ISO/IEC JTC 1/SC 22 N1319

Date: 20230828

ISO/IEC TR 24772-8

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

DRAFT DRAFT DRAFT

Programming languages — Avoiding vulnerabilities in programming languages – Vulnerability descriptions for the programming language Fortran

Élément 1ntroductive — 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

Contents

Table of Contents

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 Standard documents avoidance mechanisms 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. It can also be used in comparison with companion standards and with the language-independent report, ISO/IEC 24772-1 *Programming languages — Avoiding vulnerabilities in programming language -- Part 1: Language-independent catalogue of vulnerabilities*, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

It should be noted that this document 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.

Programming Languages — Programming language vulnerabilities – Part 8:
Vulnerability descriptions for the programming language Fortran

1. Scope

This Standard itemizes 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.

This Standard documents how the vulnerabilities described in the language-independent writeup, ISO/IEC 24772-1, are manifested in Fortran and provides mechanisms to avoid them.

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 Programming languages — Avoiding vulnerabilities in programming language --, Part 1: Language-independent catalogue of vulnerabilities
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

3.4

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

3.5

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

3.6

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

3.7

Image
one of a mutually cooperating set of instances of a Fortran program, each with its own execution state and set of data objects

3.8

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

3.9

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

3.10

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

3.11

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

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

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.

Most Fortran processors support ISO/IEC/IEEE 60559:2011, the IEEE standard for floating-point arithmetic. The floating-point standard defines binary patterns that represent floating-point values, signed zeros, and signed infinities; any other value is a *NaN* (Not a Number). This allows IEEE arithmetic to be closed, that is, every operation has a result. If an exception occurs, execution continues with the corresponding flag signaling, and the flag remains signaling until explicitly set quiet by the program. The flags are therefore called *sticky*. The flags are *overflow*, *divide_by_zero*, *invalid* (for example 0.0/0.0 or when an operand is a NaN), *underflow*, and *inexact* (when the result cannot be represented exactly). There are five corresponding Fortran exception flags. Each has a value that is either *quiet* or *signaling* and its initial value is quiet. There are procedures for finding and for resetting the value of a flag. If a flag is signaling on entry to a procedure, the processor will set it to quiet on entry and restore it to signaling on return. This allows exception handling within the procedure to be independent of the state of the flags on entry, while retaining their 'sticky' properties.

The Fortran standard places minimal constraints on the representation of entities of type character and type logical.

4.7 Interoperability

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 Allocatable variables

An allocatable variable or component is declared with a rank (dimensionality) but without data. It is associated with data by an allocate statement and is then said to be allocated. If it is an array, the allocate statement provides it with bounds. Its data is released to the system by a deallocate statement and it is then said to be unallocated. Its initial status is unallocated. Its allocation status is either allocated or unallocated. While it has many of the properties of a pointer variable, it cannot give rise to memory leakage or a dangling pointer. Assignment between allocatable variables of the same rank copies their data.

4.9 Polymorphism

Fortran supports object orientation with single inheritance. A derived type `ta` may be extended to form a new type `tb` with all the components of type `ta` plus possibly additional components. The extended type `tb` also has a parent component of type `ta` with the name `ta` and the type and type parameters of the parent type. Access to the components is illustrated by the following example.

```
type ta
   real :: x
end type
```

```
type, extends (ta) :: tb
   integer :: i
end type
type(tb) :: bobj
. . .
bobj%x  = 1
bobj%ta%x = 2 ! Overwrites the previous assignment of 1
```

A variable can be declared as polymorphic; it has a declared type and a dynamic type that is permitted to be the declared type or any extension of the declared type. A type declaration can bind existing procedures to the type; each has a binding name that can be the same as the name of the existing procedure. The existing procedure usually has a dummy argument of the type that is given the `pass` attribute. A type-bound procedure is invoked as if it were a component of the object; if the procedure has an argument with the `pass` attribute, the corresponding actual argument must be omitted from the argument list and the invoking object is passed automatically. Here is an example

```
module m
   type ta
      real :: x = 7.2
   end type
   type, extends (ta) :: tb
      integer :: i
   contains
      procedure :: proc => foo  ! hence a call to obj%proc() is equivalent to
                                ! the call foo(obj)
   end type
contains
   real function foo( arg )
      class(tb) :: arg
      foo = arg%x
   end function
end module m
. . .
   use m
   type(tb) :: bobj
   real :: y
   y = bobj%proc()   ! y is assigned the value 7.2
```

Binding names are inherited by extensions of the type but can be overridden by a specification for the same name in the definition of an extended type. Which procedure is invoked in a type-bound reference is determined by the dynamic type of the object through which the procedure is referenced. To execute alternative code depending on the dynamic type of a polymorphic entity and to gain access to the dynamic parts, the `select type` construct is provided.

