ISO/IEC JTC 1/SC 22/OWGV N0074

Editor's draft 3 of PDTR 24772, 01 June 2007

ISO/IEC JTC 1/SC 22 N 0000

Date: 2007-06-01

ISO/IEC PDTR 24772

ISO/IEC JTC 1/SC 22/OWG

Secretariat: ANSI

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

Élément introductif — Élément principal — Partie n: Titre de la partie

Document type: International standard
Document subtype: if applicable
Document stage: (20) Preparation
Document language: E

Contents

Page

Foreword

ISO (the International Organization for Standardization) is a worldwide federation of national standards bodies (ISO member bodies). The work of preparing International Standards is normally carried out through ISO technical committees. Each member body interested in a subject for which a technical committee has been established has the right to be represented on that committee. International organizations, governmental and non-governmental, in liaison with ISO, also take part in the work. ISO collaborates closely with the International Electrotechnical Commission (IEC) on all matters of electrotechnical standardization.

International Standards are drafted in accordance with the rules given in the ISO/IEC Directives, Part 2.

The main task of technical committees is to prepare International Standards. Draft International Standards adopted by the technical committees are circulated to the member bodies for voting. Publication as an International Standard requires approval by at least 75 % of the member bodies casting a vote.

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights.

ISO/IEC TR 24772 which is a Technical Report of type 3, was prepared by Joint Technical Committee ISO/IEC JTC 1, Subcommittee SC 22, Programming Languages.

Introduction

A paragraph.

The **introduction** is an optional preliminary element used, if required, to give specific information or commentary about the technical content of the document, and about the reasons prompting its preparation. It shall not contain requirements.

The introduction shall not be numbered unless there is a need to create numbered subdivisions. In this case, it shall be numbered 0, with subclauses being numbered 0.1, 0.2, etc. Any numbered figure, table, displayed formula or footnote shall be numbered normally beginning with 1.

Information Technology — Programming Languages — Guidance to Avoiding Vulnerabilities in Programming Languages through Language Selection and Use

1 Scope

1.1 In Scope

1) Applicable to the computer programming languages covered in this document.
2) Applicable to software written, reviewed and maintained for any application.
3) Applicable in any context where assured behavior is required, e.g. security, safety, mission/business criticality etc.

1.2 Not in Scope

This technical report does not address software engineering and management issues such as how to design and implement programs, using configuration management, managerial processes etc.

The specification of the application is *not* within the scope.

1.3 Approach

The impact of the guidelines in this technical report are likely to be highly leveraged in that they are likely to affect many times more people than the number that worked on them. This leverage means that these guidelines have the potential to make large savings, for a small cost, or to generate large unnecessary costs, for little benefit. For these reasons this technical report has taken a cautious approach to creating guideline recommendations. New guideline recommendations can be added over time, as practical experience and experimental evidence is accumulated.

Some of the reasons why a guideline might generate unnecessary costs include:

1) Little hard information is available on which guideline recommendations might be cost effective
2) It is likely to be difficult to withdraw a guideline recommendation once it has been published
3) Premature creation of a guideline recommendation can result in:
   i. Unnecessary enforcement coast (i.e., if a given recommendation is later found to be not worthwhile).
   ii. Potentially unnecessary program development costs through having to specify and use alternative constructs during software development.
   iii. A reduction in developer confidence of the worth of these guidelines.

1.4 Intended Audience

1.4.1 Safety

1.4.2 Security

1.4.3 Predictability

The programmers who may benefit from this document include those who are primarily experts in areas other than programming and who need to use computation as part of their work. These programmers include scientists, engineers, economists, and statisticians. These programmers require high confidence in the applications they write and use due to the increasing complexity of the calculations made (and the consequent use of teams of programmers each contributing expertise in a portion of the calculation), due to the costs of invalid results, or due to the expense of individual calculations implied by a very large number of processors used and/or very long execution times needed to complete the calculations. These circumstances give a consequent need for high reliability and motivate the need felt by these programmers for the guidance offered in this document.

1.4.4 Software Assurance

1.5 How to Use This Document

1.5.1 Writing Profiles

[*Note*: Advice for writing profiles was discussed in London 2006, no words]

2 Normative references

The following referenced documents are indispensable for the application of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

3 Terms and definitions

For the purposes of this document, the following terms and definitions apply.

