Deleted: TR

Deleted: 1

Formatted: English (US)

Formatted: German

ISO/IEC JTC 1/SC 22 N1204

Style Definition: TOC 3: Tab stops: 17.99 cm, Right,Leader: ...

Date: 20220828

Deleted: 178

Deleted: 0

Deleted: 606

ISO/IEC TR 24772-8

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

DRAFT DRAFT DRAFT

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Vulnerability descriptions for the programming language Fortran

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

Warning

This document is not an ISO International Standard. It is distributed for review and comment. It is subject to change without notice and may not be referred to as an International Standard.

Recipients of this draft are invited to submit, with their comments, notification of any relevant patent rights of which they are aware and to provide supporting documentation.

Document type: International standard
Document subtype: if applicable
Document stage: (10) development stage
Document language: E

This document followed the meeting of 15 August 2022, and consists of a review by the convenor of obvious items to be accepted, such as font changes, ToC updates and simple corrections.

Source edited at meeting 15 August 2022 plus document N1197 from John Reid

In attendance:

Stephen Michell – convenor WG 23

John Reid

Steve Lionel

Erhard Ploedereder

Excused:

Thomas Clune

WORK TO BE DONE

5. General guidance for Fortran: - Confirm or update the guidance. References!

6.X Explicitly consider whether or not the phrase “Follow the guidance of ISO/IEC 24772-1 clause 6.X” is needed for each clause.

6.X Consider in many subclause 6.X.2 the recommendation to use static analysis tools that detect situations that the compiler may not. Statements about optional detection of erroneous situations

6.8 Conversion Error – had been Numeric conversion error, so discussion about conversions between non-numeric types is needed.

6.38 Deep vs shallow copying – needs a complete writeup

Writeup first pass done.

6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] – Needs review

6.43 Redispatching [PPH] – Needs review

6.44 Polymorphic Variables – Needs review

6.59 Concurrency – Activation [CGA]

6.60 Concurrency – Directed termination [CGT]

6.61 Concurrent Data Access [CGX]

6.62 Concurrency – Premature Termination [CGS]

6.63 Protocol Lock Errors [CGM]

Deleted: E

Deleted: 23

Deleted: May

Deleted: 2.

Deleted: Source documents are N1169 (previous version of this document)....

Deleted: ¶
Tom Clune – USA¶
Erhard Ploedereder – liaison¶
Regrets:

Deleted: Vipul Parekh¶

Deleted: ¶

Deleted: ¶

6.64 Uncontrolled Format String [SHL]

6.65 Modifying constants [UJO]

Contents

Table of Contents

Deleted: 16
Deleted: 17
Deleted: 18
Deleted: 18
Deleted: 18
Deleted: 20
Deleted: 21
Deleted: 22
Deleted: 22
Deleted: 24
Deleted: 24
Deleted: 25
Deleted: 26
Deleted: 26
Deleted: 27
Deleted: 27
Deleted: 28
Deleted: 28
Deleted: 29
Deleted: 29
Deleted: 30

Deleted: 31
Deleted: 31
Deleted: 32
Deleted: 32
Deleted: 33
Deleted: 33
Deleted: 34
Deleted: 34
Deleted: 35
Deleted: 36
Deleted: 36
Deleted: 37
Deleted: 37
Deleted: 38
Deleted: 38
Deleted: 39
Deleted: 39
Deleted: 39
Deleted: 40
Deleted: 40
Deleted: 41
Deleted: 42
Deleted: 42
Deleted: 42
Deleted: 43
Deleted: 43
Deleted: 44
Deleted: 44
Deleted: 44
Deleted: 45
Deleted: 45
Deleted: 46
Deleted: 46
Deleted: 47
Deleted: 47
Deleted: 48
Deleted: 48
Deleted: 48

Deleted: 49

Deleted: 49

Deleted: 50

Deleted: 52

Page

Foreword

ISO (the International Organization for Standardization) and IEC (the International Electrotechnical Commission) form the specialized system for worldwide standardization. National bodies that are members of ISO or IEC participate in the development of International Standards through technical committees established by the respective organization to deal with particular fields of technical activity. ISO and IEC technical committees collaborate in fields of mutual interest. Other international organizations, governmental and non-governmental, in liaison with ISO and IEC, also take part in the work. In the field of information technology, ISO and IEC have established a joint technical committee, ISO/IEC JTC 1.

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

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

In exceptional circumstances, when the joint technical committee has collected data of a different kind from that which is normally published as an International Standard ("state of the art", for example), it may decide to publish a Technical Report. A Technical Report is entirely informative in nature and shall be subject to review every five years in the same manner as an International Standard.

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

ISO/IEC 24772-8, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

Introduction

This Technical Report provides guidance for the programming language Fortran so that application developers considering Fortran or using Fortran will be better able to avoid the programming constructs that lead to vulnerabilities in software written in the Fortran language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This technical can also be used in comparison with companion technical reports and with the language-independent report, 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

It should be noted that this Technical Report is inherently incomplete. It is not possible to provide a complete list of programming language vulnerabilities because new weaknesses are discovered continually. Any such report can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use – Vulnerability descriptions for the programming language Fortran

1. Scope

This Technical Report specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.

Vulnerabilities described in this technical report document the way that the vulnerability described in the language-independent writeup (in 24772-1) are manifested in Fortran.

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.

ISO/IEC 24772-1 Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages, Part 1, General Guidance
ISO/IEC 1539-1:2018, Information technology -- Programming languages -- Fortran -- Part 1: Base language
ISO 80000–2:2009, *Quantities and units — Part 2: Mathematical signs and symbols to be use in the natural sciences and technology*
ISO/IEC 2382–1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

ISO/IEC/IEEE 60559-2011, Information technology – Microprocessor Systems – Floating-Point arithmetic

3. Terms and definitions, symbols and conventions

3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382–1, in 24772-1, ISO/IEC 1539-1:2018 and the following apply. Other terms are defined where they appear in *italic* type.

The precise statement of the following definitions can be found in the Fortran standard.

3.2

argument association

association between an effective argument and a dummy argument

3.3

assumed-shape array

a dummy argument array whose shape is assumed from the corresponding actual argument

assumed-size array: a dummy argument array whose size is assumed from the corresponding actual argument

3.4

deleted feature

a feature that existed in older versions of Fortran but has been removed from later versions of the standard

explicit interface: an interface of a procedure that includes all the characteristics of the procedure and names for its dummy arguments

image: one of a mutually cooperating set of instances of a Fortran program; each has its own execution state and set of data objects

implicit typing: an archaic rule that declares a variable upon use according to the first letter of its name

kind type parameter: a value that determines one of a set of processor-dependent data representation methods

module: a separate scope that contains definitions that can be accessed from other scopes

obsolescent feature: a feature that is not recommended because better methods exist in the current standard

processor: combination of computing system and mechanism by which programs are transformed for use on that computing system

processor dependent: not completely specified in the Fortran standard, having one of a set of methods and semantics determined by the processor

pure procedure: a procedure subject to constraints such that its execution has no side effects

type: named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values

4 Language concepts

4.1 General

Fortran is the oldest international standard programming language with the first Fortran processors appearing over fifty years ago. During half a century of computing, computing technology has changed immensely, and Fortran has evolved via several revisions of the standard. Also, during half a century of computing and in response to customer demand, some popular processors supported extensions. There remains a substantial body of Fortran code that is written to previous versions of the standard or with extensions to previous versions, and before modern techniques of software development came into widespread use. The process of revising the standard has been done carefully with a goal of protecting applications programmers' investments in older codes.

Very few features were deleted from older revisions of the standard; those that were deleted were little used, or redundant with a superior alternative, or error-prone with a safer alternative. Many modern processors generally continue to support deleted features from older revisions of the Fortran standard, and even some extensions from older processors, and do so with the intention of reproducing the original semantics. Also, there exist automatic means of replacing at least some archaic features with modern alternatives. Even with automatic assistance, there might be reluctance to change existing software due to its having proven itself through usage on a wider variety of hardware than is in general use at present, or due to issues of regulation or certification. The decision to modernize trusted software is made cognizant of many factors, including the availability of resources to do so and the perceived benefits. This document does not attempt to specify criteria for modernizing trusted old code.

4.2 Fortran standard concepts and terminology

The Fortran standard, ISO/IEC 1539-1 is written in terms of a *processor* which includes the language translator (that is, the compiler or interpreter, and supporting libraries), the operating system (affecting, for example, how files are stored, or which files are available to a program), and the hardware (affecting, for example, the machine representation of numbers or the availability of a clock). The Fortran standard specifies how the contents of files are interpreted. The standard does not specify the size or complexity of a program that might cause a processor to fail.

A program conforms to the Fortran standard if it uses only forms and relationships between forms specified by the standard and does so with the interpretation given by the standard. A program unit is standard-conforming if it can be included in an otherwise standard-conforming program in a way that is standard conforming.

The Fortran standard allows a processor to support features, not defined by the standard, provided such features do not contradict the standard. Use of such features, called *extensions* in this document, should be avoided. Processors are able to detect and report the use of some extensions.

This document assumes that diagnostics for non-standard forms and relationships are always enabled.

4.3 Deleted and redundant features

Annexes B.1 and B.2 of ISO/IEC 1539-1:2018 standard lists eight features of older versions of Fortran that have been deleted because they were redundant and considered largely unused. Although no longer part of the standard, they are supported by many processors to allow old programs to continue to run. Annex B.3 lists twelve features of Fortran that are regarded as obsolescent because they are redundant – better methods are available in the current standard. The obsolescent features are described in the standard using a small font. The use of any deleted or obsolescent feature should be avoided. It should be replaced by a modern counterpart for greater clarity and reliability (by automated means if possible). Processors are able to detect and report the use of these features.

4.4 Non-standard extensions

The Fortran standard defines a set of intrinsic procedures and intrinsic modules, and allows a processor to extend this set with further procedures and modules. A program that uses an intrinsic procedure or module not defined by the standard is not standard-conforming. A program that uses an entity not defined by the

standard from a module defined by the standard is not standard-conforming. Use of intrinsic procedures or modules not defined by the standard should be avoided. Processors are able to detect and report the use of intrinsic procedures or modules not defined by the standard.

The Fortran standard does not completely specify the effects of programs in some situations, but rather allows the processor to employ any of several alternatives. These alternatives are called *processor dependencies* and are summarized in Annex A.2 of the standard. The programmer should not rely for program correctness on a particular alternative being chosen by a processor. In general for real and complex entities, the representation of quantities, the results of operations, and the results of calculations performed by intrinsic procedures are all processor-dependent approximations of their respective exact mathematical equivalent.

4.5 Conformance to the standard

Formatted: Heading 3

Although strenuous efforts have been made, and are ongoing, to ensure that the Fortran standard provides an interpretation for all programs that conform to it, circumstances occasionally arise where the standard fails to do so. If the standard fails to provide an interpretation for a program, the program is not standard-conforming.