4.10 Parallelism

4.10.1 Images and coarrays

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 the executing 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.

All data objects are local to their respective image, but a data object declared as a *coarray* can be accessed from another image. This access is accomplished by using *cosubscripts* in square brackets to indicate the image being accessed. A coarray can be scalar or an array.

4.10.2 Locks

The `lock` and `unlock` statements provide a 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 scalar coarray or an element of an array 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.

Data protection can also be achieved with a `critical` construct, which limits execution of the construct to one image at a time:

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

This would, however, prevent several images executing the code at the same time for different values of `p`.

4.10.3 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.

4.10.4 Segments

Any statement that implies ordering between the execution of statements on different images is known as an *image control statement*. For example, `sync all` and `sync images` are image control statements.

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 statements imply a partial ordering of all the segments on all the images.

If the value of a variable or a part of it is altered in a segment, it is permitted to be referenced in another segment only if the two segments are ordered, or if the variable is:

- `integer` of kind `atomic_int_kind` or `logical` of kind `atomic_logical_kind` (see clause 4.10.5);
- asynchronous (see clause 4.10.6); or
- volatile (see clause 4.10.7).

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.

4.10.5 Atomic actions

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. Such variables are not volatile by the Fortran language rules.

4.10.6 Asynchronous variables

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

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

The `asynchronous` attribute is useful both for I/O of large blocks of data and for interoperating with parallel-processing packages such as MPI. MPI provides procedures such as `MPI_Irecv` and `MPI_Isend` for nonblocking transfer of data between processes. For example, in the code

```
subroutine UpdateBuf( buf, … )
   real :: buf(100, 100)
   . . . code that involves buf.
   block
      asynchronous :: buf
      call MPI_Irecv(buf,. . . req, . . . )
      . . . code that does not involve buf.
      call MPI_Wait(req, . . . )
   end block
    . . . code that processes buf.
```

`MPI_Irecv` initiates input communication and can return while the communication (reading values into `buf`) is still underway. The code between `MPI_Irecv` and `MPI_Wait` can execute without waiting for this communication to complete provided it does not involve `buf`. Similar code with the call of `MPI_Irecv` replaced by a call of `MPI_Isend` is asynchronous output communication. It should be noted that any attempt to access `buf` between the `MPI_Irecv` and the `MPI_Wait` can result in corruption of data, at least.

4.10.7 Volatile 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.

4.10.8 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.

4.10.9 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.

4.10.10 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 avoidance mechanisms 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.

Number	Software developers can	References
1.	Use static analysis tools, including Fortran compilers, to detect problematic code, such as • Language features that are obsolescent, non-conforming, or deleted • Uninitialized variables • Integer overflows Enable the compiler's detection of such code	6.22, 6.25, 6.53, 6.56, 6.57, 6.54, 6.58
2	Enable bounds checking and pointer checking throughout development of a code and only disable such checking during production runs when performance requirements cannot be met otherwise.	6.8 6.14
3	Use all run-time checks that are available during development to detect: • Uninitialized variables • Real value exceptions • Integer overflows • Null pointer checks • Dangling pointer checks	6.2 6.15 6.36 6.52
4	Declare all variables and use `implicit none` to enforce this.	6.17 6.21 6.54 7.1
5	Use an allocatable object in an assignment where differently-sized objects might occur so the left-hand side object is reallocated as needed.	6.8 6.9 6.38
6	Use allocatable objects in preference to pointer objects unless pointer assignment is required.	6.13 6.14 6.33, 6.38, 6.39
7	Avoid implicit interfaces; use explicit interfaces.	6.11, 6.32, 6.34, 6.46, 6.49, 6.53, 6.56, 6.57
8	Avoid using keywords as names and reusing names in nested scopes.	6.17, 6.20
9	In `select` constructs, cover cases that are expected never to occur with a `default` clause to ensure that unexpected	6.27 6.44

