

Software Criticality Analysis of COTS/SOUP

Peter Bishop^{1,2}, Robin Bloomfield^{1,2}, Tim Clement¹, Sofia Guerra¹

¹Adelard

²CSR, City University

Drysdale Building, Northampton Square

London EC1V 0HB, UK

[pgb,reb,tpc,aslg}@adelard.co.uk](mailto:{pgb,reb,tpc,aslg}@adelard.co.uk)

Abstract. This paper describes the Software Criticality Analysis (SCA) approach that was developed to support the justification of commercial off-the-shelf software (COTS) used in a safety-related system. The primary objective of SCA is to assess the importance to safety of the software components within the COTS and to show there is segregation between software components with different safety importance. The approach taken was a combination of Hazops based on design documents and on a detailed analysis of the actual code (100kloc). Considerable effort was spent on validation and ensuring the conservative nature of the results. The results from reverse engineering from the code showed that results based only on architecture and design documents would have been misleading.

1 Introduction

This paper describes the Software Criticality Analysis (SCA) approach that was developed to support the justification of a commercial off-the-shelf software (COTS) that is used in a safety-related C&I system. The primary objective of the criticality analysis is to assess the importance to safety of the software components within the COTS and to show there is segregation between software components with different safety importance. There are both economic and safety assurance motivations for using SCA on pre-developed software products or COTS in safety-related applications. The software products are typically not implemented to comply with the relevant industry safety standards, and may therefore require additional documentation, testing, field experience studies and static analysis of the code to justify its use in a safety context. There can therefore be considerable benefits in identifying or limiting the safety criticality of software components, as this can reduce the costs of justifying the overall system.

This work can be seen as an extension of the IEC 61508 concept of SIL that applies to safety functions to provide a safety indication of software component criticality. It is also an extension of the various work reported on software failure modes and effects analysis and Hazops. The distinctive features of our analysis include the application of such techniques to COTS and the validation of the assumptions underlying the safety analysis. This validation had a large impact on the results. The overall approach is very similar to the one advocated in a study we undertook for the UK HSE [2] for justifying “software of uncertain pedigree” (SOUP) and provides more detail to support this overall COTS/SOUP framework.

2 Identifying the software concerned

A pre-requisite for the software criticality analysis (SCA) is the identification of the main software components and their interconnections. A number of different representations could be

used for this purpose: design documents and expert opinion or source code analysis. In this particular project, both methods of identification were used, as described below.

2.1 Design Documents and Expert Opinion

In the first stage of subsystem identification, the design documentation was analysed to establish the major software components such as:

- operating system kernel
- device drivers
- support services (communications, remote program loading)
- application level scheduler for running control applications
- interpreter and “function blocks” for executing control application function

The overall software architecture was re-engineered with the aid of experts used to maintaining and developing the system and from an examination of the system documentation and code. This described the major software partitions and provided a mapping from the software files to the architecture component. (Note the structure of the source files did not readily map into the architecture partitions). At a more detailed level, the design experts provided names of the top-level software components within the system and also identified which elements were to be used in this particular application. The unused software would not represent a threat provided it was really not used or activated.

2.2 Analysis of the Software Structure

There is a risk that design and maintenance documents and expert opinion may not represent the actual software implementation. To provide a rigorous traceable link between the criticality analysis and the software, the structure was reverse engineered from the code. This established the actual architecture and identified the “top” procedures in the software call trees and all the software that is influenced by these as the basis for analysis. There were difficulties in relying on the maintenance documentation or the design documentation that was not written from an assurance viewpoint. As the project continued and the code-based analysis was shown to be feasible, greater reliance was placed on using the code analysis to provide the structure of the software, with the experts and the function call names providing the semantics of the code.

The 100k lines of C code were analysed to identify the top-level call-trees and the supporting subroutines. This analysis was implemented using the CodeSurfer [5] tool, which parses the C source code and generates a description of all the dependencies between the C elements (data and subroutines).

The system was then represented as a collection of top-level functions, covering the entry points into the system and the interfaces to major sub-components of the system. These were identified by a combination of analysis of the call graph extracted from the C components of the code using CodeSurfer [5], and domain knowledge.

Some of the top-level procedures could be immediately assigned the highest criticality by inspection, as they were essential to the functioning of the entire system. These would typically include low level interrupt service routines that are triggered by hardware events and key scheduling functions. The remaining top-level functions were candidates for a Hazops to determine their criticality.