Processors are required to provide a mode that detects deviation from the standard so far as can be determined from syntax rules and constraints during translation only, and not during execution of a program. Many processors offer runtime checks and debugging aids. For example, most processors support options to report when, during execution, an array subscript is found to be out-of-bounds in an array reference.

Generally, the Fortran standard is written as specifying what a correct program produces as output, and not how such output is actually produced. That is, the standard specifies that a program executes *as if* certain actions occur in a certain order, but not that such actions actually occur. A means other than those specified by Fortran (for example, a debugger) might be able to detect such particulars.

4.6 Numeric model

Formatted: Heading 3

The values of numeric data objects are described in terms of a bit model, an integer model, and a floating-point model. Inquiry intrinsic procedures return values that describe the model rather than any particular hardware. The Fortran standard places minimal constraints on the representation of entities of type character and type logical.

4.7 Interoperability

Formatted: Heading 3

Interoperability of Fortran program units with program units written in other languages is defined in terms of a *companion processor*. A Fortran processor is its own companion processor, and might have other companion processors as well. The interoperation of Fortran program units is defined as if the companion processor is defined by the C programming language.

4.8 Parallelism

4.8.1 Images and coarrays

Formatted: Font: +Headings (Cambria), 12 pt, Bold

Fortran is an inherently parallel programming language, with program execution consisting of one or more asynchronously executing replications, called *images*, of the program. The standard makes no requirements of how many images exist for any program, nor of the mechanism of inter-image communication. Inquiry intrinsic procedures are defined to allow a program to detect the number of images in use, and which replication a particular image represents. Synchronization statements are defined to allow a program to synchronize its images; the `sync all` statement provides a barrier for all images and the `sync images` statement provides a barrier for specified images. Statements executed on one image ahead of its execution of an `event post` statement precede statements executed on another image after its execution of a corresponding `event wait` statement. The `critical` construct defines a scope in which only one image at a time is permitted to execute.

A data object can be declared as a *coarray* which allows it to be accessed from another image by using *cosubscripts* in square brackets to indicate the image.

4.8.2 Locks

Formatted: Font: +Headings (Cambria), 12 pt, Bold

The `lock` and `unlock` statements provide another mechanism for ensuring that data on an image are accessed by one image at a time. A lock is a scalar variable of the derived type `lock_type` that is defined in the intrinsic module `iso_fortran_env`. A lock must be a coarray or a subobject of a coarray. It has one of two states: *locked* and *unlocked*. The only way to change the value of a lock is by executing a `lock` or `unlock` statement. If a lock variable is locked, it can be unlocked only by the image that locked it. If a `lock` statement is executed for a lock variable that is locked by another image, the image normally waits for the lock to be unlocked by that image but there is an option to continue execution in this case. An error condition occurs for a `lock` statement if the lock variable is already locked by the executing image, and for an `unlock` statement if the lock variable is not already locked by the executing image. Here is a simple example of the use of a lock:

```
lock (stack_lock[p])
   stack_size[p] = stack_size[p] + 1
   stack(stack_size[p])[p] = job
unlock (stack_lock[p])
```

Here stack_lock is a coarray and p is a local scalar denoting an image. Several images may execute this code at the same time but no two can be altering data on the same image `p` at the same time. The same data protection could be achieved with a `critical` construct:

```
critical
   stack_size[p] = stack_size[p] + 1
   stack(stack_size[p])[p] = job
end critical
```

but this would prevent several images executing the code at the same time for different values of `p`.

4.8.3 Teams

Teams are sets of images; a team is expected to execute independently of other teams. The set of all images forms the *initial team*. The `form team` statement subdivides a team into a set of new teams. The `change team` construct defines a scope in which the new teams execute.

4.8.4 Segments

Any statement that implies ordering between the execution of statements on different images is known as an *image control statement*. On each image, the set of statements executed between two image control statements is known as a *segment*. The orderings imposed by the image control statments imply a partial ordering of all the segments on all the images. Unless a coarray is atomic (next paragraph), if its value (or of part of it) is altered in a segment, it must not be referenced in another segment unless the two segments are ordered.

There is an exception for the segment ordering rule for integers of kind `atomic_int_kind` and logicals of kind `atomic_logical_kind`. These may be referenced and defined in unordered segments by intrinsic subroutines including `atomic_define`, `atomic_ref`, and `atomic_or`. The system insures that for each variable all such actions occur sequentially. The execution of a `sync memory` statement defines a boundary on an image between two segments, each of which can be ordered in some user-defined way with respect to segments on other images. This can be done, for example, by applying the intrinsic subroutines `atomic_define` and `atomic_ref` to an atomic variable. Atomic variables are not volatile by the Fortran language rules.

4.8.5 Asynchronous variables

Another exception for the segment ordering rule is that a variable may be declared as `asynchronous`. This indicates that the variable might be referenced or defined by non-Fortran procedures. It is initiated by execution of a communication initiation procedure and completed by execution of a corresponding communication completion procedure. The programmer is responsible for ensuring that the variable:

a. is not referenced between execution of an input communication initiation procedure and execution of the corresponding communication completion procedure; and
b. is not defined between execution of an output communication initiation procedure and execution of the corresponding communication completion procedure.

`asynchronous` is used both for I/O of large blocks of data and for interoperating with parallel-processing packages such as MPI that have procedures for nonblocking transfer of data from one process to another.

4.8.6 Asynchronous variables

A further exception for the segment ordering rule is that a variable may be declared as `volatile`. This indicates to the compiler that, at any time, the variable might be changed and/or examined from outside the Fortran program. The feature needs to be used with care. If two processes access the variable at the same time, an inconsistent value might be obtained.

Formatted: Font: (Default) +Headings (Cambria), 12 pt, Bold, Not Expanded by / Condensed by

Deleted: ¶

Commented [SM1]: More text from JR

Commented [SM2]: Further text from JR

4.8.7 Collective subroutines

There are several intrinsic subroutines that are *collective* in the sense that the images of the current team collaborate to perform an action, such as summation.

4.8.8 Image failure

It is optional for a Fortran system to support continued execution in the presence of failed images. If an image is regarded by the system as failed, it remains failed until execution terminates. The constant `stat_failed_image` in the intrinsic module `iso_fortran_env` is positive if failed image handling is supported and negative otherwise. If it is positive, it is used for the value of a `stat=` specifier or `stat` argument if a failed image is involved in an image control statement, a reference to an object with cosubscripts, or an invocation of a collective subroutine or atomic subroutine, and no other error condition occurs. The intrinsic function `image_status` provides a test for the failure of a specified image, and the intrinsic function `failed_images` returns an array of image indices of failed images in the current team.

4.8.9 Do concurrent

Another concurrency mechanism provided by Fortran is the `do concurrent` construct. It permits concurrent execution of the iterations of a loop within the execution of a single image. It does not give the user visibility or control of separate threads of execution performing the operations. By using this construct, the programmer asserts that there are no interdependencies between loop iterations. The language processor is responsible for organizing the use of threads or other mechanisms such as pipelining or the use of GPUs.

5 General guidance for Fortran

In addition to the top 20 generic programming rules from ISO IEC 24772-1 clause 5.2, additional rules from this section apply specifically to the Fortran programming language. The recommendations of this clause are restatements of recommendations from clause 6, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

Every guidance provided in this section, and in the corresponding Part section, is supported material in Clause 6 of this document, as well as other important recommendations.

What do we do with generic rules that do not apply to this Part?

What guidance do we give when the generic rule is highly qualified here?

Number	Recommended avoidance mechanism	References
1.	Ensure that processor reports non-standard forms and relationships.	
1	Never use implicit typing. Always declare all variables. Use implicit none to enforce this.	

Formatted: Font: +Headings (Cambria), 12 pt, Bold

Formatted: Font: +Headings (Cambria), 12 pt, Bold

Formatted: Font: +Headings (Cambria), 12 pt

Formatted: Font: (Default) +Headings (Cambria), 12 pt, English (US)

Deleted: Fortran is an inherently parallel programming language, with program execution consisting of one or more asynchronously executing replications, called *images*, of the program. The standard makes no requirements of how many images exist for any program, nor of the mechanism of inter-image communication. Inquiry intrinsic procedures are defined to allow a program to detect the number of images in use, and which replication a particular image represents. Synchronization statements are defined to allow a program to synchronize its images. Within an image, many statements involving arrays are specifically designed to allow efficient vector instructions. Several constructs for iteration are specifically designed to allow parallel execution.¶
Fortran is the oldest international standard programming language with the first Fortran processors appearing over fifty years ago. During half a century of computing, computing technology has changed immensely, and Fortran has evolved via several revisions of the standard. Also, during half a century of computing and in response to customer demand, some popular processors supported extensions. There remains a substantial body of Fortran code that is written to previous versions of the standard or with extensions to previous versions, and before modern techniques of software development came into widespread use. The process of revising the standard has been done carefully with a goal of protecting applications programmers' investments in older codes. Very few features were deleted from older revisions of the standard; those that were deleted were little used, or redundant with a superior alternative, or error-prone with a safer alternative. Many modern processors generally continue to support deleted features from older revisions of the Fortran standard, and even some extensions from older processors, and do so with the intention of reproducing the original semantics. Also, there exist automatic means of replacing at least some archaic features with modern alternatives. Even with automatic assistance, there might be reluctance to change existing software due to its having proven itself through usage on a wider variety of hardware than is in general use at present, or due to issues of regulation or certification. The decision to modernize trusted software is made cognizant of many factors, including the availability of resources to do so and the perceived benefits. This document does not attempt to specify criteria for modernizing trusted old code.

2	Use explicit conversion intrinsics for the conversion of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.	
3	Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used. Similarly, use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.	
4	Use whole array assignment, operations, and bounds inquiry intrinsics where possible.	
5	Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.	
6	Use default initialization in the declarations of pointer components.	
7	Specify pure (or elemental) for procedures where possible for greater clarity of the programmer's intentions.	
8	Code a status variable for all statements that support one, and examine its value prior to continuing execution for faults that cause termination, provide a message to users of the program, perhaps with the help of the error message	

	generated by the statement whose execution generated the error.	
9	Avoid the use of common and equivalence. Use modules instead of common to share data. Use allocatable data instead of equivalence.	
10	Supply an explicit interface to specify the external attribute for all external procedures invoked.	

6 Specific Guidance for Fortran

6.1 General

What about static analysis tools for Fortran? This document says nothing about static analysis other than the compiler.

This clause contains specific advice for Fortran about the possible presence of vulnerabilities as described in 24772-1 and provides specific guidance on how to avoid them in Fortran program code. This section mirrors 24772-1 clause 6. For example, the vulnerability "Type System [IHN]" that is found in 6.2 of 24772-1, is addressed with Fortran-specific guidance in clause 6.2 in this document.