	cases are detected and processed, for example by emitting an error message.	
10	Specify argument intents to allow further checking of argument usage.	6.32 6.65
11	Avoid the use of the intrinsic function `transfer`.	6.53
12	Use procedures from a trusted library to perform calculations where floating-point accuracy is needed.	6.4
13	Test all diagnostic status values returned by procedure calls	6.36
14	Include an `iostat` or `stat` variable when possible and check its value to ensure no errors occurred.	6.6 6.8 6.14 6.59
15	For parallel programming • Use coarrays only when communication among images is necessary. • Use collective subroutines whenever possible.	6.61 6.63

6 Specific analysis for Fortran

6.1 General

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 the analysis of Fortran-specific issues addressed in clause 6.2 in this document.

6.2 Type system [IHN]

6.2.1 Applicability to language

The vulnerabilities documented in ISO/IEC … apply to Fortran. They are partially mitigated by Fortran's strong type system, as described below.

The Fortran type system is a strong type system consisting of data types 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.

Intrinsic assignment between user-defined types is only permitted if the types are *type compatible*, and therefore the vulnerability associated with structural equivalence documented in ISO/IEC 24772-1 does not exist in Fortran.

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
```

can be used to distinguish Celcius and Fahrenheit temperatures. The following code would not conform to the standard.

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

6.2.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 subclause 6.2.5;
- 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 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;
- 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.

6.3.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the language-provided intrinsics whenever bit manipulations are necessary, especially those that occupy more than one integer.
- Encapsulate bit strings inside derived types to exclude numeric operations on them.
- 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`) .
- Avoid using 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.

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 much 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; most processors support the rounding mode being changed during execution under program

control. A separate rounding mode is provided for input/output formatting conversions; this rounding mode is required to be supported and can also be changed during execution.

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

6.4.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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, and use compiler options where available to detect such usage.
- Avoid using floating-point variables as loop indices, as it is a deleted feature; use integer variables instead;
- Use intrinsic inquiry procedures, when needed, to determine the properties of the representation in use;
- Avoid the use of bit operations to get or to set the parts of a floating-point quantity, and use intrinsic procedures to provide the functionality when needed;
- Where the IEEE intrinsic modules and the IEEE real kinds are in use, use the intrinsic module procedures to determine the limits of the processor's conformance to ISO/IEC/IEEE 60559 and to determine the limits of the representation in use;
- Where the IEEE intrinsic modules are in use, use the intrinsic module procedures to detect and control the available rounding modes and exception flags.

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 and arrays are indexed by them. The arrays must be dimensioned accordingly to accommodate all these (and other) integer values as indices. 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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;
- Avoid the use of variables assigned enumeration values in arithmetic operations, or the use of variables 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 numeric types is allowed, with the associated vulnerabilities documented in ISO/IEC 24772-1 subclause 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.

Intrinsic assignment provides automatic conversion between default and ASCII character kinds, and from these kinds to ISO/IEC 10646 character kind.

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 nonzero value.

Conversions between incompatible types can be achieved by explicitly invoking user-provided conversion functions, e.g.

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

for conversion from Fahrenheit to Centigrade.

6.6.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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, and 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;
- Include an IOSTAT variable in each IO statement and check its value after each IO operation to ensure any errors that occurred are processed appropriately.

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.

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 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.

Any undetected bounds violations result in undefined behaviour; see 6.56 Undefined behaviour [EWF].

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.

If the character variable that defines an internal file is too small for the output sent to it, an error condition results. This can be detected with an `iostat=` or `err=` specifier; without one of these, error termination occurs.

Most implementations include an optional facility for bounds checking. These bounds checks 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 functions so the length of character entities can be reliably found.

6.8.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 code development and only disable such checking during production runs when performance requirements cannot be met otherwise and after extensive static anlaysis and testing to ensure that bounds are not ignored.;
- Use whole array assignment, operations, and bounds inquiry intrinsics where possible;
- Use allocatable arrays where array operations involving differently-sized arrays might occur so the left-hand side array is reallocated as needed;
- Obtain array bounds from array inquiry intrinsic procedures wherever needed and 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 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 that the truncate-or-blank-fill semantic will protect against storing outside the assigned variable;
- Consider using the `iostat=` specifier when there is a risk that an internal file is too small for the output sent to it;

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 can be affected by this vulnerability when an array subscript is outside its bounds. The Fortran standard requires that each array subscript be within its bounds, not simply that the resulting offset be within the array as a whole, but implementations are not required to diagnose this.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.9.5;