3.1 Language Vulnerability

A feature or combination of features of a programming language which can cause, or is strongly correlated with, a weakness, a hazard, or a bug.

3.2 Application Vulnerability

A security vulnerability or safety hazard.

3.3 Security Vulnerability

A set of conditions that allows an attacker to violate an explicit or implicit security policy.

3.4 Safety Hazard

Should definition come from, IEEE 1012-2004 IEEE Standard for Software Verification and Validation, 3.1.11, IEEE Std 1228-1994 IEEE Standard for Software Safety Plans, 3.1.5, IEEE Std 1228-1994 IEEE Standard for Software Safety Plans, 3.1.8 *or* IEC 61508-4 and ISO/IEC Guide 51*?*

3.5 Safety-critical software

Software for applications where failure can cause very serious consequences such as human injury or death.

3.6 Software quality

The degree to which software implements the needs described by its specification.

3.7 Predictable Execution

The property of the program such that all possible executions have results which can be predicted from the relevant programming language definition and any relevant language-defined implementation characteristics and knowledge of the universe of execution.

Note: In some environments, this would raise issues regarding numerical stability, exceptional processing, and concurrent execution.

Note: Predictable execution is an ideal which must be approached keeping in mind the limits of human capability, knowledge, availability of tools etc. Neither this nor any standard ensures predictable execution. Rather this standard provides advice on improving predictability. The purpose of this document is to assist a reasonably competent programmer approach the ideal of predictable execution.

4 Symbols (and abbreviated terms)

5 Vulnerability issues

Software vulnerabilities are unwanted characteristics of software that may allow software to behave in ways that are unexpected by a reasonably sophisticated user of the software. The expectations of a reasonably sophisticated user of software may be set by the software's documentation or by experience with similar software. Programmers build vulnerabilities into software by failing to understand the expected behavior (the software requirements), or by failing to correctly translate the expected behavior into the actual behavior of the software.

This document does not discuss a programmer's understanding of software requirements. This document does not discuss software engineering issues per se. This document does not discuss configuration management; build environments, code-checking tools, nor software testing. This document does not discuss the classification of software vulnerabilities according to safety or security concerns. This document does not discuss the costs of software vulnerabilities, nor the costs of preventing them.

This document does discuss a reasonably competent programmer's failure to translate the understood requirements into correctly functioning software. This document does discuss programming language features known to contribute to software vulnerabilities. That is, this document discusses issues arising from those features of programming languages found to increase the frequency of occurrence of software vulnerabilities. The intention is to provide guidance to those who wish to specify coding guidelines for their own particular use.

A programmer writes source code in a programming language to translate the understood requirements into working software. The programmer combines in sequence language features (functional pieces) expressed in the programming language so the cumulative effect is a written expression of the software's behavior.

A program's expected behavior might be stated in a complex technical document, that can result in a complex sequence of features of the programming language. Software vulnerabilities occur when a reasonably competent programmer fails to understand the totality of the effects of the language features combined to make the resulting software. The overall software may be a very complex technical document itself (written in a programming language whose definition is also a complex technical document).

Humans understand very complex situations by chunking, that is, by understanding pieces in a hierarchal scaled scheme. The programmer's initial choice of the chunk for software is the line of code. (In any particular case, subsequent analysis by a programmer may refine or enlarge this initial chunk.) The line of code is a reasonable initial choice because programming editors display source code lines. Programming languages are often defined in terms of statements (among other units), which in many cases are synonymous with textual lines. Debuggers may execute programs stopping after every statement to allow inspection of the program's state. Program size and complexity is often estimated by the number of lines of code (automatically counted without regard to language statements).

5.1 Issues arising from lack of knowledge

While there are many thousands of programmers in the world, there are only several tens of authors engaged in designing and specifying those programming languages defined by international standards. The design and specification of a programming language is very different than programming. Programming involves selecting and sequentially combining features from the programming language to (locally) implement specific steps of the software's design. In contrast, the design and specification of a programming language involves (global) consideration of all aspects of the programming language. This must include how all the features will interact with each other, and what effects each will have, separately and in any combination, under all foreseeable circumstances. Thus, language design has global elements that are not generally present in any local programming task.