6.2 Type System [IHN]

6.2.1 Applicability to language

The Fortran type system is a strong type system consisting of the data type and type parameters. A type parameter is an integer value that specifies a parameterization of the type; a derived type (defined by the user) need not have any type parameters. Objects of the same type that differ in the value of their type parameter(s) might differ in representation, and therefore in the limits of the values they can represent. For many purposes for which other languages use type, Fortran uses the type, type parameters, and rank of a data object. A conforming processor supports at least two kinds of type real and a complex kind corresponding to each supported real kind. Double precision real is required to provide more digits of decimal precision than default real. A conforming processor supports at least one integer kind with a range of 10^{18} or greater.

The compatible types in Fortran are the numeric types: integer, real, and complex. No coercion exists between type logical and any other type, nor between type character and any other type. Among the numeric types, coercion might result in a loss of information or an undetected failure to conform to the standard. For example, if a double-precision real is assigned to a single-precision real, round-off is likely; and if an integer operation results in a value outside the supported range, the program is not conforming. This might not be detected. Likewise, assigning a value to an integer variable whose range does not include the value, renders the program not conforming.

An example of coercion in Fortran is (assuming `rkp` names a suitable real kind parameter):

```
real(kind=rkp) :: a
integer :: i
a = a + i
```

which is automatically treated as if it were:

```
a = a + real(i, kind=rkp)
```

Objects of derived types are considered to have the same type when their type definitions are from the same original text (which can be made available to other program units by module use). Sequence types and bind(c) types represent a narrow exception to this rule. Sequence types are less commonly used because they are less convenient to use, cannot be extended, and cannot interoperate with types defined by a companion processor. Bind(c) types are, in general, only used to interoperate with types defined by a companion processor; they also cannot be extended.

A derived type can have type parameters and these parameters can be applied to the derived type's components. Default assignment of variables of the same derived type is component-wise. Default assignment can be overridden by an explicitly coded assignment procedure. For derived-type objects, type changing assignments and conversion procedures are required to be explicitly coded by the programmer. Other than default assignment, each operation on a derived type is defined by a procedure. These procedures can contain any necessary checks and coercions.

In addition to the losses mentioned in Clause 6 of ISO/IEC 24772-1, intrinsic assignment of a complex entity to a noncomplex variable only assigns the real part.

Intrinsic functions can be used in constant expressions that compute desired kind type parameter values. Also, the intrinsic module `iso_fortran_env` supplies named constants suitable for kind type parameters.

6.2.2 Guidance to language users

- Use kind values based on the needed range for integer types via the `selected_int_kind` intrinsic procedure, and based on the range and precision needed for real and complex types via the `selected_real_kind` intrinsic procedure.
- Use explicit conversion intrinsics for conversions of values of intrinsic types, even when the conversion is within one type and is only a change of kind. Doing so alerts the maintenance programmer to the fact of the conversion, and that it is intentional.
- Use inquiry intrinsic procedures to learn the limits of a variable's representation and thereby take care to avoid exceeding those limits.
- Use derived types to avoid implicit conversions.
- Use compiler options when available to detect during execution when a significant loss of information occurs.
- Use compiler options when available to detect during execution when an integer value overflows.

6.3 Bit Representation [STR]

6.3.1 Applicability to language

The vulnerability associated with the difficulty of bit-oriented manipulations as described in ISO/IEC 24772-1 clause 6.3.1 applies to Fortran but is mitigated because all bit operations can be performed by referencing intrinsic procedures.

The vulnerability associated with endianness does not apply to Fortran, since Fortran defines bit positions by a *bit model* described in Subclause 16.3 of the standard. Care should be taken to understand the mapping between an external definition of the bits (for example, a control register) and the bit model. The programmer can rely on the bit model, which depends only on the number of bits in each integer datum and not on how the implementation interprets these bits as an integer value.

Fortran allows constants to be defined by binary, octal, or hexadecimal digits, collectively called BOZ constants. BOZ constants can only be used to initialize variables and as arguments to intrinsic functions that perform bit operations or convert to the numeric types.

These values can be assigned to named constants thereby providing a name for a mask. Such constants may be placed in an integer aligned to the right using the int intrinsic, for example,

```
i = int(o'716',kind(i)).
```

If the size of I is 8 bits, then the final value would be o'316', not o'716', as the user intended. One can ensure that the integer is long enough by using the bit_size intrinsic, for example

```
if ( bit_size (i) >= 9 ) then
   i = int(o'716',kind(i))
else
    .....
```

A further complication arises if a BOZ constant is interpreted as a real number since real numbers can have a number of representations.

Derived types in Fortran can be used to isolate bit operations within the type definition and thus to prevent its direct use by the user of the derived type.

Fortran provides access to individual bits within an integer by bit manipulation intrinsic procedures. Of particular use, shift procedures are provided to manipulate a bit field held in more than a single integer.

The bit model does not provide a bit representation for negative integer values.

(Aside – Fortran does not define the mapping of sequential unformatted files and this can lead to a vulnerability when using such files between programs compiled with different language processors, or even between different versions of the same processor. This could be placed in in clause 7)

6.3.2 Guidance to language users

- Use the language-provided intrinsics whenever bit manipulations are necessary, especially those that

Formatted: Normal

Deleted: Fortran defines bit positions by a *bit model* described in Subclause 13.3 of the standard. Care should be taken to understand the mapping between an external definition of the bits (for example, a control register) and the bit model. The programmer can rely on the bit model regardless of endian, or other hardware peculiarities.¶
Fortran allows constants to be defined by binary, octal, or hexadecimal digits, collectively called *BOZ constants*. These values can be assigned to named constants thereby providing a name for a mask.¶
Fortran provides access to individual bits within a storage unit by bit manipulation intrinsic procedures. Of particular use, double-word shift procedures are provided to extract bit fields crossing storage unit boundaries.¶
The bit model does not provide an interpretation for negative integer values. There are distinct shift intrinsic procedures to interpret, or not interpret, the left-most bit as the sign bit.¶

occupy more than one integer.

- Use the intrinsic procedure `bit_size` to determine the size of the bit model supported by the kind of integer in use.
- Be aware that the Fortran standard uses the term “left-most” to refer to the highest-order bit, and the term “left” to mean towards the highest-order bit (as in `shiftl`) .
- Do not use compiler extensions that allow variables of logical type to hold bit string values, because the results may vary between implementations.
- Avoid compiler extensions that accept BOZ constants in non-standard usage.
- Encapsulate bit strings inside derived types to exclude numeric operations on them.

6.4 Floating-point Arithmetic [PLF]

6.4.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1 clause 6.4 is applicable to Fortran. Most language processors support parts of the ISO/IEC/IEEE 60559:2011 standard and facilities are provided for the programmer to detect the extent of conformance.

The rounding mode in effect during translation might differ from the rounding mode in effect during execution; the rounding mode can be changed during execution under program control. A separate rounding mode is provided for input/output formatting conversions; this rounding mode can also be changed during execution.

Fortran provides intrinsic procedures to give values describing the limits of any representation method in use, to provide access to the parts of a floating-point quantity, and to set the parts.

6.4.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.4.5.
- Use procedures from a trusted library to perform calculations where floating-point accuracy is needed. Understand the use of the library procedures and test the diagnostic status values returned to ensure the calculation proceeds as expected.
- Avoid creating a logical value from a test for equality or inequality between two floating-point expressions. Use compiler options where available to detect such usage.
- Do not use floating-point variables as loop indices, a deleted feature; use integer variables instead. A floating-point value can be computed from the integer loop variable as needed.
- Use intrinsic inquiry procedures to determine the limits of the representation in use when needed.
- Avoid the use of bit operations to get or to set the parts of a floating point quantity. Use intrinsic procedures to provide the functionality when needed.
- Use the intrinsic module procedures to determine the limits of the processor’s conformance to IEEE 754, and to determine the limits of the representation in use, where the IEEE intrinsic modules and the IEEE real kinds are in use.
- Use the intrinsic module procedures to detect and control the available rounding modes and exception flags, where the IEEE intrinsic modules are in use.

Deleted: Create objects of

Commented [SM5]: This should be justified in 6.3.1

Deleted: <#>hide use of bit intrinsic procedures within defined operators and to separate those objects subject to arithmetic operations from those objects subject to bit operations.¶
especially those that occupy more than one storage unit. Choose shift intrinsic procedures cognizant of the need to affect the sign bit, or not.¶
Use the intrinsic procedure `bit_size` to determine the size of the bit model supported by the kind of integer in use.¶
Be aware that the Fortran standard uses the term “left-most” to refer to the highest-order bit, and the term “left” to mean towards (as in `shiftl`), or from (as in `maskl`), the highest-order bit.¶
Be aware that the Fortran standard uses the term “right-most” to refer to the lowest-order bit, and the term “right” to mean towards (as in `shiftr`), or from (as in `maskr`), the lowest-order bit.¶
Avoid bit constants made by adding integer powers of two in favour of those created by the bit intrinsic procedures or encoded by BOZ constants.¶
Use bit intrinsic procedures to operate on individual bits and bit fields, ¶
Create objects of derived type to hide use of bit intrinsic procedures within defined operators and to separate those objects subject to arithmetic operations from those objects subject to bit operations.

Commented [SGM6]: Confirm that the FP issues updated in -1 at the June 2015 meeting are reflected here.

Deleted: Fortran supports floating-point data. Furthermore, m

6.5 Enumerator Issues [CCB]

6.5.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1 clause 6.5 is applicable to Fortran since Fortran provides enumeration values for interoperation with C programs that use C enums. Their use is expected most often to occur when a C enum appears in the function prototype whose interoperation requires a Fortran interface.

Vulnerabilities associated with indexing arrays with enumeration types do not apply to Fortran since enum literals are simply named integer constants. The Fortran variables to be assigned the enumeration values are of type integer and the correct kind to interoperate with C variables of C type enum.

Deleted: ¶
The Fortran enumeration values are integer constants of the correct kind to interoperate with the corresponding C enum. The Fortran variables to be assigned the enumeration values are of type integer and the correct kind to interoperate with C variables of C type enum.

6.5.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.5.5.
- Use enumeration values in Fortran only when interoperating with C procedures that have enumerations as formal parameters and/or return enumeration values as function results.
- Ensure the interoperability of the C and Fortran definitions of every enum type used.
- Ensure that the correct companion processor has been identified, including any companion processor options that affect enum definitions.
- Do not use variables assigned enumeration values in arithmetic operations, or to receive the results of arithmetic operations if subsequent use will be as an enumerator.

6.6 Conversion Errors [FLC]

6.6.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1 clause 6.6 is applicable to Fortran .

Fortran processors are required to support two kinds of type real and are required to support a complex kind for every real kind supported. Fortran processors are required to support at least one integer kind with a range of 10^{18} or greater and most processors support at least one integer kind with a smaller range.

Automatic conversion among these types is allowed, with the associated vulnerabilities documented in ISO/IEC 24772-1 clause 6.6.

Fortran does not provide intrinsic assignment between unrelated types for conforming programs. The programmer can create explicit conversion routines between unrelated types.