- Ensure that consistent bounds information about each array is available throughout a program;
- Enable bounds checking, when available, throughout development of a code, and only disable bounds checking during production runs and 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, and 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 Unchecked array indexing [XYZ].

6.10.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can use the avoidance mechanisms of 6.8.2 Buffer boundary violations [HCB].

6.11 Pointer type conversions [HFC]

6.11.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.11 is applicable to Fortran in the following cases: in the context of polymorphic pointers; the C-style pointer conversion intrinsics; and in the use of implicit interfaces for procedure pointers and dummy procedure arguments. All other pointer conversions are forbidden.

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 incompatible arguments. Similarly, a dummy procedure with an implicit interface can be associated with an actual procedure that has a different interface, with the risk of passing incompatible arguments. 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. All use of sequence types is error prone because no checks are made by the compiler for components of the wrong type or shape.

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.

6.11.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.11.5;
- Avoid implicit interfaces; use explicit interfaces instead;
- Avoid the use of C-style pointers, unless necessary to interface with C programs;
- Avoid sequence types.

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. For a pointer whose association status is defined, the Fortran intrinsic procedure `associated` determines whether a pointer

- has a valid target, i.e. is not NULL, or
- is associated with a particular target.

This vulnerability also occurs for a pointer whose pointer association status is undefined, meaning that a request about its association status is unreliable.

In Fortran, it is invalid to reference an allocatable variable or component (see clause 4.8) that is not allocated.

Some processors include an optional facility for pointer checking.

6.13.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms s of ISO/IEC 24772-1 clause 6.13.5;
- Ensure that all pointers have a defined association status before use, either by initialization or by pointer assignment;
- Consider using `allocatable` instead of `pointer` when possible, since the allocation status of allocatable variables or allocatable components cannot be undefined;
- Use static analysis tools and compiler options where available to enable pointer checking during development of a code;
- Use the `associated` intrinsic procedure before referencing a target through a pointer if there is any possibility of the pointer being null;
- Use default initialization in the declarations of pointer components.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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;
- Use compiler options where available to detect dangling references;
- Enable pointer checking throughout development of code and only disable such checking during production runs when performance requirements cannot be met otherwise;
- Avoid pointer-assigning a pointer to a target if the pointer might have a longer lifetime than the target or the target attribute of the target, and 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can avoid using shift intrinsics where integer multiplication or division is intended.

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.

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

6.17.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.17.5;
- Declare all variables and use `implicit none` to enforce the declaration of all variables before use;
- Disable implicit typing through the use of the `implicit none` statement.
- Avoid using consecutive underscores in a name;
- Avoid using keywords as names;
- 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.18.5.
- Declare all variables and use `implicit none` to enforce the declaration of all variables before use;

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.19.5
- Declare all variables and use `implicit none` to enforce the declaration of all variables before use;

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.20.5;
- Avoid reusing 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:
- Avoid implicit typing, always declare all variables, and 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 Missing 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:
- Use the avoidance mechanisms 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 components of 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 identify 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.23.5;
- Consult the Fortran reference manual or suitable reference books for definitive information on specific operator precedence and associativity issues.

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. Non-intrinsic Fortran functions are permitted to have side effects, unless the function is declared to have the `pure` attribute. Within 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 results 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 there. 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 that cannot be 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 the `select case` construct, the `select type` construct and the `select rank` construct. In each of these constructs, control never flows from one alternative to another, but it can happen that no case is executed unless a default clause is included in each usage.

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

6.27.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.27.5;
- Cover cases that are expected never to occur with a `default` clause to ensure that unexpected cases are detected and processed, for example by emitting an error message;
- Avoid the use of the computed `go to` statement.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 `goto` statements;
- Use names on block constructs to provide matching of initial statement and end statement for each construct.

6.29 Loop control variable abuse [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, except in circumstances documented here, where the compiler does not enforce prohibitions defined by the Fortran standard.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- 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.

The vulnerability is mitigated in Fortran, as Fortran 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 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.

6.30.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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.

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, but is mitigated to some extent.

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 attributes, including type, kind, and rank, of arguments and result variables of functions.