The analysis showed that the conceptually independent components at the design level could actually share code (such as utility functions). In such cases, the shared code must have the same classification as the most critical call tree it supports.

The main limitation of such a control flow analysis is that it assumes that there are no “covert channels” (e.g. via data shared between trees) that can affect an apparently independent software call tree. A later SCA validation (described in Section 5) addressed this aspect and this resulted in some changes to the safety classification of the software components.

The approach used to assess and classify the safety impact of the identified software components is described in the next section.

3 Assessing the Safety Impact of Failure - HAZOPS

One of the key techniques for assessing the impact of failure is the software equivalent of a hazard and operability study (Hazops) [4]. The Hazops was carried out broadly in accordance with the guidance in Interim Defence Standard 00-58 [3]. The analysis was initially directed towards functions that we expected to have a criticality of less than 4 (the assumed criticality of the system as a whole) in order to derive and justify that lower criticality.

The Hazops process also developed recommendations such as:

- The need for further study of system behaviour: for example the results were not readily apparent to the Hazops team.
- The need for further analysis to underwrite the assumptions in the Hazops. Often these concerned the incredibility of certain failure modes.
- The need to consider additions to design.

After the assessment of failure impact the software components were sentenced as discussed in the following section.

4 Ranking software components

The basis of the classification scheme is that software and hardware in the system have different types of requirements. Broadly speaking these arise from the need to show correctness for the safety system functions, the correctness of the internal fault handling features and lack of impact when the fault handling features fail.

These requirements will depend on the frequency of the challenge, which in turn depends on the overall system architecture and design. In general, because of the redundant architecture and the periodic manual testing, the fault handling features will have lower integrity requirements. The most important events are likely to be:

- Failure of common protection functions (where correct operation is important).
- Interference failures of supporting functions (like fault handling and system maintenance) that affect the integrity of the protection functions.

In this section we describe the process of ranking the components previously identified (Section 3) according to risk assessed on likely failure rate and consequence. This ranking builds on the Hazop analysis described above. The results of the criticality analysis are summarised in a Safety Criticality Index (SCI) that defines the importance to safety. Each failure was attributed a value based on its (system-level) consequence, the estimated frequency of demand, and the degree of mitigation of the consequence. The SCI was then derived from this and a sentence encapsulates the results of this assessment. It should be noted that the same software item might be in several classes at once and in general the highest classification would take precedence, which is a conservative assumption.

4.1 Defining the SCI

The SCI is an indication of the importance to safety of a software element. It does not indicate the importance of that software to the qualification process. It may be that very strong and easily defended arguments exist to show that a piece of software meets very stringent requirements and other, more difficult, arguments are needed for lower criticalities.

It is useful to distinguish different types of risk from the software:

1. it defeats a safety function
2. it spuriously initiates a safety function
3. it defeats the handling of a safety function after its initiation

The SCI calculation evaluated these separately and for each type of risk considered:

- the worst case consequence on the computer-based safety function
- the frequency with respect to demands placed on the safety function
- the mitigation

The basis for the SCI calculation is a semi-quantitative calculation of the risk from the software. These figures are expressed relative to each of the types of risk of the safety functions considered and deal in approximate orders of magnitude. The Software Criticality Index is defined as the logarithm of the risk. The details are shown in Appendix A.

In order to systematically sentence the SCI of the components, rules are needed to relate the consequence of the failure mode to the loss of the system functions (as SCI is relative to this), and to the length of time exposure of the system to the risk.

For example, for a process protection application, rules are needed for the severity of:

- Loss of a controlled process shutdown function.
- The consequences of spuriously initiating a shutdown.

There are a number of other factors that might change the consequence:

- Failure only affects a single channel of a redundant system (e.g. faulty diagnostic software).
- A delayed shutdown rather than a total defeat of the shutdown function.

There are also mitigations that would need to be taken into account. For example:

- The identification of precise failure modes that defeat a shutdown means that there are often no specific mitigations. However, the failure mode that defeats the shutdown is often one of many. The mitigation is a result of taking into account the fact that another failure may occur first that does not defeat the shutdown. This could be verified by examining the impact of actual historic errors.

The frequency component is also influenced by the number of demands on the component.

The rules needed in sentencing are for:

- Moving from a continuous operation, or that of many demands, to figures for failures per demand.
- When a second software failure is needed to expose the fault we need a rule for the reduction in frequency index for defeating the safety function.
- The impact of the hardware failure rate that can lead to a demand on the fault detection and management software. These failures often only lead to loss of a single component in a redundant architecture and the SCI is normally dominated by the consequence of other common mode failures.