Equivalence between objects of character and integer types as well as between objects of logical and numeric types is obsolescent, and will be diagnosed by compilers with a warning when requested.

Conversion is automatic between default and ASCII character kinds, and from these kinds to ISO/IEC 10646 kind.

Deleted:

Deleted: s

Fortran uses IO statements for conversion between character and numeric types. If the field width is insufficient on output then asterisks are used. If a value on input cannot be represented, the outcome is processor dependent but an error condition should be expected. If the Fortran processor detects an error on input or output, then the IOSTAT variable is set to a non-zero value.

Deleted: 7

Fortran provides the capability to identify different units of measure through the use of distinct derived types. For example, the derived types

```
type centigrade
   real :: temp
end type
type fahrenheit
   real :: temp
end type
```

might be used for Celcius and Fahrenheit temperatures and the function

```
type (centigrade) function FtoC(t)
   type (Fahrenheit) :: t
   FtoC%temp = (t%temp-32.0)/1.8
end function
```

for conversion from Fahrenheit to Centigrade.

The following code would not conform to the standard and is diagnosed by many compilers.

```
type (fahrenheit) :: f
type (centigrade) :: c
c = f                    ! Non-conforming
```

6.6.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.6.5
- Use the kind selection intrinsic procedures to select sizes of variables supporting the required operations and values.
- Use a temporary variable with a large range to read a value from an untrusted source so that the value can be checked against the limits provided by the inquiry intrinsics for the type and kind of the variable to be used.
- Use a temporary variable with a large range to hold the value of an expression before assigning it to a variable of a type and kind that has a smaller numeric range to ensure that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
- Use derived types and put checks in the applicable defined assignment procedures.
- Use static analysis or compiler features to identify conversions that can lose or corrupt information.
- Use compiler options when available to detect and report during execution when a loss or corruption of information occurs.
- Consider using simple derived types to hold numeric values that can represent different unit systems (such as radians vs degrees) and provide explicit conversion functions as needed.
- Include an IOSTAT variable in each IO statement and check its value to ensure no errors occurred.

Deleted:
Deleted: (More)
Formatted: Space After: 10 pt, Line spacing: Multiple 1.15 li
Formatted: Font: (Default) Courier New, 10.5 pt
Formatted: Font: (Default) +Body (Calibri), 11 pt, English (US)
Formatted: Font: (Default) Courier New, 10.5 pt
Formatted: Font: (Default) Courier New, 10.5 pt
Formatted: Font: (Default) Courier New, 10.5 pt
Formatted: Font: (Default) Courier New, 10.5 pt
Formatted: Font: (Default) Courier New, 10.5 pt
Commented [SM9]: Not permitted, or illegal and diagnosed by the language processor?
Deleted: ¶
Formatted: Font: (Default) Courier New, 10.5 pt
Deleted: ¶
Deleted: ¶
When assigning an expression of one type and kind to a variable of a type and kind that might have a smaller numeric range, check that the value of the expression is within the allowed range for the variable. Use the inquiry intrinsics to supply the extreme values allowed for the variable.
Deleted: whether
Deleted: will
Deleted: significant
Deleted: Use compiler options when available to detect during execution when an integer value overflows.
Formatted: English (UK)

6.7 String Termination [CJM]

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.7 is not applicable to Fortran since strings are not terminated by a special character and the string length is maintained by the implementation.

Deleted: TR

Deleted: .

Deleted: ¶

6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]

6.8.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.8 is applicable to Fortran as follows. A Fortran program might be affected by this vulnerability in two situations. The first is that an array subscript could be outside its bounds, and the second is that a character substring index could be outside its length. The Fortran standard requires that each array subscript be separately within its bounds, not simply that the resulting offset be within the array as a whole.

Fortran does not mandate array subscript checking to verify in-bounds array references, nor character substring index checking to verify in-bounds substring references.

The Fortran standard requires that array shapes conform for whole array assignments and operations where the left-hand side is not an allocatable object. However, Fortran does not mandate that array shapes be checked during whole-array assignments and operations.

When a whole-array assignment occurs to define an allocatable array, the allocatable array is resized, if needed, to the correct size. When a whole character assignment occurs to define an allocatable character, the allocatable character is resized, if needed, to the correct size.

When a character assignment defines a non-allocatable character variable and a length mismatch occurs, the assignment has a blank-fill (if the value is too short) or truncate (if the value is too long) semantic; this is also true for input. If this happens for an allocatable character variable, the variable defined is resized, if needed, to the correct size; but this does not happen for input.

Deleted: .

Deleted: Otherwise

Commented [SM10]: AI – what happens for IO? – For output, truncates or blank fills

Most implementations include an optional facility for bounds checking. These are likely to be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, or because the local declaration of the bounds might be inconsistent with those of the actual argument. It is therefore preferable to use an assumed-shape array as a procedure dummy argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which can be used to obtain the bounds of arrays where such information is available. Fortran also provides character length intrinsic inquiry intrinsics so the length of character entities can be reliably found.

6.8.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.8.5
- Ensure that consistent bounds information about each array is available throughout a program.

- Enable bounds checking throughout development of a code. Disable bounds checking during production runs only for program units that are critical for performance.
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local arrays.
- Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Use allocatable character variables where assignment of strings of varying sizes is expected so the left-hand side character variable is reallocated as needed.
- Use intrinsic assignment for the whole character variable rather than looping over substrings to assign data to statically-sized character variables so the truncate-or-blank-fill semantic protects against storing outside the assigned variable.

Deleted: ¶

Deleted: or allocatable¶ dummy arguments

Deleted: automatic

Deleted: widely-

Deleted: explicit loops

Deleted: ¶

6.9 Unchecked Array Indexing [XYZ]

6.9.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.9 is applicable to Fortran.

A Fortran program might be affected by this vulnerability in the situation an array subscript could be outside its bounds. The Fortran standard requires that each array subscript be separately within its bounds, not simply that the resulting offset be within the array as a whole, but implementations are not required to diagnose this.

Fortran requires that the value assigned to a non-allocatable array conforms to the shape of the target. In an assignment to an allocatable array, the allocatable array is reallocated if needed to conform to the shape of the source. In an assignment to an allocatable character variable, the variable is reallocated, if needed, to the correct length.

Most processors include an optional facility for bounds checking. These are likely to be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, or because the local declaration of the bounds might be inconsistent with those of the actual argument. It is therefore preferable to use an assumed-shape array as a procedure argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.

Fortran provides a set of array bounds intrinsic inquiry procedures which can obtain the bounds of arrays where such information is available.

6.9.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.9.5.
- Include sanity checks to ensure the validity of any values used as index variables.
- Ensure that consistent bounds information about each array is available throughout a program.
- Enable bounds checking, when available, throughout development of a code. Disable bounds checking

Commented [SM12]: Should the first comment not be to follow the advice of Part 1?

during production runs only for program units that are critical for performance.

- Use whole array assignment, operations, and bounds inquiry intrinsics where possible.
- Obtain array bounds from array inquiry intrinsic procedures wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable arrays as procedure dummy arguments to ensure that array shape information is passed to all procedures where needed, and can be used to dimension local arrays.
- Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.
- Declare the lower bound of each array extent to fit the problem, thus minimizing the use of subscript arithmetic.

6.10 Unchecked Array Copying [XYW]

6.10.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1 clause 6.10 is applicable to Fortran. See clause 6.9.

6.10.2 Guidance to language users

Follow the guidance of clause 6.9.2.

6.11 Pointer Type Conversions [HFC]

6.11.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.11 is applicable to Fortran in the following cases: in the context of polymorphic pointers; in the use of `c_ptr` and `c_funptr;` and in the use of implicit interfaces for procedure pointers and dummy procedure arguments. All other pointer conversions are strongly typed.

A non-polymorphic pointer is declared with a type and can be associated only with an object of its type. A polymorphic pointer that is not unlimited polymorphic is declared with a type and can be associated only with an object of its type or an extension of its type. An unlimited polymorphic pointer can be used to reference its target only by using a type with which the type of its target is compatible in a select type construct. A procedure pointer can only be associated with a procedure target. These restrictions are enforced during compilation.

A procedure pointer with an implicit interface can be associated with a procedure target that has a different implicit interface, with the risk of passing incorrect number or types of parameters. Similarly, a dummy procedure can be associated with an actual procedure that has a different interface, with the risk of passing incorrect number or types of parameters. Either case can result in arbitrary failures.

When an unlimited polymorphic pointer has a target of a *sequence type* or an interoperable derived type, a type-breaking cast can occur.

A pointer appearing as an argument to the intrinsic module procedure `c_loc` effectively has its type changed to the intrinsic type `c_ptr`, which can be recast to any type.

A procedure pointer appearing as an argument to the intrinsic module procedure `c_funloc` effectively has its type changed to the intrinsic type `c_funptr`, which can be recast to any procedure pointer.

Deleted: automatic

Deleted: s

Deleted: <#>Arrays can be declared in modules which makes their bounds information available wherever the array is available. ¶

Deleted: Fortran provides array assignment, so this vulnerability applies.¶
An array assignment with shape disagreement is prohibited, but the standard does not require the processor to check for this.¶
When a whole-array assignment occurs to define a non-coarray allocatable array, the non-coarray allocatable array is resized, if needed, to the correct size. When a whole character assignment occurs to define a non-coarray allocatable character, the non-coarray allocatable character is resized, if needed.¶
Most implementations include an optional facility for bounds checking. These are likely to be incomplete for a dummy argument that is an explicit-shape or assumed-size array because of passing only the address of such an object, and/or the reliance on local declaration of the bounds. It is therefore preferable to use an assumed-shape or allocatable array as a procedure dummy argument. The performance of operations involving assumed-shape arrays is improved by the use of the `contiguous` attribute.¶
Fortran provides a set of array bounds intrinsic inquiry procedures which can be used to obtain the bounds of arrays where such information is available.

Deleted: Ensure that consistent bounds information about each array is available throughout a program.¶
Enable bounds checking throughout development of a code.
Disable bounds checking during production runs only for program units that are critical for performance.¶
Use whole array assignment, operations, and bounds inquiry intrinsics where possible.¶
Obtain array bounds from array inquiry intrinsics wherever needed. Use explicit interfaces and assumed-shape arrays or allocatable array as procedure dummy arguments to ensure that array bounds information is passed to all procedures where needed, including dummy arguments and automatic arrays.¶
Use allocatable arrays where arrays operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed.

Deleted: This vulnerability is not applicable to Fortran

Deleted: in most circumstances.

Moved (insertion) [1]

Deleted: When an unlimited polymorphic pointer has a target of a *sequence type* or an interoperable derived type, a type-break ... [1]

Deleted:

Deleted:

Deleted: An unlimited polymorphic pointer can also be assigned to a sequence type or bind(c) type pointer; this is unsafe, and ca ... [2]

Deleted: might

Moved up [1]: When an unlimited polymorphic pointer has a target of a sequence type or an interoperable derived type, a type-

Deleted: c_

Deleted: f_pointer

Deleted: c_p

Deleted: tr

Deleted: .