Fortran does not specify the argument-passing mechanism, but rather specifies the rules of *argument association*. These rules are generally implemented either by reference or by copy. 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. Copying can be limited by the programmer specifying `intent(in)`, `intent(out)`, or `value`.

Incorrect choice of parameter passing mechanism is therefore minimized, provided the intent specifications for the arguments are supplied and correct. Moreover, the vulnerability of passing an incorrect address of a data structure is limited by the requirement that targets of pointers always have the correct type, kind, and rank.

On the other hand, a vulnerability arises if the programmer relies on a particular parameter mechanism but the compiler chooses a different one. This is particularly the case when aliasing is present.

Aliasing cannot occur for arguments declared with the `value` attribute. Aliasing does not accord with the Fortran standard, but its detection is unlikely unless runtime checks are available and are employed. Aliasing effects inside procedures can depend on the argument-passing mechanism chosen by the compiler.

The vulnerability of an uninitialized result value or `intent(out)` argument exists, when it is not assigned a value in the subprogram.

6.32.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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.
- If available, use runtime checks against aliasing, at least during development.
- Ensure that the result of a function is assigned, potentially through the use of static analysis tools or explicit runtime checks.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.33.5;
- Avoid pointer assignment to a target if the pointer association has 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.34.5;
- Use explicit interfaces, for example by placing procedures inside a module or making them internal procedures;
- Use a processor or a static analysis tool that check all interfaces;
- 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. Also, some recursion patterns cannot be reduced to iterations, short of programming a call stack explicitly.

6.35.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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.

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. Many Fortran statements and some intrinsic procedures return such a status value. A failure by the invoking program to request the status value when there is an error results in error termination of the program. Some programmers, however, in order to "keep going" request the status value but do not examine it. This can result in unbounded program errors when subsequent steps in the program rely upon the previous statements having completed successfully, see 6.56 Undefined behaviour [EWF].

The intrinsic module `ieee_exceptions` is defined by the standard for the support of floating-point exceptions (see clause 4.6) and is provided by most processors. Accessing this module allows the program to test the Fortran flags.

Fortran does not support exception handling of the kind described in ISO IEC 24772-1 subclause 6.36.3 para 4. For each of the Fortran flags, some processors allow control during program execution of whether to halt image execution or continue after the flag is raised. Halting is not precise and may occur any time after the exception has occurred.

Fortran does not support detection of integer overflow (see clause 6.15), but some compilers have an option for detecting it.

6.36.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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;
- Check and respond to all status values that are returned by an intrinsic procedure or by a library procedure;
- Use compiler options where available to detect integer overflow.

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 function or the obsolescent features `common`, `equivalence`, and `entry`. In particular, standard Fortran does not provide other means to convert between unrelated types.

The intrinsic function `transfer` permits the unchecked copying from a value to a specified (different) type.

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

6.37.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Avoid use of the `transfer` intrinsic unless its use is unavoidable, and then document the use carefully;
- Avoid the use of `common` to share data. Use module variables instead;
- Avoid the use of `equivalence`. If the intent is to save storage space, use allocatable data instead;
- Avoid the use of `entry`, but instead use a module containing any private data items, with a module procedure for each entry point and the shared code in a private module procedure;
- Use compiler options where available to detect the obsolescent features `common`, `equivalence`, and `entry`.

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 deep copy except for pointer components. The operator => performs *pointer assignment.*

For assignment (the operator `=`), data structures that do not contain pointers are completely copied. *Allocatable* components (see clause 4.8) are completely copied, and 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.

6.38.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- 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.

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 cannot cause memory leaks. Multiple allocations using pointers or allocatables may cause fragmentation.

6.39.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.39.5;
- 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 preconditions and postconditions, but the programmer may have this in mind and include tests in the code. Fortran has no mechanism to prevent “has-a” inheritance.

6.42.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.42.5;
- Consider enforcing preconditions and postconditions by inserting explicit checks in the code.

6.43 Redispatching [PPH]

6.43.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.43 applies to Fortran as Fortran semantics imply redispatching of nested calls..

6.43. 2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.43.5;
- Monitor the depth of recursion and limit it;
- Declare type bound procedures as `non_recursive` if they are not intended ever to be called recursively.
- When overriding a type-bound procedure, check that its uses by other procedures bound to the type are not affected.

6.44 Polymorphic variables [BKK]

6.44.1 Applicability to language

The vulnerability specified in ISO/IEC TR 24772-1:2019 clause 6.44 applies to Fortran, as Fortran provides polymorphic variables. However, the vulnerability is mitigated by restricting casts to safe ones only. Only the vulnerabilities related to the handling of existing data components by casting operations remain.

