure Protection: Actions Needed to Address Significant Weaknesses in TSA's Pipeline Security Program Management*. United States Government Accountability Office. GAO-19-48, December 2018. Retrieved from https://www.gao.gov/assets/gao-19-48.pdf Accessed 18th January 2022

Goodin, D. (2010). *McAfee false positive bricks enterprise PCs worldwide.* April 2010. Retrieved from: https://www.theregister.com/2010/04/21/mcafee_false_positive/ Accessed 30th July 2021

Greenberg, A. (2021). *The Full Story of the Stunning RSA Hack Can Finally Be Told*. May 2021. Retrieved from WIRED: https://www.wired.com/story/the-full-story-of-the-stunning-rsa-hack-can-finally-be-told/ Accessed 30th July 2021

Hoffman, M & Winston, T. (2021). *Recommendations Following the Colonial Pipeline Cyber Attack.* Retrieved from Dragos website: https://www.dragos.com/blog/industry-news/recommendations-following-the-colonial-pipeline-cyber-attack/ Accessed18th January 2022

Hunter, B. (2006). *Assuring Separation of Safety and Non-safety Related Systems*. 11th Australian Workshop on Safety Related Programmable Systems (SCS'06), Melbourne: Conferences in Research and Practice in Information Technology, Vol. 69. Retrieved from: https://dl.acm.org/doi/pdf/10.5555/1274236.1274243 Accessed 19th January 2022

Hunter, B. (2009). *Integrating safety and security into the system lifecycle.* In Improving Systems and Software Engineering Conference (ISSEC), Canberra, Australia, p. 147. August 2009

Hunter, B. (2013). *Verifying Security-Control Requirements and Validating their Effectiveness.* INCOSE Insight, Volume 16 Issue 2, pp 45-48. June 2015

IEC (n.d). *Industrial communication networks - Network and system security*, IEC 62443, all parts separately dated. International Electrotechnical Commission, Geneva

IEC. (2010). *Functional safety of electrical/electronic/programmable electronic safety-related systems.* IEC 61508. International Electrotechnical Commission, Geneva.

IEC. (2019a). *Technical Report: Industrial-process measurement, control and automation – Framework for functional safety and security.* IEC TR 63069:2019, International Electrotechnical Commission, Geneva

IEC. (2019b). *Technical Report: Safety of machinery - Security aspects related to functional safety of safety-related control systems.* IEC TR 63074:2019, International Electrotechnical Commission, Geneva

ISA. (2017). Cybersecurity Related to the Functional Safety Lifecycle. ISA-TR84.00.09-2017, International Society for Automation, Research Triangle

ISACA. (2016). *The Merging of Cybersecurity and Operational Technology*. Information Systems Audit and Control Association (ISACA) White Paper.

ISACA. (2021). *ISACA Survey: IT Security and Risk Experts Share Ransomware Insights in the Aftermath of the Colonial Pipeline Attack*. May 24, 2021. Retrieved from ISACA: https://www.isaca.org/why-isaca/about-us/newsroom/press-releases/2021/it-security-and-risk-experts-share-ransomware-insights-in-the-colonial-pipeline-attack Accessed 30th July 2021

ISO/IEC. (2020). *Technical Report: Internet of things (IoT) — Industrial IoT*. ISO/IEC TR 30166:2020, International Organization for Standardization and International Electrotechnical Commission, Geneva

Kelso, M. (2020). *Pipelines Continue to Catch Fire and Explode*. Retrieved from: https://www.fractracker.org/2020/02/pipelines-continue-to-catch-fire-and-explode/ Accessed 30th July 2021

Moore, S. (2021). *Gartner Predicts By 2025 Cyber Attackers Will Have Weaponized Operational Technology Environments to Successfully Harm or Kill Humans.* Retrieved from: https://www.gartner.com/en/newsroom/press-releases/2021-07-21-gartner-predicts-by-2025-cyber-attackers-will-have-we Accessed 30th July 2021

NIST. (2015). Guide to Industrial Control Systems (ICS) Security. SP 800-82 Rev. 2, May 2015. Retrieved from National Institute of Standards and Technology: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-82r2.pdf Accessed 19th January 2022

NIST. (2018a). *Framework for Improving Critical Infrastructure Cybersecurity.* April 2018. Retrieved from National Institute of Standards and Technology: https://nvlpubs.nist.gov/nistpubs/cswp/nist.cswp.04162018.pdf Accessed 19th January 2022

NIST. (2018b). *Systems Security Engineering: Considerations for a Multidisciplinary Approach in the Engineering of Trustworthy Secure Systems.* SP 800-160 Vol 1, updated March 2018. Retrieved from National Institute of Standards and Technology: https://nvlpubs.nist.gov/nistpubs/SpecialPublications/NIST.SP.800-160v1.pdf Accessed 19th January 2022