Deleted:

6.11.2 Guidance to language users

- Avoid implicit interfaces; use explicit interfaces instead.
- Avoid the use of C-style pointers, unless necessary to interface with C programs.
- Avoid sequence types as target types of unlimited polymorphic pointers.

6.12 Pointer Arithmetic [RVG]

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.12 is not applicable to Fortran since there is no mechanism for pointer arithmetic in Fortran.

6.13 Null Pointer Dereference [XYH]

6.13.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.13 is applicable to Fortran.

A Fortran pointer by default is initially undefined and not nullified. A pointer is nullified by pointer assigning to a null pointer, assigning to the result of the `null` intrinsic procedure, or by the `nullify` statement.

The Fortran intrinsic procedure `associated` determines whether a pointer that is not undefined has a valid target, or whether it is associated with a particular target.

Some processors include an optional facility for pointer checking.

6.13.2 Guidance to language users

- Use `allocatable` instead of `pointer` when possible.
- Use static analysis tools and compiler options where available to enable pointer checking during development of a code throughout.
- Use the `associated` intrinsic procedure before referencing a target through the pointer if there is any possibility of it being disassociated.
- Associate pointers before referencing them.
- Use default initialization in the declarations of pointer components.
- Use initialization in the declarations of all pointers that have the `save` attribute.

6.14 Dangling Reference to Heap [XYK]

6.14.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.14 is applicable to Fortran because it has pointers, and separate `allocate` and `deallocate` statements for them.

6.14.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.14.5
- Use allocatable objects in preference to pointer objects whenever the facilities of allocatable objects are sufficient.

Commented [SM13]: Research a guidance to avoid sequence types.

Deleted: Further casts could be made if the pointer is processed by procedures written in a language other than Fortran.¶

Commented [SM14]: What about the guidance of 24772-1 clause

Deleted: <#>Avoid C interoperability features in programs that do not interoperate with other languages.¶

Deleted: <#>Avoid use of sequence types.

Deleted: A Fortran pointer should not be referenced when its status is disassociated or nullified.¶

Deleted: only

Deleted: when it is done explicitly, either

Commented [SM15]: What about the guidance of 24772-1 clause 6.13.5?

Commented [SM16]: Ended here. Potentially, rationalize the guidance to the applicability.

Deleted: is

- Use compiler options where available to detect dangling references.
- Use compiler options where available to enable pointer checking throughout development of a code. Disable pointer checking during production runs only for program units that are critical for performance.
- Do not pointer-assign a pointer to a target if the pointer might have a longer lifetime than the target or the target attribute of the target. Check actual arguments that are argument associated with dummy arguments that are given the target attribute within the referenced procedure.
- Check for successful deallocation when deallocating a pointer by using the `stat=` specifier.

6.15 Arithmetic Wrap-around Error [FIF]

6.15.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.15 is applicable to Fortran . This vulnerability is applicable to Fortran for integer values. Some processors have an option to detect this vulnerability at run time.

6.15.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.15.5
- Use the intrinsic procedure `selected_int_kind` to select an integer kind value that will be adequate for all anticipated needs.
- Use compiler options where available to detect during execution when an integer value overflows.

6.16 Using Shift Operations for Multiplication and Division [PIK]

6.16.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.16 is applicable to Fortran. Fortran provides bit manipulation through intrinsic procedures that operate on integer variables. Specifically, both shifts that replicate the left-most bit and shifts that do not are provided as intrinsic procedures with integer operands.

6.16.2 Guidance to language users

- Do not use shift intrinsics where integer multiplication or division is intended.

Deleted: <#>Follow the guidance of ISO/IEC 24772-1:2019 clause 6.16.5¶
Separate integer variables into those on which bit operations are performed and those on which integer arithmetic is performed.¶

6.17 Choice of Clear Names [NAI]

6.17.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.17 is applicable to Fortran. Fortran is a single-case language; upper case and lower case are treated identically by the standard in names.

A name can include underscore characters, except in the initial position. The number of consecutive underscores is significant but might be difficult to see.

When implicit typing is in effect, a misspelling of a name results in a new variable. Implicit typing can be disabled by use of the implicit `none` statement.

Formatted: Font: (Default) Courier New

Fortran has no reserved names. Language keywords are permitted as names.

6.17.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.17.5
- Declare all variables and use `implicit none` to enforce this.
- Do not use consecutive underscores in a name.
- Do not use keywords as names when there is any possibility of confusion.
- Be aware of language rules associated with the case of external names and with the attribute `bind(C)`.

6.18 Dead store [WXQ]

6.18.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.18 is applicable to Fortran.

6.18.2 Guidance to Language Users

Follow the guidance of ISO/IEC 24772-1:2019 clause 6.18.5

6.19 Unused Variable [YZS]

6.19.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1:2019 clause 6.19 is applicable to Fortran. Fortran has separate declaration and use of variables and does not require that all variables declared be used, so this vulnerability applies.

6.19.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.19.5

6.20 Identifier Name Reuse [YOW]

6.20.1 Applicability to language

The vulnerability as specified in ISO/IEC in 24772-1:2019 clause 6.20 is applicable to Fortran. Fortran has several situations where nested scopes occur. These include:

- Module procedures have a nested scope within their module host.
- Internal procedures have a nested scope within their (procedure) host.
- A block construct might have a nested scope within the host scope.
- An array constructor might have a nested scope.

The index variables of some constructs, such as `do concurrent` or array constructor implied do loops, are local to the construct. A select name in an `associate` or `select type` construct is local to the construct.

6.20.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.20.5.
- Do not reuse a name within a nested scope.
- Clearly comment the distinction between similarly-named variables, wherever they occur in nested scopes.
- Be aware of the scoping rules for statement entities and construct entities

6.21 Namespace Issues [BJL]

6.21.1 Applicability to language

The vulnerability specified in 24772-1:2019 clause 6.22 does not apply to Fortran because the import of homographs into a unit results in compilation failure on an attempt to access one of the named items, i.e. the ambiguity is diagnosed. These ambiguities can be resolved by renaming one or both of the homographs on import.

A similar vulnerability exists, however, when implicit typing is used within a scope, and a module is accessed via use association without an *only* list. Specifically, a variable that appears in the local scope but is not explicitly declared, might have a name that is the same as a name that was added to the module after the module was first used. This can cause the declaration, meaning, and the scope of the affected variable to change. See also clause 6.45 “Extra intrinsics”.

6.21.2 Guidance to language users

- Avoid implicit typing. Always declare all variables. Use `implicit none` to enforce this.
- Use a global `private` statement in all modules to require explicit specification of the `public` attribute.
- Use an `only` clause on every `use` statement.
- Use renaming to resolve name collisions.

6.22 Initialization of Variables [LAV]

6.22.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.22 applies to Fortran. The value of a variable that has never been given a value is undefined. It is the programmer’s responsibility to guard against use of uninitialized variables.

Supplying an initialization in the declaration of a local variable, or in a `data` statement, causes the variable to be located in static storage, so later invocations of the unit will see the last stored value from the previous invocation. This can be avoided by using executable statements to initialize local variables.

6.22.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.22.5???

- Favour explicit initialization in executable statements for objects of intrinsic type and default initialization for objects of derived type. When providing default initialization, provide default values for all components.
- Use type value constructors to provide values for all components.
- Use compiler options, where available, to find instances of use of uninitialized variables.
- Use other tools, for example, a debugger or flow analyzer, to detect instances of the use of uninitialized variables.

6.23 Operator Precedence and Associativity [JCW]

6.23.1 Applicability to language

The vulnerability as specified in ISO/IEC 24772-1 clause 6.23 applies to Fortran.

Fortran specifies an order of precedence for operators. The order for the intrinsic operators is well known except among the logical operators `.not.`, `.and.`, `.or.`, `.eqv.`, and `.neqv.`. In addition, any monadic defined operator, and any dyadic defined operator have a position in this order, but these positions are not well known.

6.23.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.23.5.
- Consult the Fortran reference manual or suitable textbooks for definitive information.

6.24 Side-effects and Order of Evaluation [SAM]

6.24.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.22 applies to Fortran. Fortran functions are permitted to have side effects, unless the function is declared to have the `pure` attribute. Within some expressions, the order of invocation of functions is not specified. The standard explicitly requires that evaluating any part of an expression does not change the value of any other part of the expression, but there is no requirement for this to be diagnosed by the processor.

Further, the Fortran standard allows a processor to ignore any part of an expression that is not needed to compute the value of the expression. Processors vary as to how aggressively they take advantage of this permission.

6.24.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.24.5.
- Replace any function with a side effect by a subroutine so that its place in the sequence of computation is certain.
- Assign function values to temporary variables and use the temporary variables in the original expression.
- Declare a function as `pure` whenever possible.

6.25 Likely Incorrect Expression [KOA]

6.25.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.25 applies to Fortran, however Fortran's likely incorrect expressions are not those documented. Some of Fortran's issues arise because processors may extend the language with syntax that conflicts with the standard.

Some processors allow an operator immediately preceding a unary operator, which should be avoided. This can be detected by using processor options to detect violations of the standard. A common mistake is to confuse intrinsic assignment (=) and pointer assignment (=>). Programmers sometimes assume that logical operators can be used on numeric values.

6.25.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.25.5.
- Use an automatic tool to simplify expressions.
- Check for assignment versus pointer assignment carefully when assigning to names having the pointer attribute.
- Enable the compiler's detection of nonconforming code.

6.26 Dead and Deactivated Code [XYQ]

6.26.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.26 applies to Fortran. There is no requirement in the Fortran standard for processors to detect code that cannot be executed. It is entirely the task of the programmer to remove such code.

The developer should justify each case of statements not being executed.

If desirable to preserve older code for documentation (for example, of an older numerical method), the code should be converted to comments. Alternatively, a source code control package can be used to preserve the text of older versions of a program.

6.26.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.26.5.
- Use an editor or other tool that can transform a block of code to comments to do so with dead or deactivated code.
- Use a version control tool to maintain older versions of code when needed to preserve development history.

6.27 Switch Statements and Static Analysis [CLL]

6.27.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.27 applies to Fortran. Fortran has a `select case` construct, and control never flows from one alternative to another.

Fortran has a computed `go to` statement that allows control to flow from one alternative to another, and allows other unexpected flow of control.

The vulnerabilities associated with select-case blocks and enumeration types with "holes" apply to Fortran.

Commented [SM17]: Moved from 6.5 enumeration types

6.27.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.27.5.
- Cover cases that are expected never to occur with a case default clause to ensure that unexpected cases are detected and processed, perhaps emitting an error message.

6.28 Demarcation of Control Flow [EOJ]

6.28.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.28 applies primarily to deprecated constructs of Fortran. Modern Fortran supports block constructs for choice and iteration, which have separate end statements for do, select, and if constructs. Furthermore, these constructs can be named which reduces visual confusion when blocks are nested.