Upcasts, as described in ISO/IEC TR 24772-1:2019 clause 6.44, are implicit in assignments and parameter passing, which always allow a value of an object of dynamic type to be assigned to a polymorphic 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. Use of the `class default` guard statement in the `select type` statement guarantees that all cases are covered; however, its use can mask subsequently-added child types for which explicit handling is necessary.

6.44.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC TR 24772-1:2019 clause 6.44.5;
- Use the `class default` guard statement to provide code that indicates an error or clearly document why such behaviour is acceptable;
- Avoid using the intrinsic function `transfer` to perform an unsafe cast.

6.45 Extra intrinsics [LRM]

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 or extra intrinsic modules but requires language processors to be able to diagnose their usage. The use of such intrinsics is not standard-conforming, even if the processor that provides them is standard-conforming.

6.45.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.45.5;
- Specify that a procedure has the `intrinsic` attribute in a scope where the intrinsic procedure is referenced;
- Specify `intrinsic` or `non_intrinsic` on a `use` statement for a module;
- Use compiler options to detect use of non-standard intrinsic procedures and modules.
- Use static analysis tools and human review to detect the use of extra intrinsics.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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 for interoperability with C features specify the interactions and thereby limit the extent of this vulnerability.

When exchanging character strings with C, it is crucial to handle the fact that C terminates all strings with NUL and that Fortran uses a different mechanism to specify string length.

When interoperating with C, Fortran strings correspond to C strings; the NUL terminator must be handled explicitly.

Be aware that certain Fortran dummy arguments interoperate with a "C descriptor", as defined by the Fortran standard. These will require special code in the other language procedure to properly receive or pass the argument.

6.47.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms documented in ISO/IEC 24772-1 clause 6.47.5;
- 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;
- Perform IO on any given file in one programming language only.

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 is mitigated in Fortran since Fortran supports explicit interface specification to libraries written in other languages. However, it is not required that the interface specification be given or enforced by the compiler. Also, the actual interface correspondence with the foreign language remains unchecked.

6.49.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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;
- Use processor options and static analysis tools to detect and report signature mismatches.
- 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.

6.50.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- For libraries written in other languages, use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.50.5
- Wrap the procedure in the foreign code to 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, see clause 6.36 Ignored error status and unhandled exceptions [OYB].

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, in which case, the vulnerability applies.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Avoid use of the C pre-processor `cpp`;
- Avoid pre-processors generally, and where deemed necessary, ensure that a Fortran mode is 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 6.52 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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms 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`;
- Avoid the use of `common` and `equivalence`;
- Use multiple compilers from different sources or explicit static analysis tools to detect erroneous situations.
- 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.

The `save` attribute for a local variable causes its definition to be retained across calls to its subprogram. This also makes `save` variables shared between recursive invocations of a procedure or shared in a `do concurrent` construct if declared within that construct.

Supplying an initial value for a local variable as part of the declaration implicitly gives it the `save` attribute, which might be unexpected by the developer. However, the default initialization of a component of a variable of derived type does not affect the `save` attribute of that variable.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.54.5;
- Use the processor, multiple processors or other static analysis tools to detect and identify obsolescent or deleted features and replace them by better methods;
- Avoid explicit and implicit usages of the `save` attribute in recursive invocations of a procedure and in `do concurrent` constructs ;
- Specify the `save` attribute when supplying an initial value;
- Use `implicit none` to enforce explicit declarations.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability to language

The vulnerability specified in ISO/IEC 24772-1:2019 clause 6.55 applies to Fortran. Examples include:

- The order of evaluation of actual arguments of a procedure call is unspecified.
- Short circuit of logical operations is unspecified in Fortran.
- Freedom is given to the language processor to evaluate a mathematically equivalent expression, despite the order of evaluation of compound expressions being specified by the language. In the case of real arithmetic, rounding errors can therefore lead to different results.

Many relevant cases listed in ISO/IEC 24772-1:2019 clause 6.55 are implementation-defined behaviour. See clause 6.57 Implementation-defined behaviour [FAB].

6.55.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can use the avoidance mechanisms of ISO/IEC TR 24772-1 clause 6.55.5.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.56.5;

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 1539-1:2018 contains a list of processor dependencies for which implementations should document the actual behaviour.

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 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.57.5;
- Use processor options and static analysis tools to detect and report use of processor-dependent non-standard features;
- Obtain diagnostics from more than one source, for example, use code checking tools or multiple Fortran compilers;
- Specify the `intrinsic` attribute for all intrinsic procedures and modules referenced.
- Avoid the use of 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. Fortran was originally defined 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 for which better methods are available in ISO/IEC 1539-1:2018. The obsolescent features are identified by small font in the standard and are summarized in Annex B.2 of that standard. Any use of these deleted and obsolescent features may, according to ISO/IEC 1539-1:2018, produce results not in accord with the modern programmer's expectations and can be beyond the knowledge of modern code reviewers.