4.2 Assigning a criticality index (SCI)

The results of the Hazops were reviewed and consolidated and then the definition of SCI was applied to each of the functions considered. For each top-level function, we considered features that impact the consequence, frequency or mitigation. Further analyses on the overall results were performed in order to:

- Derive the worst case criticality index in each risk category (e.g. spurious execution of safety function, safety function defeated) for each procedure in the Hazops, by taking the maximum over the subfunctions considered.
- Derive the worst case criticality index over all categories for each procedure in the Hazops.
- Map each procedure to the criticality associated with its top-level caller. Where there were many callers, the maximum value was taken.
- Summarise the criticality assignment to procedures by showing the number of procedures of each criticality.

This analysis required only the correct assignment of SCIs to the top-level procedures. Propagation down the tree is a mechanical matter of taking for each procedure the maximum criticality of all the top-level procedures from which it can be reached, and so to which it can contribute. The coverage obtained in this way can be checked automatically and showed that the analysis had covered all the procedures in the build of the software analysed.

These rules for propagation of the criticality index down the call tree are conservative and another pass of the SCI assignments was later undertaken to remove some of the conservatism during the SCA validation.

5 Validation of the SCA

The criticality analysis required validation that:

1. The analysis was applied to the correct object, i.e.:
 - It applied to the actual code being used in the system, i.e. the documents used all related to the same versions as the product. This was addressed by the re-engineering of the program structure from the actual code.
 - The Hazops were adequately undertaken (with respect to procedure, people and documentation). This was addressed by the review of the results and by the check on consistency of the SCI assignment as described in Section 5.1.
2. The assumptions of behaviour of software and surrounding system were correct, i.e. that:
 - The judgements used about the code partitioning were correct. Verification of this involved covert channel analysis (see Section 5.2) and was addressed by subsequent static analysis of pointers [1] and by the trust placed in the reverse engineering and expert judgement.
 - The mitigations assumed (e.g. benefits from redundant channels, time skew) were correct. The behaviour of the software was as assumed by the experts. This could be checked by confirmatory tests or analysis.

The process of assigning software criticality indices to procedures depends on analysis by domain experts and on a detailed understanding of the function of the code and the system level consequences of failure. When dealing with a large body of complex code, it is possible that this analysis will miss some aspect of the function that should affect the criticality assigned. One way to verify the self-consistency of an assignment is to examine the interconnections between procedures that should place a constraint on their relative criticalities. This can be done mechanically and helps to build confidence in the result (although clearly it is possible for an assignment to be internally consistent but not meet the external criticality requirements of the application).

There are three aspects to such verification. The first is to decide what consistency conditions should be satisfied by an assignment of criticalities. These will be defined in Section 5.1. The second is to determine how these conditions can be evaluated mechanically. Finally, where this information cannot be extracted completely and exactly, we need to decide whether we can

make a useful approximation, and whether that approximation is conservative. A restricted analysis limits the amount of consistency checking that can be done, while a conservative analysis may result in apparent inconsistencies that further investigation shows not to be problems with the assignment. Section 6 summarises the results from an application of these methods and describes how the apparent inconsistencies found were resolved.

5.1 Consistency of SCI assignments

The consistency checks we made were based on the data flows between procedures. We assigned to each variable the maximum criticality of any procedure that used its value, and checked that any procedure that affected its value had a criticality at least as high. (This makes the conservative assumption that all variables contribute to the critical function of the procedure). In these consistency checks we considered three ways in which one procedure can affect a value in another:

- **Downwards call consistency.** This condition addresses the constraint imposed by the communication of results from a called procedure to a calling one through its results or through pointers to local variables in its parameter list. We made the conservative assumption that all called procedure results can affect any calling procedure result, and hence a called procedure should be no less critical than its caller.
- **Static data consistency.** This condition addresses the constraints placed on the criticality of a procedure by the static variables it modifies. The criticality of a procedure that changes a static variable must be at least as high as the criticality of the variable.
- **Upwards call consistency.** One procedure may call another that changes static variables, either directly or through a chain of intermediate procedures. Under the conservative assumption that all variables affect the value, the criticality of every caller should thus be at least as high as the criticality of the callee.

Checking the consistency conditions above required information about the code, for example the call graph and the subset of procedures returning results directly to their callers, either by an explicit return parameter or via pointers to local variables. This information was extracted mechanically using CodeSurfer. However, the code was too large to analyse in one piece, so it was divided into smaller components (projects). A reasonable amount of effort was needed to synthesise a view of the whole project in a conservative way.