6.28.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.28.5.
- Use the block form of the do-loop, together with cycle and exit statements, rather than the non-block do-loop.
- Use the `if` construct or `select case` construct whenever possible, rather than statements that rely on labels, that is, the arithmetic `if` and `go to` statements.
- Use names on block constructs to provide matching of initial statement and end statement for each construct.

6.29 Loop Control Variables [TEX]

6.29.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.29 does not apply to standard Fortran, however some circumstances arise that are documented here.

A Fortran `do` construct has the trip increment and trip count established when the `do` statement is executed. These do not change during the execution of the loop.

The program is prohibited from changing the value of an iteration variable during execution of the loop. The processor is usually able to detect violation of this rule, but there are situations where this is difficult or requires use of a processor option; for example, an iteration variable might be changed by a procedure that is referenced within the loop.

6.29.2 Guidance to language users

- Ensure that the value of the iteration variable is not changed other than by the loop control mechanism during the execution of a `do` loop.
- Verify that where the iteration variable is an actual argument, it is associated with an `intent(in)` or a `value` dummy argument.

6.30 Off-by-one Error [XZH]

6.30.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.30 applies to Fortran as described below.

Fortran is not very susceptible to this vulnerability because it permits explicit declarations of upper and lower bounds of arrays, which allows bounds that are relevant to the application to be used. For example, latitude can be declared with bounds -90 to 90, while longitude can be declared with bounds -180 to 180. Thus, user-written arithmetic on subscripts can be minimized.

This vulnerability is applicable to a mixed-language program containing both Fortran and C, since arrays in C always have the lower bound 0 while the default in Fortran is 1, and one can reduce the overall complexity in the programmer's mind by declaring the Fortran arrays with lower bounds of zero.

The vulnerability associated with off-by-one errors in loops applies to Fortran. The `lbound` and `ubound` intrinsics provide a safe mechanism for iterating over array subscripts.

Commented [M18]: a) default in Fortran is 1, right? Maybe say so.
b) I think the "explcit arithmetic argument is spurious.
c) off-by-one does not only apply to arrays. It is just as bad in loops (solution: iterators).

Deleted:

6.30.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.30.5.
- Declare array bounds to fit the natural bounds of the problem.
- Declare interoperable (with C) arrays with the lower bound 0.
- Use `lbound` and `ubound` intrinsics to specify loop bounds instead of numeric literals.

Deleted:

Formatted: Font: 10.5 pt

Formatted: Font: (Default) Courier New, 10.5 pt

6.31 Unstructured Programming [EWD]

6.31.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.31 applies to Fortran.

As the first language to be formally standardized, Fortran has older constructs that allow an unstructured programming style to be employed.

These features have been superseded by better methods. The Fortran standard continues to support these archaic forms to allow older programs to function. Some of them are obsolescent, which means that the processor is required to be able to detect and report their usage.

Automatic tools are the preferred method of refactoring unstructured code. Only where automatic tools are unable to do so should refactoring be done manually.

Refactoring efforts should always be thoroughly checked by testing of the new code.

6.31.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.31.5.
- Use the compiler or static analysis tools to detect unstructured programming and the use of old or obsolescent features.
- Use a tool to automatically refactor unstructured code.
- Replace unstructured code manually with modern structured alternatives only where automatic tools are unable to do so.
- Use the compiler or other code analysis tool to detect archaic usage.

6.32 Passing Parameters and Return Values [CSJ]

6.32.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.32 applies to Fortran.

Fortran does not specify the argument passing mechanism, but rather specifies the rules of *argument association*. These rules are generally implemented either by pass-by-reference, by value, by copy-in/copy-out, by descriptor, or by copy-in.

More restrictive rules apply to coarrays and to arrays with the contiguous attribute. Rules for procedures declared to have a C binding follow the rules of C.

Module procedures, intrinsic procedures, and internal procedures have explicit interfaces. An external procedure has an explicit interface only when one is provided by a procedure declaration or interface body. Such an interface body could be generated automatically using a software tool. Explicit interfaces allow processors to check the type, kind, and rank of arguments and result variables of functions.

6.32.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.32.5.
- Specify explicit interfaces by placing procedures in modules where the procedure is to be used in more than one scope, or by using internal procedures where the procedure is to be used in one scope only.
- Specify argument intents to allow further checking of argument usage.
- Specify `pure` (or `elemental`) for procedures where possible for greater clarity of the programmer's intentions.
- Use a compiler or other tools to automatically create explicit interfaces for external procedures.

6.33 Dangling References to Stack Frames [DCM]

6.33.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.33 applies to Fortran when a local target does not have the `save` attribute and the pointer has a lifetime longer than the target. However, the intended functionality is often available with allocatables, which do not suffer from this vulnerability. The Fortran standard

explicitly states that the lifetime of an allocatable function result extends to its use in the expression that invoked the call.

6.33.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.33.5.
- Do not pointer-assign a pointer to a target if the pointer association might have a longer lifetime than the target or the `target` attribute of the target.
- Use allocatable variables in preference to pointers wherever they provide sufficient functionality.

6.34 Subprogram Signature Mismatch [OTR]

6.34.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.34 applies to Fortran.

The Fortran term denoting a procedure's signature is its interface.

The Fortran standard requires that interfaces match, but does not require that the processor diagnoses mismatches. However, processors do check this when the interface is explicit. Some processors can check interfaces if inter-procedural analysis is requested.

Explicit interfaces are provided automatically for intrinsic procedures or when procedures are placed in modules or are internal procedures within other procedures.

6.34.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.34.5.
- Use explicit interfaces, preferably by placing procedures inside a module or another procedure.
- Use a processor or a static analysis tool that check all interfaces, especially if this can be checked during compilation with no execution overhead.
- Use a processor or other tool to create explicit interface bodies for external procedures.

6.35 Recursion [GDL]

6.35.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.35 applies to Fortran since it supports recursion. In Fortran 2018, procedures are recursive by default; the keyword `non_recursive` is required to indicate the opposite. Previous versions provide the `recursive` attribute to permit recursion.

Recursive calculations are attractive in some situations due to their close resemblance to the most compact mathematical formula of the quantity to be computed.

6.35.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.35.????
- Prefer iteration to recursion, unless it can be proved that the depth of recursion can never be large.

6.36 Ignored Error Status and Unhandled Exceptions [OYB]

6.36.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.36 applies to Fortran

Many Fortran statements and some intrinsic procedures return a status value. In most circumstances, status error values that are not requested by the invoking program result in the error termination of the program. Some programmers, however, in order to "keep going" request the status value but do not examine it. This results in a program crash without an explanation when subsequent steps in the program rely upon the previous statements having completed successfully.

Fortran consistently uses a scheme of status values where zero indicates success, a positive value indicates an error, and a negative value indicates some other information.

Other than via the IEEE intrinsic modules, Fortran does not support exception handling.

6.36.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.36.5.
- When the default behaviour of program termination is undesirable, code a status variable for all statements that support one, examine its value prior to continuing execution for faults that cause termination, and take appropriate action.
  Note: Appropriate action may be providing a message to users of the program (perhaps with the help of the error message generated by the statement whose execution generated the error), logging the error, or invoking termination or recovery actions.
- Check and respond to all status values that might be returned by an intrinsic procedure or by a library procedure.

6.37 Type-breaking Reinterpretation of Data [AMV]

6.37.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.37 applies to Fortran only in the context of the `transfer` intrinsic or the deprecated features of `common` and `equivalence`. In particular, standard Fortran does not provide the means to convert between unrelated types.

Transfer permits the unchecked copying from a value to a specified (different) type.

Storage association via common or equivalence statements, or via the transfer intrinsic procedure can cause a type-breaking reinterpretation of data. Type-breaking reinterpretation via common and equivalence is not standard-conforming.

6.37.2 Guidance to language users

- Avoid use of the `transfer` intrinsic unless its use is unavoidable, and then document the use carefully.
- Do not use `common` to share data. Use module variables instead.
- Do not use `equivalence`. If the intent is to save storage space, use allocatable data instead.

- Use compiler options where available to detect violation of the rules for `common` and `equivalence`.

6.38 Deep vs. Shallow Copying [YAN]

6.38.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1 clause 6.38. applies to Fortran. Both deep copy and shallow copy are supported by the language. The operator = performs a one-level deep copy, the operator => performs *pointer assignment.*

Data structures in Fortran that do not contain pointers are completely copied. *Allocatable* components are completely copied, pointer components have only the pointer copied. If the allocatable object has already been allocated but has a different shape or different dynamic type, then the target will be deallocated, reallocated to the shape and dynamic type the source, and the copy is completed; for arrays, the lower bound of the copied array is `1` in each dimension. If no reallocation is necessary, the left-hand side of the assignment retains its bounds and dynamic type, and does not assume the lower bound of the right.

Commented [SM19]: Put a discussion of

6.38.2 Guidance to language users

- Use allocatable components in preference to pointer components.
- Copy the objects referred to by pointer components if there is any possibility that the aliasing of a shallow copy would affect the application adversely.
- When possible, use 1 as the lower bound of array dimensions to avoid indexing mistakes.

6.39 Memory Leaks and Heap Fragmentation [XYL]

6.39.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.39 applies to Fortran as described below.

The misuse of pointers in Fortran can cause a memory leak. However, the intended functionality is often available with allocatables, which do not suffer from this vulnerability.

Commented [M20]: Really? And how is that done, because, as soon as you have individual dealllocations, fragmentation is very hard to avoid.
Same is true for memory-leaks when deallocate calls are missing.

6.39.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.20.5.
- Use allocatable data items rather than pointer data items whenever possible.
- Use final routines to free memory resources allocated to a data item of derived type.
- Use a tool during testing to detect memory leaks.

6.40 Templates and Generics [SYM]

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.40 does not apply to Fortran since Fortran does not support templates or generics.

6.41 Inheritance [RIP]

6.41.1 Applicability to language

The vulnerability specified in ISO/IEC TR 24772-1:2019 clause 6.41 applies to Fortran since Fortran supports inheritance and redefinition of type-bound subprograms. Fortran supports single inheritance only, so the complexities associated with multiple inheritance do not apply. The problem of accidental redefinition is partially mitigated by the `non overridable` attribute which prevents overriding by all subclasses. There is no mechanism to restrict a *type-bound procedure* to be a redefinition or a new procedure, respectively. Hence the vulnerabilities of accidental redefinition and non-redefinition apply.

6.41.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.41.5.
- Declare a type-bound procedure to be `non overridable` when necessary to ensure that it is not overridden by subclasses.

6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP]

6.42.1 Applicability to language

The vulnerability specified in ISO/IEC TR 24772-1:2019 clause 6.42 applies to Fortran. Fortran provides no mechanism to specify and enforce pre- and postconditions, or to prevent “has-a”-inheritance.