6.58.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can use the processor, multiple processors and static analysis tools 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 during program activation; however the semantics of Fortran do not separate the consequences of failure during activation from failures during general execution, hence the vulnerabilities involved in activation are subsumed by the vulnerabilities described in clause 6.62 Concurrency -- Premature termination.

Images in Fortran all start asynchronously but the mechanism is not specified by the language. Failure of an image can be detected by an executing image by executing one of the intrinsic functions `failed_images` and `image_status`, or by examining the status variable after executing a statement that involves access to data on another image.

To ensure that all images have activated successfully, one can insert a `sync all` statement with an `iostat=` *specifier*. If this detects a failed image, all images can be terminated by any image executing an `error stop` statement.

The construct `do concurrent` gives permission to execute a set of iterations of a loop body in parallel but the mechanisms by which this is achieved are not specified. It is the responsibility of the implementation to indicate that the image has failed if it is unable to execute the construct.

6.59.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1:2019 clause 6.59.5;
- At the start of the program insert a `sync all` statement with an `iostat=` specifier to ensure that all images have activated successfully.

6.60 Concurrency – Directed termination [CGT]

6.60.1 Applicability to language

The vulnerability of external termination of another image, as described in ISO/IEC 24772-1 clause 6.60,does not apply to Fortran which supports external termination only of all images are terminated by the `error stop` statement. There remains the vulnerability associated with ignored requests to terminate and the vulnerability associated with delayed termination.

6.60.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.60.5 as applicable;

6.61 Concurrent data access [CGX]

6.61.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.61 applies to Fortran. It is mitigated by several language features. Data are accessible across image boundaries:

- By using an image selector in square brackets.
- By invoking a collective (intrinsic) procedure (see clause 4.10.8).
- By invoking a procedure that has an image selector in square brackets.

All atomic changes of values of variables (clause 4.10.8) occur sequentially. For all coarray data, Fortran provides the following mechanisms for serializing the alteration of the value of a variable on one image from its access by another image:

- The `sync all` and `sync images` statements (clause 4.10.1).
- Events (clause 4.10.1).
- The `critical` construct (clause 4.10.1).
- Locks (clause 4.10.2).
- Teams (clause 4.10.3).
- Collectives (clause 4.10.8).

6.61.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.61.5;
- Use a coarray only when communication among images is necessary for that object;
- Use one or more of the following mechanisms to ensure correct execution when executing on more than one image;
    - Use the `sync_all` statement to separate the alteration of the value of a coarray variable on one image from its access by any other image;
    - Use the `sync_images` statement 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;
    - Use a collective subroutine whenever suitable;
    - Use integer variables of kind `atomic_int_kind` and logical variables of kind `atomic_logical_kind` and use atomic intrinsic subroutines including `atomic_define`, `atomic_ref`, and `atomic_or` to guarantee sequential access;

    - Use the `event post` statement in one image and the corresponding `event wait` statement on another image to impose sequential ordering;
    - Use the `critical` construct to limit execution of a section of code to one image at a time; if performance using critical sections is unacceptable, use locks and perform analysis to show correct lock behaviour;
- Avoid the use of the `volatile` attribute;
- Avoid the use of the `asynchronous` attribute except for use with a parallel-processing package such as MPI for nonblocking data transfer;
- Avoid the use of the `sync memory` statement for defining and ordering segments.

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, as images can prematurely terminate in various ways. It is mitigated by language features for detecting failed images (clause 4.10.9) and conditionally continuing execution in their presence.

6.62.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.62.5;
- Use the intrinsic functions `failed_images`, and `image_status` to detect failed images;
- 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;
- If continued execution is not desired in the presence of failed images, follow a strategy that ensures safe termination of the executing images;
- If a procedure needs to abort, avoid executing a `stop` statement – instead return with an error flag set.

6.63 Protocol lock errors [CGM]

6.63.1 Applicability to language

The vulnerabilities as described in ISO/IEC 24772-1 clause 6.63 apply to Fortran with “image” corresponding to the term “thread”. There are several mechanisms (see clause 6.61.1) for ensuring that the sequencing of the execution of the images leads to the intended results. It is essential to use one or more of these mechanisms to avoid the disruptions discussed in ISO/IEC 24772-1 clause 6.63.

6.63.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Use the avoidance mechanisms of ISO/IEC 24772-1 clause 6.63.5;
- Use the avoidance mechanisms listed in 6.61.2, bullet 3.

6.64 Reliance on external format strings [SHL] [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. If the format string is constant, then it cannot be influenced by external input or by program state.

6.64.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

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

6.65 Modifying constants

6.65.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.65 is applicable to Fortran. The vulnerability is mitigated by the following language properties.

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. However Fortran permits a pointer to an `in` subprogram dummy argument, and a subsequent write via the pointer.

Fortran compilers usually do not prevent the use of a constant as an actual argument in the absence of an intent specification.

6.65.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

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

7 Language specific vulnerabilities for Fortran

7.1 General

The vulnerabilities document in this clause are specific to Fortran.

7.2 Source form

7.2.1 Applicability to language

Fortran has an obsolescent source form called “fixed” where blanks are not significant in parsing the source code, and a source form called “free” where blanks are significant. A famous example of the vulnerability associated with fixed source form is