The creation of the abstractions which become programming language standards therefore involve consideration of issues unneeded in many cases of actual programming. Therefore perhaps these issues are not routinely considered when programming in the resulting language. These global issues may motivate the definition of subtle distinctions or changes of state not apparent in the usual case wherein a particular language feature is used. Authors of programming languages may also desire to maintain compatibility with older versions of their language while adding more modern features to their language and so add what appears to be an inconsistency to the language.

A reasonably competent programmer therefore may not consider the full meaning of every language feature used, as only the desired (local or subset) meaning may correspond to the programmer's immediate intention. In consequence, a subset meaning of any feature may be prominent in the programmer's overall experience.

Further, the combination of features indicated by a complex programming goal can raise the combinations of effects making a complex aggregation within which some of the effects are not intended**.**

5.1.1 Issues arising from unspecified behaviour

While every language standard attempts to specify how software written in the language will behave in all circumstances, there will always be some behavior which is not specified completely. In any circumstance, of course, a particular compiler will produce a program with some specific behavior (or fail to compile the program at all). Where a programming language is insufficiently well defined, different compilers may differ in the behavior of the resulting software. The authors of language standards often have an interpretations or defects process in place to treat these situations once they become known, and, eventually, to specify one behavior. However, the time needed by the process to produce corrections to the language standard is often long, as careful consideration of the issues involved is needed.

When programs are compiled with only one compiler, the programmer may not be aware when behavior not specified by the standard has been produced. Programs relying upon behavior not specified by the language standard may behave differently when they are compiled with different compilers. An experienced programmer may choose to use more than one compiler, even in one environment, in order to obtain diagnostics from more than one source. In this usage, any particular compiler must be considered to be a different compiler if it is used with different options (which can give it different behavior), or is a different release of the same compiler (which may have different default options or may generate different code), or is on different hardware (which may have a different instruction set). In this usage, a different computer may be the same hardware with a different operating system, with different compilers installed, with different software libraries available, with a different release of the same operating system, or with a different operating system configuration.

5.1.2 Issues arising from implementation defined behaviour

In some situations, a programming language standard may specifically allow compilers to give a range of behavior to a given language feature or combination of features. This may enable more efficient execution on a wider range of hardware, or enable use of the language in a wider variety of circumstances.

The authors of language standards are encouraged to provide lists of all allowed variation of behavior (as many already do). Such a summary will benefit applications programmers, those who define applications coding standards, and those who make code-checking tools.

5.1.3 Issues arising from undefined behaviour

In some situations, a programming language standard may specify that program behavior is undefined. While the authors of language standards naturally try to minimize these situations, they may be inevitable when attempting to define software recovery from errors, or other situations recognized as being incapable of precise definition.

Generally, the amount of resources available to a program (memory, file storage, processor speed) is not specified by a language standard. The form of file names acceptable to the operating system is not specified (other than being expressed as characters). The means of preparing source code for execution may not be specified by a language standard.

5.2 Issues arising from human cognitive limitations

The authors of programming language standards try to define programming languages in a consistent way, so that a programmer will see a consistent interface to the underlying functionality. Such consistency is intended to ease the programmer's process of selecting language features, by making different functionality available as regular variation of the syntax of the programming language. However, this goal may impose limitations on the variety of syntax used, and may result in similar syntax used for different purposes, or even in the same syntax element having different meanings within different contexts.

Any such situation imposes a strain on the programmer's limited human cognitive abilities to distinguish the relationship between the totality of effects of these constructs and the underlying behavior actually intended during software construction.

Attempts by language authors to have distinct the language features expressed by very different syntax may easily result in different programmers preferring to use different subsets of the entire language. This imposes a substantial difficulty to anyone who wants to employ teams of programmers to make whole software products or to maintain software written over time by several programmers. In short, it imposes a barrier to those who want to employ coding standards of any kind. The use of different subsets of a programming language may also render a programmer less able to understand other programmer's code. The effect on maintenance programmers can be especially severe.

5.3 Predictable execution

If a reasonably competent programmer has a good understanding of the state of a program after reading source code as far as a particular line of code, the programmer ought to have a good understanding of the state of the program after reading the next line of code. However, some features, or, more likely, some combinations of features, of programming languages are associated with relatively decreased rates of the programmer's maintaining their understanding as they read through a program. It is these features and combinations of features which are indicated in this document, along with ways to increase the programmer's understanding as code is read.