6.42.2 Guidance to language users

Follow the guidance of ISO/IEC 24772-1 clause 6.42.5.

6.43 Redispatching [PPH]

6.43.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.45 applies to Fortran, since calls to type-bound procedures inside inherited implementation dispatch to the dynamic type of the object in question. In Fortran, calls to procedures only dispatch when prefixed with an object.

Furthermore, Fortran allows the name of an ancestor type to prefix a call to a type-bound procedure, in which case the call is directed at the desired implementation of the procedure that applies to the named ancestor type. For example, `call obj%ancestor%method()` where `ancestor` is the name of an ancestor type of the dynamic type of `obj`. However, this is not permitted for the current type.

6.43. 2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.43.5.
- Where redispatching is undesirable, do not prefix the call of a procedure with the name of an object.

6.44 Polymorphic Variables

6.44.1 Applicability to language

The vulnerability specified in ISO/IEC TR 24772-1:2019 clause 6.45 applies to Fortran, as Fortran provides polymorphic variables.

Upcasts, as described in ISO/IEC TR 24772-1:2019 clause 6.45, are implicit in assignments and parameter passing, which always allow a value of an object of dynamic type to be assigned to a variable declared to be of any of its non-abstract ancestor types. Crosscasts or other unsafe casts are not possible in Fortran.

Downcasts are realized by `select type` constructs, where a variable selected upon assumes the selected type as its declared type for the extent of the respective block**.** Among matching guard statements, the block following the most specific guard is executed. If there is no matching guard statement, no block is executed.

The vulnerability of not handling the potential error when no guard statement matches the select type construct remains. See 6.36 Ignored error status and unhandled exceptions [OYB]. Use of the `class default` guard statement in the `select type` statement guarantees that all cases are covered.

Deleted: matches

Deleted: s

Commented [SM21]: This is not a panacea. Default cases can mask other errors, such as the addition of a new class derivative but forgetting to add specific code for that derivative.

6.44. 2 Guidance to language users

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

- Follow the guidance of ISO/IEC TR 24772-1:2019 clause 6.44.5.?
- Ensure that the default case in `select type` statements is handled.

Formatted: None, Space Before: 0 pt, After: 10 pt, Line spacing: Multiple 1.15 li, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm, Don't keep with next

Formatted: Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

6.45 Extra Intrinsics [LRM]

Formatted: Heading 3

6.45.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.45 applies to Fortran

Fortran permits a processor to supply extra intrinsic procedures. The processor that provides extra intrinsic procedures may be standard-conforming; the program that uses one is not.

Deleted: might

6.45.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.45.5.
- Specify that an intrinsic or external procedure has the `intrinsic` or `external` attribute, respectively, in the scope where the reference occurs.
- Use compiler options to detect use of non-standard intrinsic procedures.

6.46 Argument Passing to Library Functions [TRJ]

6.46.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.46 applies to Fortran since Fortran allows use of libraries written in other languages or generated by other Fortran processors.

6.46.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.46.5.
- Use libraries from reputable sources with reliable documentation and understand the documentation to appreciate the range of acceptable input.
- Verify arguments to library procedures when their validity is in doubt.
- Use condition constructs such as `if` and `where` to prevent invocation of a library procedure with invalid arguments.
- Provide explicit interfaces for library procedures. If the library provides a module containing interface bodies, use the module.

6.47 Inter-language Calling [DJS]

6.47.1 Applicability to Language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.47 applies to Fortran but is mitigated as specified below.

Fortran supports interoperating with functions and data that can be specified by means of the C programming language. The facilities provided by interoperability with C features specify the interactions and thereby limit the extent of this vulnerability.

Commented [SM22]: This needs more explanation.

When interoperating with C, Fortran arrays of single characters correspond to C strings; the NUL terminator must be added explicitly.

Commented [SM23]: More discussion needed

[SM – What about in C compatibility mode?]

6.47.2 Guidance to Language Users

- Correctly identify the companion processor, including any options affecting its types.
- Use the C interoperability features of Fortran (the `iso_c_binding` module, the `ISO_Fortran_binding.h` header file, and the `bind(C)` attribute), and use the correct constants therein to specify the type kind values needed.
- Use the value attribute as needed for dummy arguments.

6.48 Dynamically-linked Code and Self-modifying Code [NYY]

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.48 does not apply to Fortran.

The Fortran standard does not discuss the means of program translation, so any use or misuse of dynamically linked libraries is processor dependent. Fortran does not permit self-modifying code.

6.49 Library Signature [NSQ]

6.49.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.49 applies to Fortran.

6.49.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.49.5.
- Use explicit interfaces for the library code if they are available. Avoid libraries that do not provide explicit interfaces.
- Carefully construct explicit interfaces for the library procedures where library modules are not provided.

6.50 Unanticipated Exceptions from Library Routines [HJW]

6.50.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.50 applies to Fortran since Fortran allows the use of libraries and does not provide an exception handling capability.

Commented [M24]: Kontradiction! Earlier it was said that Fortran does not have exceptions. Is

Commented [M25R24]: there a similar issue with status values?

6.50.2 Guidance to language users

Formatted: Font: +Headings (Cambria), 12 pt, Bold

Formatted: Font: 12 pt

Formatted: Normal

- Follow the guidance of ISO/IEC 24772-1 clause 6.50.5.
- Translate exceptions into Fortran conformant status values and handle each error situation.
- Check any return flags present and, if an error is indicated, take appropriate actions when calling a library procedure.

6.51 Pre-Processor Directives [NMP]

6.51.1 Applicability to language

The vulnerability in ISO/IEC 24772-1 clause 6.51 does not apply to Fortran standard-conforming programs since the Fortran standard does not include pre-processing. However, some Fortran programmers employ the C pre-processor `cpp`, or other pre-processors.

The C pre-processor, as defined by the C language, is unaware of several Fortran source code properties. Some suppliers of Fortran processors also supply a Fortran-aware version of `cpp`, often called `fpp`. Unless a Fortran-aware version of `cpp` is used, unexpected results, not always easily detected, can occur.

Other pre-processors might or might not be aware of Fortran source code properties. Not all pre-processors have a Fortran-aware mode that could be used to reduce the probability of erroneous results.

6.51.2 Guidance to language users

- Avoid use of the C pre-processor `cpp`.
- Avoid pre-processors generally. Where deemed necessary, a Fortran mode should be set.
- Use processor-specific modules in place of pre-processing wherever possible.

6.52 Suppression of Language-defined Run-time Checking [MXB]

6.52.1 Applicability to Language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.35 does not apply directly to Fortran since Fortran does not require the use of runtime checks to detect runtime errors. However, the Fortran standard has many requirements that cannot be statically checked and while many processors provide options for run-time checking, the standard does not require that any such checks be provided.

6.52.2 Guidance to Language Users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.52.5.
- Use all run-time checks that are available during development.
- Use all run-time checks that are available during production running, except where performance is critical.
- Use several processors during development to check as many conditions as possible.

6.53 Provision of Inherently Unsafe Operations [SKL]

6.53.1 Applicability to Language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.53 applies to Fortran as described below.

Commented [M26]: This para belongs elsewhere … on subprog signature mismatch.

The types of actual arguments and corresponding dummy arguments are required to agree, but few processors check this unless the procedure has an explicit interface.

The intrinsic function `transfer` provides the facility to transform an object of one type to an object of another type that has the same physical representation.

A variable of one type can be storage associated through the use of common and equivalence with a variable of another type. Defining the value of one causes the value of the other to become undefined. A processor might not be able to detect this.

There are facilities for invoking C functions from Fortran and Fortran procedures from C. While there are rules about type agreement for the arguments, it is unlikely that processors will check them.

6.53.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.53.5.
- Provide an explicit interface for each external procedure or replace the procedure by an internal or module procedure.
- Avoid the use of the intrinsic function `transfer`.

Formatted: Expanded by 0.45 pt

- Avoid the use of `common` and `equivalence`.
- Use the compiler or other automatic tool for checking the types of the arguments in calls between Fortran and C, make use of them during development and in production running except where performance would be severely affected.

6.54 Obscure Language Features [BRS]

6.54.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.54 applies to Fortran since Fortran has a number of deleted and obsolescent features, plus items described below.

For use of deleted and obsolescent features, see 6.58 Deprecated language features [MEM]. Such usage can produce semantic results not in accord with the modern programmer's expectations or the knowledge of modern code reviewers. The same applies to processor-defined language extensions.

Variables can be *storage-associated* through the use of `common` and `equivalence`. Assigning the value a variable alters the value of all variables storage-associated with it. They may be of different types, in which case assigning the value of one causes the value of the other to become undefined.

Supplying an initial value for a local variable implies that it has the `save` attribute, which might be unexpected by the developer. This also makes `save`d variables shared in a multithreaded environment. If a derived type has a component with an initial value, then variables of that type have the `save` attribute.

If implicit typing is used, a simple spelling error will unexpectedly introduce a new name. The intended effect on the given variable will be lost without any processor diagnostic.

6.54.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.54.5.
- Use the processor or other static analysis tools to detect and identify obsolescent or deleted features and replace them by better methods.
- Avoid the use of `common` and `equivalence`.
- Avoid explicit and implicit usages of the `save` attribute in multithreaded contexts.
- Specify the `save` attribute when supplying an initial value.
- Use `implicit none` to require explicit declarations.

6.55 Unspecified Behaviour [BQF]

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.55 does not apply to Fortran. As all relevant cases are implementation defined behaviour. See clause 6.57 Implementation defined behaviour [FAB].

6.56 Undefined Behaviour [EWF]

6.56.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.56 applies to Fortran.

A Fortran processor is unconstrained unless the program uses only those forms and relations specified by the Fortran standard, and gives them the meaning described therein.

The behaviour of non-standard code can change between processors.

A processor is permitted to provide additional intrinsic procedures. One of these might be invoked instead of an intended external procedure with the same name.

6.56.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.56.5.
- Use processor options to detect and report use of non-standard features.
- Obtain diagnostics from more than one source, for example, use code checking tools.
- Supply an explicit interface to specify the `external` attribute for all external procedures invoked.
- Avoid use of non-standard intrinsic procedures.
- Specific the `intrinsic` attribute for all non-standard intrinsic procedures.

6.57 Implementation-Defined Behaviour [FAB]

6.57.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.57 applies to Fortran.

Implementation defined behaviour is known within the Fortran standard as processor-dependent behaviour. Annex A.2 of ISO/IEC 1539-1 (2010) contains a list of processor dependencies for which implementations should document the actual behaviour.

Different processors might process processor dependencies differently. Relying on one behaviour is not guaranteed by the Fortran standard.

Reliance on one behaviour where the standard explicitly allows several is not portable. The behaviour is liable to change between different processors.

6.57.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1:2019 clause 6.57.5.
- Use processor options to detect and report use of non-standard features.
- Obtain diagnostics from more than one source, for example, use code checking tools.
- Supply an explicit interface to specify the `external` attribute for all external procedures invoked.
- Avoid use of non-standard intrinsic procedures.
- Specific the `intrinsic` attribute for all non-standard intrinsic procedures.