```
do 25 i = 1.10
```

being interpreted as an assignment of 1.1 to the (undeclared) floating point variable `do25i` instead of as the loop header

```
do 25 i = 1,10
```

In addition, fixed source form ignores text beyond line position 72, whereas for free form code, all characters within the legal line length are significant (except beyond the character !). The vulnerability associated with fixed form source code is that any text placed beyond line position 72 is ignored, which can change the semantics. Numerous additional vulnerabilities are associated with fixed form source.

7.2.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can:

- Avoid fixed source form in all programs;
- Use `implicit none` to require that all variables are declared, see 6.17 Choice of clear names [NAI]

7.3 Unformatted files

7.3.1 Applicability to language

In Fortran unformatted output of a variable or expression, the internal representation of its value is written exactly as it stands to the storage medium and can be read back directly with neither roundoff nor conversion overhead into a variable of the same type, type parameters, and shape. If the variable is a pointer, for defined behaviour, it must be associated with a target and the value of the target is written; when read back the target must have the shape of the target that was written. If the variable is allocatable, it must be allocated; when read back it must be allocated and have the shape of the variable that was written. The variable is not permitted to be of a type with an ultimate component that is allocable or a pointer, unless a user-defined derived type I/O procedure has been provided. If these prerequisites are not satisfied, program behaviour is undefined. In particular, if the file is read within a program execution other than the one in which it was written, there is a danger that incorrect values will be obtained, or that the reading program runs out of data prematurely.

7.3.2 Avoidance mechanisms for language users

Fortran software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can, when using an unformatted file:

- Ensure that the properties of each variable read exactly match those of the variable or expression that was written.
- Limit access of unformatted files to the same computer system, the same compiler, and the same compiler options unless it is guaranteed that the same internal representations are in use.

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, *Information technology – Microprocessor Systems – Floating-Point arithmetic*

[9] ISO/IEC 8652, *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.

[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