Here, the term understanding means the programmer's recognition of all effects, including subtle or unintended changes of state, of any language feature or combination of features appearing in the program. This view does not imply that programmers only read code from beginning to end. It is simply a statement that a line of code changes the state of a program, and that a reasonably competent programmer ought to understand the state of the program both before and after reading any line of code. As a first approximation (only), code is interpreted line by line.

5.4 Portability

The representation of characters, the representation of true/false values, the set of valid addresses, the properties and limitations of any (fixed point or floating point) numerical quantities, and the representation of programmer-defined types and classes may vary among hardware, among languages (effecting inter-language software development), and among compilers of a given language. These variations may be the result of hardware differences, operating system differences, library differences, compiler differences, or different configurations of the same compiler (as may be set by environment variables or configuration files). In each of these circumstances, there is an additional burden on the programmer because part of the program's behavior is indicated by a factor that is not a part of the source code. That is, the program's behavior may be indicated by a factor that is invisible when reading the source code. Compilation control schemes (IDE projects, make, and scripts) further complicate this situation by abstracting and manipulating the relevant variables (target platform, compiler options, libraries, and so forth).

Many compilers of standard-defined languages also support language features that are not specified by the language standard. These non-standard features are called extensions. For portability, the programmer must be aware of the language standard, and use only constructs with standard-defined semantics. The motivation to use extensions may include the desire for increased functionality within a particular environment, or increased efficiency on particular hardware. There are well-known software engineering techniques for minimizing the ill effects of extensions; these techniques should be a part of any coding standard where they are needed, and they should be employed whenever extensions are used. These issues are software engineering issues and are not further discussed in this document.

The use of libraries to broaden the software primitives available in a given development environment is a useful technique, allowing the use of trusted functionality directly in the program. Libraries may also allow the program to bind to capabilities provided by its environment. However, these advantages are potentially offset by any lack of skill on the part of the designer of the library (who may have designed subtle or undocumented changes of state into the library's behavior), and implementer of the library (who may not have the implemented the library identically on every platform), and even by the availability of the library on a new platform. The quality of the documentation of a third-party library is another factor that may decrease the reliability of software using a library in a particular situation by failing to describe clearly the library's full behavior. If a library is missing on a new platform, its functionality must be recreated in order to port any software depending upon it**.**

Using a library usually requires that options be set during compilation and linking phases, which constitute a software behavior specification beyond the source code. Again, these issues are software engineering issues and are not further discussed in this document**.**

6. Vulnerabilities

6.1 SM-004 Out of bounds array element access

6.1.1 Description of application vulnerability

Unpredictable behaviour can occur when accessing the elements of an array outside the bounds of the array.

6.1.2 Cross reference

CWE: 129

6.1.3 Categorization

See clause 5.?.

6.1.4 Mechanism of failure

Arrays are defined, perhaps statically, perhaps dynamically, to have given bounds. In order to access an element of the array, index values for one or more dimensions of the array must be computed. If the index values do not fall within the defined bounds of the array, then access might occur to the wrong element of the array, or access might occur to storage that is outside the array. A write to a location outside the array may change the value of other data variables or may even change program code.

6.1.5 Possible ways to avoid the vulnerability

The vulnerability can be avoided by not using arrays, by using whole array operations, by checking and preventing access beyond the bounds of the array, or by catching erroneous accesses when they occur. The compiler might generate appropriate code, the run-time system might perform checking, or the programmer might explicitly code appropriate checks.

6.1.6 Assumed variations among languages

This vulnerability description is intended to be applicable to languages with the following characteristics:

- The size and bounds of arrays and their extents might be statically determinable or dynamic. Some languages provide both capabilities.
- Language implementations might or might not statically detect out of bound access and generate a compile-time diagnostic.
- At run-time the implementation might or might not detect the out of bounds access and provide a notification at run-time. The notification might be treatable by the program or it might not be.
- Accesses might violate the bounds of the entire array or violate the bounds of a particular extent. It is possible that the former is checked and detected by the implementation while the latter is not.
- The information needed to detect the violation might or might not be available depending on the context of use. (For example, passing an array to a subroutine via a pointer might deprive the subroutine of information regarding the size of the array.)