NIST. (2021). *Cybersecurity Framework Profile for Ransomware Risk Management.* NISTIR 8374 (Draft, September 2021). Retrieved from National Institute of Standards and Technology: https://nvlpubs.nist.gov/nistpubs/ir/2021/NIST.IR.8374-draft.pdf Accessed 19th January 2022

Palmer, D. (2021). *Ransomware gangs are taking aim at 'soft target' industrial control systems.* July 2021. Retrieved from ZDnet Security website: https://www.zdnet.com/article/ransomware-gangs-are-taking-aim-at-soft-target-industrial-control-systems/ Accessed 30th July 2021

Parfomak, P.W. (2012). *Pipeline Cybersecurity: Federal Policy*. Congressional Research Service Report for Congress, R42660, August 16, 2012. Retrieved from https://sgp.fas.org/crs/homesec/R42660.pdf Accessed 18th January 2022

Paul, S., Rioux, L., Gailliard, G., & Wiander, T. (2016). *Recommendations for Security and Safety Co-engineering.* The Information Technology for European Advancement (ITEA 2) project "MERgE"

Purdue Research Foundation (T. J. Williams). (1989). *A Reference Model For Computer Integrated Manufacturing (CIM).* December 1989. Instrument Society of America, Research Triangle

Smith, T. (2001). *Hacker jailed for revenge sewage attacks*. October 2001. Retrieved from: https://www.theregister.com/2001/10/31/hacker_jailed_for_revenge_sewage/ Accessed 19th January 2022. For more detail of the referenced court case, see also https://www.queenslandjudgments.com.au/caselaw/qca/2002/164 Accessed 30th July 2021

Timpson, D., & Moradian, E. (2018). *A Methodology to Enhance Industrial Control System Security.* Proceedings of the 22nd International Conference on Knowledge-Based and Intelligent Information & Engineering Systems, Belgrade, Serbia (pp. 2117-2126). Elselvier & sciencedirect.com

TSA. (2021a). *Pipeline Security Guidelines*. Retrieved from US Transport Security Administration: https://www.tsa.gov/sites/default/files/pipeline_security_guidelines.pdf Accessed 30th July 2021

TSA. (2021b). *Enhancing Pipeline Cybersecurity.* Security Directive Pipeline-2021-01. May 2021. Retrieved from: https://www.powermag.com/wp-content/uploads/2021/05/sd-pipeline-202-1-01-tsa.pdf Accessed 18th January 2022

Appendix A. Supplemental Material

A.1 Countermeasure Safety Issues

Inappropriately configured or utilised cybersecurity countermeasure can have negative impacts on safety as summarised by Table 2.

Table 2 ~ Cybersecurity and Safety Conflict Issues

Countermeasure/ Activity	Risk to Safety Function	Possible Mitigations
Penetration testing	Could disrupt safety system or cause uncommanded dangerous operation	Have safe and proven penetration testing tools – isolate dangerous operation
Patching incompatibility	Could disrupt safety system or cause uncommanded dangerous operation	Verify patch in pre-production platform
AV false positive	Could stop safety functions	Verify anti-virus update in pre-production platform
PKI certificates expiry	Could stop safety functions	Ensure safety functions not compromised by authentication failures
Firewall policy changes	Could stop safety communications or dependencies	Validate and control firewall policy especially in safety conduits
Password expiry policy	Enforced timeout on passwords may stop operator from enacting safety-related commands	Ensure effective access control management that maintains safety controls
Intrusion Protection System (IPS), Endpoint Detection and Response (EDR & XDR)	Could stop safety functions if these are visibly confused with cyber-attack	Isolate IPS, Endpoint Detection and Response (EDR) and Extended Detection and Response (XDR) systems from critical safety zones
Networked or portable malware scanning and removal tools	False positives could remove OT critical files and applications	Validate tool in OT sandbox or test environment. Set to delete files manually.
Online control system internet presence discovery tools, e.g. Shodan, Google dorks	Could expose safety-system addressing to threat actors.	Use offline tools instead, e.g. NMAP, ZENMAP

A.2 Generic Framework Profile for Safety Systems

The NIST Cybersecurity Framework (NIST 2018a) provides a complete security cycle perspective for aspects of identification, protection, detection, response, and recovery to maintain the cybersecurity of systems. NIST has published a draft CSF profile for ransomware of critical infrastructure (NIST 2021) which provides a tailoring to the profile application to Ransomware (along with other Malware) and System Safety mitigation in Operational Technology (OT). This can be useful to assess countermeasures applied to these systems and threats they address. This framework allows specific profiles to be tailored to a system and its cybersecurity risks. Table 3 proposes safety considerations for CSF cycles and categories.