5.2 Covert channels

In classical safety analysis the idea of a segregation domain is used to define a region within which a failure can propagate. This is done so that interaction of, say, hardware components of a physical plant are not overlooked when a functional or abstract view of the system is considered. A segregation domain defines an area where common cause failures are likely.

In assessing safety-critical software a great deal of effort is usually placed on trying to show segregation or non-interference of software components. This often involves sophisticated and extensive analysis and a design for assurance approach that builds in such segregation. Indeed the level of granularity that we take for a software component is strongly influenced by our ability to demonstrate a segregation domain: there may be no point in producing a very refined analysis within a segregation domain.

When we are dealing with software, the representation used defines certain types of interaction (e.g. dataflow) as intended by the designer. It is these representations that are used as a basis for criticality analysis, but the possibility remains that there are unintended interactions or interactions not captured by the notation (e.g. dynamic pointer allocation). Furthermore,

some SOUP might not even have this documentation, so that we might be relying on expert judgement to assess the possible flows.

Therefore the possibility remains of what, by analogy with the security area, we term covert channels or flows. Covert channel analysis should be addressed by the use of the guide words in the Hazops analysis and by being cautious about what is rejected as incredible. Other techniques that can possibly be used for covert channel analysis include:

- manual review
- tool-supported review (e.g. program slicing using a tool like CodeSurfer [5])
- symbolic execution (e.g. pointer analysis using Polyspace [6])
- static flow analysis
- formal verification

In the application discussed here this was addressed by program analysis and review.

6 Discussion

This section shows the results of the more detailed analysis and it summarises the issues raised in the development and application of the SCA approach. Some of these issues are common to the subsequent work on static analysis (see [\[1\]](#)).

6.1 Impact of more detailed analysis

The more detailed final analysis (involving the re-engineered call trees and the conservative, mechanical propagation of SCIs from the top-level functions downwards) identified many more unused procedures (about 30%). This resulted in a corresponding reduction in the qualification work required. The number of SCI 4 procedures was also reduced, but the detailed assessment has, in many cases, found at least one route by which the top-level procedures can become SCI 3, that was not visible in the initial analysis, so procedures tended to move from SCI 2 to SCI 3.

The impact of the more detailed analysis on the proportion of procedures in each SCI category is shown below. The initial SCI was based on the software architecture partitions.

SCI	Percentage of procedures in each SCI category	
	Initial SCI	Detailed SCI
0	10	40
1	20	5
2	40	0
3	0	35
4	30	20

As one might expect, partitions at level 4 are not homogeneously critical and contain a range of functions. It also shows that although the initial broad categorisations are indicative, it would not be sufficient to assure to the average of the partition: the detailed assignment is important.

6.2 Tools

There were problems of scale – CodeSurfer was unable to analyse the program in one go. We needed to split the software into separate projects then devise means for checking for interconnections between projects (via the database of information extracted from all the projects). Clearly tools to cope with more complex systems are desirable and this may mean that tools may need significant redesign, for example, to incrementally build a database of results. While our application had some particular pointer complexities it would seem that about 100kloc is the limit for this tool at present. Other program slicers [7] claim to scale but these comparisons are not properly benchmarked.

6.3 Levels of detail and diminishing returns

There are clearly diminishing returns in the level of detail in the SCA. More detail might limit the amount of software to be analysed, but takes more time to justify and demonstrate segregation and there is a trade-off against having more time for the actual analysis. Progressive application may be the best way of proceeding and this is what we recommend in the following table taken from [2].

SCA	Based on	Remarks
Initial SCA	Expert judgement Top level architecture High level description	If it exists, architecture may not reflect functionality so need to go a level lower.
Design SCA	Software design descriptions	If it exists, description may not capture behaviour sufficiently so need to appeal to expert judgement and code review.
Code SCA	Source code/assembler	May be too detailed to abstract behaviour. Need tools to extract control, data and information flow.

The optimum level of SCA will depend on the costs associated with the subsequent assurance activities. Assurance could make use of testing, static analysis, inspection and evaluation of field experience. Where these costs increase significantly with SCI, expending more SCA effort to limit downstream assurance costs can be justified. On the other hand, where the downstream cost is low, or relatively insensitive to SCI, the optimum choice may be to assure all software to the maximum SCI.