- Some languages provide for whole array operations that may obviate the need to access individual elements.
- Some languages may automatically extend the bounds of an array to accommodate accesses that might otherwise have been beyond the bounds. (This may or may not match the programmer's intent.)

6.1.7 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

- If possible, utilize language features for whole array operations that obviate the need to access individual elements.
- If possible, utilize language features for matching the range of the index variable to the dimension of the array.
- If the compiler can verify correct usage, then no mitigation is required beyond performing the verification.
- If the run-time system can check the validity of the access, then appropriate action may depend upon the usage of the system (e.g. continuing degraded operation in a safety-critical system versus immediate termination of a secure system).
- Otherwise, it is the responsibility of the programmer:
    - to use index variables that can be shown to be constrained within the extent of the array;
    - to explicitly check the values of indexes to ensure that they fall within the bounds of the corresponding dimension of the array;
    - to use library routines that obviate the need to access individual elements; or
    - to provide some other means of assurance that arrays will not be accessed beyond their bounds. Those other means of assurance might include proofs of correctness, analysis with tools, verification techniques, etc.

Annex A
(informative)

Guideline Recommendation Factors

A.1 Factors that need to be covered in a proposed guideline recommendation

These are needed because circumstances might change, for instance:

- Changes to language definition.
- Changes to translator behavior.
- Developer training.
- More effective recommendation discovered.

A.1.1 Expected cost of following a guideline

How to evaluate likely costs.

A.1.2 Expected benefit from following a guideline

How to evaluate likely benefits.

A.2 Language definition

Which language definition to use. For instance, an ISO/IEC Standard, Industry standard, a particular implementation.

Position on use of extensions.

A.3 Measurements of language usage

Occurrences of applicable language constructs in software written for the target market.

How often do the constructs addressed by each guideline recommendation occur.

A.4 Level of expertise.

How much expertise, and in what areas, are the people using the language assumed to have?

Is use of the alternative constructs less likely to result in faults?

A.5 Intended purpose of guidelines

For instance: How the listed guidelines cover the requirements specified in a safety related standard.

A.6 Constructs whose behaviour can very

The different ways in which language definitions specify behaviour that is allowed to vary between implementations and how to go about documenting these cases.

A.7 Example guideline proposal template

A.7.1 Coding Guideline

Anticipated benefit of adhering to guideline

- Cost of moving to a new translator reduced.
- Probability of a fault introduced when new version of translator used reduced.
- Probability of developer making a mistake is reduced.
- Developer mistakes more likely to be detected during development.
- Reduction of future maintenance costs.

Annex B
(informative)
Guideline Selection Process

It is possible to claim that any language construct can be misunderstood by a developer and lead to a failure to predict program behavior. A cost/benefit analysis of each proposed guideline is the solution adopted by this technical report.

The selection process has been based on evidence that the use of a language construct leads to unintended behavior (i.e., a cost) and that the proposed guideline increases the likelihood that the behavior is as intended (i.e., a benefit). The following is a list of the major source of evidence on the use of a language construct and the faults resulting from that use:

- a list of language constructs having undefined, implementation defined, or unspecified behaviours,
- measurements of existing source code. This usage information has included the number of occurrences of uses of the construct and the contexts in which it occurs,
- measurement of faults experienced in existing code,
- measurements of developer knowledge and performance behaviour.

The following are some of the issues that were considered when framing guidelines:

- An attempt was made to be generic to particular kinds of language constructs (i.e., language independent), rather than being language specific.
- Preference was given to wording that is capable of being checked by automated tools.
- Known algorithms for performing various kinds of source code analysis and the properties of those algorithms (i.e., their complexity and running time).

B.1 Cost/Benefit Analysis

The fact that a coding construct is known to be a source of failure to predict correct behavior is not in itself a reason to recommend against its use. Unless the desired algorithmic functionality can be implemented using an alternative construct whose use has more predictable behavior, then there is no benefit in recommending against the use of the original construct.

While the cost/benefit of some guidelines may always come down in favor of them being adhered to (e.g., don't access a variable before it is given a value), the situation may be less clear cut for other guidelines. Providing a summary of the background analysis for each guideline will enable development groups.