6.58 Deprecated Language Features [MEM]

6.58.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.58 applies to Fortran since Fortran started in the 1950's using line-oriented and unstructured code, has been revised and updated on regular cycles since that time and has a number of deprecated language features.

Because they are still used in some programs, many processors support features of previous revisions of the Fortran standard that were deleted in later versions of the Fortran standard. These are listed in Annex B.1 of the Fortran standard. In addition, there are features of earlier revisions of Fortran that are still in the standard but are redundant and might be replaced by better methods. They are described in small font in the standard and are summarized in Annex B.2. Any use of these deleted and obsolescent features might produce semantic results not in accord with the modern programmer's expectations. They might be beyond the knowledge of modern code reviewers.

6.58.2 Guidance to language users

- Use the processor to detect and identify obsolescent or deleted features and replace them by better methods.

6.59 Concurrency – Activation [CGA]

6.59.1 Applicability to language

"The vulnerability described in ISO/IEC 24772-1 clause 6.59 is applicable to Fortran. Images in Fortran all begin concurrently but the mechanism is not specified by the language. The vulnerability is mitigated in Fortran, since mechanisms are provided to query the number of images that failed during activation.

The construct `do concurrent` – gives permission to execute a set of iterations of a loop body in parallel. The means of parallelism are unspecified and hence not subject to activation as specified in ISO IEC 24772-1 clause 6.59.

CoArrays, all images execute the complete program. All images wait at an initial point.

Document teams – does it belong in 6.59 or in data synchronization?

6.59.2 Guidance to language users

TBD

6.60 Concurrency – Directed termination [CGT]

The vulnerability as described in ISO/IEC 24772-1 clause 6.60 does not apply to Fortran, since termination of another image is not supported by Fortran except for the termination of all images by the `error stop` statement. A Fortran image can only terminate itself.

Deleted: ¶
6.60.1 Applicability to language¶

Formatted: Font: (Default) Courier New, 10.5 pt

Formatted: Font: (Default) Courier New, 10.5 pt

Deleted: TBD¶
6.60.2 Guidance to language users¶

6.61 Concurrent Data Access [CGX]

6.61.1 Applicability to language

(Current text)

The vulnerability as described in ISO/IEC 24772-1 clause 6.62 applies to Fortran. Locks and critical sections, see clause 4.8, are available to limit code execution to a single image. The concept of segments (see clause 4.8) guarantees the availability of up-to-date data.

Locks have all the vulnerabilities as documented in ISO/IEC 24772-1 clause 6.61, e.g., omissions of lock and unlock statements, while critical sections avoid these problems.

Have notion of "atomic", "volatile" and "asynchronous". Atomic does not apply to variables but applies to intrinsic (attached to coarrays).

(Proposed by JR)

The vulnerability as described in ISO/IEC 24772-1 clause 6.61 applies to Fortran. Coarray data are accessible to all images by using image selectors in square brackets. There are several mechanisms, see clause 4.8, for separating the alteration of the value of a coarray variable on one image from its access by another image. To ensure correct execution, it is essential to use one or more of these mechanisms.

6.61.2 Guidance to language users

To ensure correct execution when executing on more than one image, use one or more of the following mechanisms.

- The `sync all` statement may be used to separate the alteration of the value of a coarray variable on one image from its access by any other image.
- The `sync images` statement may be used to separate the alteration of the value of a coarray variable on one image from its access by an image in a specified set of images.
- A collective subroutine should use used whenever it is suitable.
- Integer variables of kind `atomic_int_kind` and logical variables of kind `atomic_logical_kind` may be referenced and defined in unordered segments by atomic intrinsic subroutines including `atomic_define`, `atomic_ref`, and `atomic_or`. The system ensures that for each such variable all such actions occur sequentially.
- An `event post` statement may be used on one image and a corresponding `event wait` statement may be used on another. Statements executed on the first image ahead of its execution of the `event post` statement precede statements executed on the other image after its execution of the corresponding `event wait` statement.
- A critical section may be used to limit execution to one image at a time; if performance using critical sections unacceptable, use locks and use analysis to show correct lock behaviour.

Avoid

- The use of the `volatile` attribute.
- The use of the `asynchronous` attribute except for use with a parallel-processing package such as MPI for nonblocking data transfer.
- The use of the `sync memory` statement for defining and ordering segments.

Delete the following?

6.62 Concurrency – Premature termination [CGS]

6.62.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.62 applies to Fortran. It is mitigated by language features for detecting failed images (processes) and conditionally continuing execution in their presence. See clause 4.8 for an explanation of parallel execution in Fortran.

Commented [SM27]: This does not address issues with massively parallel systems yet.

6.62.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.62.5.
- Use the intrinsic functions `failed_images`, `stopped_images`, and `image_status` to detect failed and stopped images.
- If continued execution is not desired in the presence of failed images, terminate the executing images normally after they have performed any useful computation that is available.
- In order to continue execution in the presence of failed images, from time-to-time store relevant information for each team of images externally or on another team, so that the computation can be resumed on a reduced number of images or with images kept in reserve and idle replacing failed images.

6.63 Protocol Lock Errors [CGM]

6.63.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1 clause 6.63 applies to Fortran.

Commented [SM28]: John to add provide more c

To mitigate the vulnerabilities associated with explicit locks, Fortran provides safer synchronization constructs, see clause 4.8.

6.63.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-1 clause 6.63.5.
- Use other synchronizations, such as `sync all` or `sync images`, or the collective subroutines whenever possible.

Formatted: Font: Not Italic

6.64 Uncontrolled Format String [SHL]

6.64.1 Applicability to language

Most of the vulnerability as described in ISO/IEC 24772-1 clause 6.64 does not apply to Fortran. Fortran provides the ability to control input or output via format strings and mistakes in format strings may cause serious program errors. However, the format string cannot affect the access of memory beyond the data items being referenced.

6.64.2 Guidance to language users

- Wherever possible, use format strings that are constants.
- Where a variable string is needed, check that its value is within expectations.

Commented [SM29]: Check how?

6.65 Modifying constants [UJO]

6.65.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.65 is applicable to Fortran.

Fortran does not allow a constant to be the target of a pointer, and does not allow a type to have a constant as a component. Fortran also prevents all attempts to write directly to a variable declared constant and prevents passing a constant to an `out` or `inout` dummy argument in a subprogram.
Fortran compilers usually do not prevent the use of a constant as an actual argument in the absence of an intent specification.

Commented [SM30]: JR - research other ways that constants can be modified.

Compilers prevent the alteration of the value of a constant

6.65.2 Guidance to language users

- Always use intent specifications for dummy arguments
- Use the compiler or static analysis tools to detect any use of a constant that is not in accord with the Standard.

7 Language specific vulnerabilities for Fortran

8 Implications for standardization

Future standardization efforts should consider:

- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of integer overflows during program execution.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of out-of-bounds subscripts and array-shape mismatches in assignment statements during program execution.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of invalid pointer references during program execution.

- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of an invalid use of character constants as format specifiers.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of tests for equality between two objects of type real or complex.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of pointer assignment of a pointer whose lifetime is known to be longer than the lifetime of the target or the `target` attribute of the target.
- Requiring that processors have the ability to detect and report the occurrence within a submitted program unit of the reuse of a name within a nested scope.
- Providing a means to specify explicitly a limited set of entities to be accessed by host association.
- Identifying, deprecating, and replacing features whose use is problematic where there is a safer and clearer alternative in the modern revisions of the language or in current practice in other languages.

Bibliography

[1] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2004

[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 (all parts), *International terminology standards*

[7] ISO/IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

[9] ISO/IEC 8652:1995, *Information technology — Programming languages* — Ada

[11] R. Seacord, *The CERT C Secure Coding Standard*. Boston,MA: Addison-Westley, 2008.

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

[17] ISO/IEC TR 24718: 2005, *Information technology — Programming languages — Guide for the use of the Ada Ravenscar Profile in high integrity systems*

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

[20] 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.

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

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

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

1. Lecture Notes on Computer Science 5020, "Ada 2012 Rationale: The Language, the Standard Libraries," John Barnes, Springer, 2012. ???????

[25] Steve Christy, *Vulnerability Type Distributions in CVE*, V1.0, 2006/10/04

[29] Lions, J. L. ARIANE 5 Flight 501 Failure Report. Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[33] The Common Weakness Enumeration (CWE) Initiative, MITRE Corporation, (http://cwe.mitre.org/)

[34] Goldberg, David, *What Every Computer Scientist Should Know About Floating-Point Arithmetic*, ACM Computing Surveys, vol 23, issue 1 (March 1991), ISSN 0360-0300, pp 5-48.

[35] IEEE Standards Committee 754. IEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Standard 754-2008. Institute of Electrical and Electronics Engineers, New York, 2008.

[36] Robert W. Sebesta, Concepts of Programming Languages, 8th edition, ISBN-13: 978-0-321-49362-0, ISBN-10: 0-321-49362-1, Pearson Education, Boston, MA, 2008

[37] Bo Einarsson, ed. Accuracy and Reliability in Scientific Computing, SIAM, July 2005 http://www.nsc.liu.se/wg25/book

[38] GAO Report, Patriot *Missile Defense: Software Problem Led to System Failure at Dhahran, Saudi Arabia*, B-247094, Feb. 4, 1992, http://archive.gao.gov/t2pbat6/145960.pdf

[39] Robert Skeel, *Roundoff Error Cripples Patriot Missile*, SIAM News, Volume 25, Number 4, July 1992, page 11, http://www.siam.org/siamnews/general/patriot.htm

[41] Holzmann, Garard J., Computer, vol. 39, no. 6, pp 95-97, Jun., 2006, *The Power of 10: Rules for Developing Safety-Critical Code*

[42] P. V. Bhansali, A systematic approach to identifying a safe subset for safety-critical software, ACM SIGSOFT Software Engineering Notes, v.28 n.4, July 2003

[43] Ada 95 Quality and Style Guide, SPC-91061-CMC, version 02.01.01. Herndon, Virginia: Software Productivity Consortium, 1992. Available from: http://www.adaic.org/docs/95style/95style.pdf

[44] Ghassan, A., & Alkadi, I. (2003). Application of a Revised DIT Metric to Redesign an OO Design. *Journal of Object Technology* , 127-134.

[45] Subramanian, S., Tsai, W.-T., & Rayadurgam, S. (1998). Design Constraint Violation Detection in Safety-Critical Systems. The 3rd IEEE International Symposium on High-Assurance Systems Engineering , 109 - 116.

[46] Lundqvist, K and Asplund, L., "*A Formal Model of a Run-Time Kernel for Ravenscar*", The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

Index

Page 27: [1] Deleted **Stephen Michell** **5/23/22 11:52:00 AM**

Page 27: [2] Deleted **Stephen Michell** **2/25/20 1:55:00 PM**