There are also pragmatic issues that influence the level of detail needed in an SCA. If the SCA is being used to decide the amount of testing required, all that might be required is an estimate for the highest SCI in a testable collection of software components (some part of the pruned call tree).

6.4 Hazops

A range of expertise is needed for the safety analysis: expertise in the software, the computer equipment and application needed to assess failure consequences. The volume of work can be

significant if there are many top-level functions to assess. This has an impact on resources, of course, and also on the credibility of the results as large Hazops can become difficult to sustain when there are only a few experts available.

The representation used for the Hazops is a key issue. The differences between the actual code and the block diagrams and the prevalence of unused code (not just from configuration of the product but also left in from development and testing) meant that validation of the SCI required detailed analysis at the code level.

There are also some issues with the Hazops process as to how much can be done by individuals and then reviewed (a desktop approach) and how much needs to be done by groups. Apart from formal outputs, the Hazops led to increased understanding of the system and it would be useful to somehow recognise this.

Hazops requires knowledge of the consequence of failures. There is a need to validate some of these judgements about the behaviour of complex systems: failure impact may be amenable to some automated support (e.g. see for example [8]).

6.5 SCI scheme

There is an issue of how to have a consistent index when some of the application is demand driven and some continuous – this is also an issue with the use of IEC61508 and SILs generally.

In the safety application, there was a need for continuous control of the process during the shutdown that can take many hours. This meant that fail-stop was not an option for that part of the equipment and this can complicate the SCI and safety arguments. It meant that the use of arguments that faults would be detected is not sufficient.

The SCI scheme is necessary for COTS/SOUP but the definitions and approach could be put on a more rigorous basis. The link to standards could also be elaborated and the need for pre-defined categories related to SILs discussed (see for example the proposals in [9]).

6.6 SCA validation

There was a problem of conservatism causing everything to float up a criticality level (as is often the case with multilevel security). We therefore need conservative but usable propagation rules. The detailed SCA reduced the amount of level 4 software at the expense of needing to show certain verifications. The pragmatic approach to reducing the propagation was to verify some properties so that, given this verification, the SCI assignments were correct.

6.7 Overall assurance activities

The criticality analysis provides a focus for the overall assurance activities and the safety case design. In practice, there is the issue of how many criticality indexes levels are really needed. More importantly we lack an empirical evidence (rather than expert judgement, compliance with standards) of what techniques and what rigour is needed for the different criticality indexes and the different types of software. This is an unresolved problem in all current standards.

7 Conclusions

The overall conclusions of this work are that:

1. Software criticality analysis enabled the safety assurance activities to be focused on specific software components. Not only does this bring safety benefits but also reduced the effort required for a whole range of subsequent validation activities – software reviews, testing, field experience studies as well as static analysis.
2. A useful by-product of SCA is the identification of the software and its structure – this provides an independent assessment of the system design documents and where these are of doubtful provenance provides some re-engineering of them.
3. The Hazops method can be adapted to assess the safety impact of software component failures.
4. The safety criticality index (SCI) used to classify the criticality of software components was useful, but in practice the number of levels requiring different treatment in subsequent qualification is fairly small. So there is scope for simplification and closer coupling between the definition of the SCI and its subsequent use.
5. The level of detail considered in the SCA should be a trade-off between the effort required to validate software at a given criticality and the effort required to justify the partitioning of components into different criticality levels. A more detailed SCA might reduce the amount of software classified at the highest criticality, but the effort required for classification might be more than the validation effort saved. The optimum level depends on the effort required in the subsequent validation activities.
6. There is a need for improved tool support for analysing code structure and for assessing the impact of failures. This is especially true if the approach here is to be increased in scale to even larger systems. The use of different programming languages or the adherence to safe subsets and coding guidelines would increase the scale of what the tools can cope with. However, many systems of interest will be written in C, developed over many years by different groups of people.
7. In general, some expertise in the application, design and code is needed to make the SCA tractable. However, in the cases when expertise is not available, this may be compensated by further testing, static analysis or code review. Further conclusions on the generality of the approach will result from our current work on applying the SCA approach to a different class of SOUP/COTS.

Acknowledgements

This work was partly funded under the HSE Generic Nuclear Safety Research Programme under contract 40051634 and is published with the permission of the Industry Management Committee (IMC). This work was assisted by the EPSRC Dependability Interdisciplinary Research Collaboration (DIRC), grant GR/N13999.