Annex A provides a template for the information that should be supplied with each guideline.

It is unlikely that all of the guidelines given in this technical report will be applicable to all application domains.

B.2 Documenting of the selection process

The intended purpose of this documentation is to enable third parties to evaluate:

- the effectiveness of the process that created each guideline,

- the applicability of individual guidelines to a particular project.

Annex C (informative) Template for use in proposing vulnerabilities

C. Skeleton template for use in proposing vulnerabilities

C.1 6.<*x*> <*unique immutable identifier*> <*short title*>

Notes on template header. The number "x" depends on the order in which the vulnerabilities are listed in Clause 6. It will be assigned by the editor. The "unique immutable identifier" is intended to provide an enduring identifier for the vulnerability description, even if their order is changed in the document. The "short title" should be a noun phrase summarizing the description of the application vulnerability. No additional text should appear here.

C.1.1 6.<*x*>.1 Description of application vulnerability

Replace this with a brief description of the application vulnerability. It should be a short paragraph.

C.1.2 6.<*x*>.2 Cross reference

CWE: *Replace this with the CWE identifier. At a later date, other cross-references may be added.*

C.1.3 6.<*x*>.3 Categorization

See clause 5.?. *Replace this with the categorization according to the analysis in Clause 5. At a later date, other categorization schemes may be added.*

C.1.4 6.<*x*>.4 Mechanism of failure

Replace this with a brief description of the mechanism of failure. This description provides the link between the programming language vulnerability and the application vulnerability. It should be a short paragraph.

C.1.5 6.<*x*>.5 Possible ways to avoid the vulnerability

Replace this with a description of the various points at which the chain of causation could be broken. It should be a short paragraph.

C.1.6 6.<*x*>.6 Assumed variations among languages

This vulnerability description is intended to be applicable to languages with the following characteristics:

Replace this with a bullet list summarizing the pertinent range of characteristics of languages for which this discussion is applicable. This list is intended to assist readers attempting to apply the guidance to languages that have not been treated in the language-specific annexes.

C.1.7 6.<*x*>.7 Avoiding the vulnerability or mitigating its effects

Software developers can avoid the vulnerability or mitigate its ill effects in the following ways:

Replace this with a bullet list summarizing various ways in which programmers can avoid the vulnerability or contain its bad effects. Begin with the more direct, concrete, and effective means and then progress to the more indirect, abstract, and probabilistic means.

Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2001

[2] ISO/IEC TR 10000-1, *Information technology — Framework and taxonomy of International Standardized Profiles — Part 1: General principles and documentation framework*

[3] ISO 10241, *International terminology standards — Preparation and layout*

[4] ISO/IEC TR 15942:2000, "Information technology - Programming languages - Guide for the use of the Ada programming language in high integrity systems"

[5] Joint Fighter Air Vehicle: C++ Coding Standards for the System Development and Demonstration Program. Lockheed Martin Corporation. December 2005.

[6] ISO/IEC 9899:1999, *Programming Languages* – C

[7] ISO/IEC 1539-1:2004, *Programming Languages* – Fortran

[8] ISOISO/IEC 8652:1995/Cor 1:2001/Amd 1:2007, Information technology -- *Programming languages* – Ada

[9] ISO/IEC 15291:1999, Information technology - Programming languages - Ada Semantic Interface Specification (ASIS)

[10] Software Considerations in Airborne Systems and Equipment Certification. Issued in the USA by the Requirements and Technical Concepts for Aviation (document RTCA SC167/DO-178B) and in Europe by the European Organization for Civil Aviation Electronics (EUROCAE document ED-12B).December 1992.

[11] IEC 61508: Parts 1-7, Functional safety: safety-related systems. 1998. (Part 3 is concerned with software).

[12] ISO/IEC 15408: 1999 Information technology. Security techniques. Evaluation criteria for IT security.

[13] J Barnes. High Integrity Software - the SPARK Approach to Safety and Security. Addison-Wesley. 2002.

[14] R. Seacord Preliminary draft of the CERT C Programming Language Secure Coding Standard. ISO/IEC JTC 1/SC 22/OWGV N0059, April 2007.

[15] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2004 (second edition)[1].

[1] The first edition should not be used or quoted in this work.