Table 3 ~ Framework Profile Considerations for Safety

NIST CSF Cycle	Category	Functional Safety Considerations
Identify (ID)	Asset Management (ID.AM)	Safety assets and configuration identified.
	Business Environment (ID.BE)	Organizational safety roles and responsibilities assigned
	Governance (ID.GV)	Safety regulatory requirements established.
	Risk Assessment (ID.RA)	Safety included in hierarchy of risk assessment.
	Supply Chain Risk Management (ID.SC)	Response and recovery plans are tested to include safety responses.
Protect (PR)	Identity Management, Authentication and Access Control (PR.AC)	Network segmentation and air-gapping of safety related systems established and maintained
	Awareness and Training (PR.AT)	Awareness and training include interaction between safety and security responsibilities.
	Data Security (PR.DS)	Penetration and visibility testing is conducted with safety in mind
	Information Protection Processes and Procedures (PR.IP)	Response and recovery plans regularly tested with safety and cybersecurity actions Vulnerability management and patching includes impact and integrity of safety systems
	Maintenance (PR.MA)	Changes to safety system are approved, logged, and performed in a manner that prevents unauthorized access
	Protective Technology (PR.PT)	Fail-safe and fail-secure mechanisms are established to deal with dangerous cyber events.

NIST CSF Cycle	Category	Functional Safety Considerations
Detect (DE)	Anomalies and Events (DE.AE)	System safety is included in determination of event impact.
	Security Continuous Monitoring (DE.CM)	Isolated safety zones are monitored for evidence of intrusion.
	Detection Processes (DE.DP)	Cybersecurity intrusion detection and protection systems do not compromise safety functions, e.g. false positives
Respond (RS)	Response Planning (RS.RP)	Joint safety and security response plans are executed to protect safety during attack.
	Communications (RS.CO)	Information is shared between safety and security personnel and coordination is practiced to minimise dangerous failures.
	Analysis (RS.AN)	Safety and security personnel cooperate on forensics and accident analysis that results from event.
	Mitigation (RS.MI)	Safety and security personnel coordinate mitigation of the event including precautionary shutdowns
	Improvements (RS.IM)	Safety and security personnel cooperate on lessons learned from the event.
Recover (RC)	Recovery Planning (RC.RP	Restoration plans are coordinated between safety and security.
	Improvements (RC.IM)	Lessons learned are included into response plans and processes. Gaps in safety and security protections are acted on.
	Communications (RC.CO)	Restoration plans are coordinated between safety and security including return to safe-state operations.

This collation page left blank intentionally.

THE SAFETY-CRITICAL SYSTEMS CLUB

Safety-Critical Systems eJournal
Invitation to Submit a Paper

Safety-Critical Systems eJournal ~ Call for Papers

Please register as a reader of this new SCSC journal at https://scsc.uk/journal

Authors of papers published in the journal will be offered a year's free membership of the Safety-Critical Systems Club!



www.scsc.uk

In its thirtieth anniversary year, the Safety-Critical Systems Club is introducing a new on-line journal. Its mission is to publish high-quality, peer-reviewed articles on the subject of systems safety. Types of paper will include, but are not limited to:

Technical Articles: Written by practitioners and describing practical safety assurance techniques and their industrial applications.

Integration Studies: Written by practitioners reporting upon successful (or otherwise) synergies achieved in practice with other assurance domains, e.g. security, environment, and resilience.

Position Papers: Written by, or on behalf of, Regulators, Standardisation Organisations, or other official bodies, setting out their position on a topic, e.g. the interpretation of a particular standard or regulation.

Review Articles: Papers highlighting recent developments and trends in some aspect of safety-critical systems or of their use in a particular industrial sector.

Historical Articles: Papers describing the development of safety assurance in an industrial sector; how we got to where we are today.

Perspectives: The authors' personal opinions on a subject, e.g. whether to use statistical methods in particular scenarios.

Reports: The lessons learned from incidents or the outcomes of trials with a description of scenarios or methods and a discussion of the results obtained.

Working Group Outputs: Written by Safety-Critical Systems Club Working Groups to include discussions, underpinning theory, or guidelines.

If you would like to contribute a paper, and/or offer yourself as a reviewer, please register on the journal website, https://scsc.uk/journal

A document template containing Author Guidelines is available on the website.

The remit of the Safety-Critical Systems Club is to support the Safety Engineering Community through the dissemination of technical and educational information. To ensure this information is available to the community, unless otherwise agreed, it will be published under the Creative Commons Attribution 4.0 License (CC BY 4.0), which allows it to be used and reused providing appropriate credit it given. To view a copy of this license, visit https://creativecommons.org/licenses/by/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.