The views expressed in this report are those of the authors and do not necessarily represent the views of the members of the IMC or the Health and Safety Commission/Executive. The IMC does not accept liability for any damage or loss incurred as a result of the information contained in this paper.

All necessary clearances for the publication of this paper have been obtained. If accepted, the author will prepare the final manuscript in time for inclusion in the conference proceedings and will present the paper at the conference.

References

- [1] PG Bishop, RE Bloomfield, Tim Clement, Sofia Guerra and Claire Jones. *Static Analysis of COTS Used in Safety Application*. Adelard document D198/4308/2, 2001.
- [2] PG Bishop, RE Bloomfield and PKD Froome. *Justifying the use of software of uncertain pedigree (SOUP) in safety-related applications*. Report No: CRR336 HSE Books 2001 ISBN 0 7176 2010 7, http://www.hse.gov.uk/research/crr_pdf/2001/crr01336.pdf.
- [3] Interim Defence Standard 00-58, *Hazop studies on Systems Containing Programmable Electronics. Part 1: Requirements. Part 2: General Application Guidance*. Issue 2 MoD 2000.
- [4] D J Burns, R M Pitblado, *A Modified Hazop Methodology for Safety Critical System Assessment*, in *Directions in Safety-critical Systems*, Felix Redmill and Tom Anderson (eds), Springer Verlag, 1993.
- [5] *CodeSurfer user guide and technical reference*. Version 1.0, Grammatech, 1999.
- [6] *PolySpace Technologies*, <http://www.polyspace.com>.
- [7] F. Tip, "A Survey of Program Slicing Techniques", *Journal of Programming Languages*, Vol.3, No.3, pp.121-189, September, 1995. <http://citeseer.nj.nec.com/tip95survey.html>
- [8] T Cichocki and J Gorski, *Formal support for fault modelling and analysis*, in U Voges (ed): SAFECOMP 2001, LNCS 2187, pp 202-211, Springer-Verlag, 2001.
- [9] Rainer Faller, *Project Experience with IEC 61508 and Its Consequences*, in U Voges (ed): SAFECOMP 2001, LNCS 2187, pp 212-226, Springer-Verlag, 2001.

Appendix A Treating the SCI probabilistically

There is a probabilistic basis to the SCI assignment formula given in the main body of the report.

Let us define:

$riskP$	the annual risk of demand failures
$conP$	the consequences of the protection system failing on demand
$pdfP$	the probability of failure of the protection system per demand
$demand_rate$	trip demands per year

For each software component i , let us define:

$risk_i$	the risk from the component i failing on demand
con_i	the consequence of component i failing on demand
pf_sw_i	the probability of failure on demand given erroneous behaviour of software component i
psw_err_i	probability of erroneous behaviour in component i given a software demand
nsw_dem_i	number of times software component i is activated on a demand

For a given safety function, the risk is:

$$riskP = conP \times pdfP \times demand_rate$$

For a software component of the system:

$$risk_i = con_i \times pf_{sw_i} \times psw_{err_i} \times nsw_{dem_i} \times demand_{rate}$$

We can express this as a relative risk:

$$risk_{ratio_i} = con_i \times pf_{sw_i} \times psw_{err_i} \times nsw_{dem_i} / (conP \cdot pfdP)$$

We are interested in the impact of the erroneous behaviour software component i . If we set $psw_{err_i} = 1$ or differentiate with respect psw_{fl_i} , we obtain:

$$criticality_i = (con_i / conP) \times pf_{sw_i} \times nsw_{dem_i} / pfdP$$

Taking logs to get an index of importance and defining:

$$c_i = \log (con_i / conP)$$

$$m_i = \log (1/pf_{sw_i}) \text{ (as } pf_{sw} < 1, m_i \text{ is a positive integer } \geq 0)$$

$$f_i = \log (nsw_{dem_i})$$

$$sci_i = \log (criticality_i)$$

$$SIL = \log(1/pfdP) \quad \text{(relates the lower bound on the pfd target)}$$

We simply get:

$$sci_i = SIL + c_i + f_i - m_i$$

In practice the sentencing scheme described in the main paper uses a consequence index that was a combination of $SIL + c_i$. In the equation above c_i is a negative index as con_i may be less severe than the $conP$ consequence that may arise in the worst case (e.g. delayed trip rather than no trip). Hence the overall consequence is related to the SIL (0..4) of the safety function, but modified by consideration of the expected result of the component failure.

If the software component can contribute to different safety function failures, then a similar calculation is required for each one. The maximum criticality of the software component can then be determined.