Date: 2021-11-05

ISO/IEC/JTC 1/SC 22/WG 23 N1346

ISO/IEC WD 24772-2

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

# Programming languages — Avoiding vulnerabilities in programming languages – Part 2: Vulnerability descriptions for the programming language Ada

*Langages de programmation — Conduite pour éviter les vulnérabilités dans les langages de programmation — Partie 2: Description des vulnérabilités pour le langage de programmation Ada*



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

## Contents



Deleted: vi
Deleted: vii
Deleted: 8
Deleted: 8
Deleted: 8
Deleted: 13
Deleted: 14
Deleted: 14
Deleted: 21
Deleted: 21
Deleted: 21
Deleted: 22
Deleted: 22
Deleted: 23
Deleted: 24
Deleted: 25
Deleted: 25
Deleted: 25
Deleted: 25
Deleted: 26
Deleted: 26
Deleted: 26
Deleted: 27
Deleted: 27
Deleted: 27
Deleted: 28
Deleted: 29
Deleted: 29
Deleted: 30
Deleted: 30
Deleted: 30
Deleted: 32
Deleted: 32
Deleted: 33
Deleted: 34
Deleted: 34
Deleted: 35

Deleted: 35
Deleted: 35
Deleted: 36
Deleted: 37
Deleted: 37
Deleted: 38
Deleted: 39
Deleted: 39
Deleted: 40
Deleted: 40
Deleted: 41
Deleted: 41
Deleted: 42
Deleted: 42
Deleted: 43
Deleted: 43
Deleted: 44
Deleted: 44
Deleted: 45
Deleted: 45
Deleted: 45
Deleted: 46
Deleted: 46
Deleted: 47
Deleted: 47
Deleted: 47
Deleted: 48
Deleted: 49
Deleted: 50
Deleted: 51
Deleted: 51
Deleted: 51
Deleted: 52
Deleted: 52
Deleted: 53
Deleted: 54
Deleted: 54
Deleted: 54
Deleted: 55

Deleted: 56

Deleted: 58

## 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 24772-2, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

This document replaces ISO IEC TR 24772-2:2020. The main changes between this document and the previous version are that material has been added for some vulnerabilities to reflect addition knowledge gained since the publication of TR 24772-2:2020.

## Introduction

This document is part of a series of documents that describe how vulnerabilities arise in programming languages. ISO/IEC 24772-1 addresses vulnerabilities that can arise in any programming language and hence is language independent. The other parts of the series are dedicated to individual languages.

This document provides guidance for the programming language Ada, so that application developers considering Ada or using Ada will be better able to avoid the programming constructs that can lead to vulnerabilities in software written in the Ada language and their attendant consequences. This document can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that could lead to vulnerabilities in their software. This Document can also be used in comparison with companion Documents and with the language-independent report, ISO/IEC 24772-1, *Information Technology – Programming Languages— Avoiding vulnerabilities in programming languages,* 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 document can only describe those that have been found, characterized, and determined to have sufficient probability and consequence.

# Information Technology — Programming Languages — Avoiding vulnerabilities in programming languages – Part 2: Vulnerability descriptions for the programming language Ada

Deleted: Guidance to

## 1. Scope

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

Vulnerabilities described in this Document record the way that the vulnerability described in the language-independent document ISO/IEC ISO/IEC 24772-1:2022 are manifested in Ada.

## 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 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 24772-1:2022, *Programming languages - Avoidance mechanisms for avoiding vulnerabilities in programming languages - Part 1: Language-independent guidance*

ISO/IEC 8652:2022 Programming languages – Programming language Ada

## 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 TR 24772-1, and the following apply. Other terms are defined where they appear in *italic* type.

**3.1.1 abnormal state**
state of an object whose initialization or assignment has been disrupted by an abort or the failure of a language-defined check

**3.1.2 access-to-object**
pointer to an object.

**3.1.3 access-to-subprogram**
pointer to a subprogram (function or procedure).

**3.1.4 access type**
type for objects that designate (point to) objects or subprograms

Note: This is often called a pointer type in other languages.

**3.1.5 access value**
value of an access type that is either null or designates another object or subprogram

**3.1.6 allocator**
construct that allocates storage from the heap or from a storage pool

**3.1.7 aspect specification**
mechanism used to specify assertions about the behaviour of subprograms, types and objects as well as operational and representational attributes of various kinds of entities

**3.1.8 atomic**
characteristic of a volatile object that guarantees that every access to the object is an indivisible access to the entity in memory

**3.1.9 attribute**
characteristic of a declared entity that can be queried by special syntax to return a value corresponding to the requested attribute

**3.1.10 bit ordering**
implementation defined value that is either *High_Order_First* or *Low_Order_First* that permits the specification or query of the way that memory bits are numbered within a representation clause

**3.1.11 bounded error**
error that need not be detected either prior to or during execution, but if not detected falls within a bounded range of possible effects

**3.1.12 case statement**
statement that provides multiple paths of execution dependent upon the value of the selecting expression, but which will have only one of the alternative sequences selected

**3.1.13 case expression**
expression that provides multiple paths of execution dependent upon the value of the selecting expression, but which will have only one of the alternative dependent expressions evaluated

**3.1.14 case choices**
alternatives defined in the case statement or case expression which are required to be of the same type as the type of the selecting expression in the case statement or case expression, and by which all possible values of the selecting expression must be covered

**3.1.15 compilation unit**
smallest Ada syntactic construct that can be submitted to the compiler, and that is usually held in a single compilation file

**3.1.16 configuration pragma**
directive to the compiler that is used to select partition-wide or system-wide options and that applies to all compilation units appearing in the compilation or all future compilation units compiled into the same environment

**3.1.17 controlled type**
type descended from the language-defined type controlled or limited_controlled which is a specialized type in Ada where the declarer can tightly control the initialization, assignment, and finalization of objects of the type

**3.1.18 dead store**
assignment to a variable that is not used in subsequent instructions

**3.1.19 default expression**
expression that is used to initialize a component, formal object, or formal parameter when an explicit expression, actual object, or actual parameter is not provided

**3.1.20 discrete type**
integer type or enumeration type

**3.1.21 discriminant**
parameter for a composite type that is used at elaboration of each object of the type to configure the object

**3.1.22 endianness**
byte ordering

**3.1.23 enumeration representation clause**
clause used to specify the internal codes for enumeration literals

**3.1.24 enumeration type**
discrete type defined by an enumeration of its values, which are named by identifiers or character literals, including the types Character and Boolean

**3.1.25 erroneous execution**
unpredictable result of an execution arising from an error that is not bounded by the language, but that need not be detected by the implementation either prior to or during run-time

**3.1.26 exception**
mechanism to detect an exceptional situation and to initiate processing dedicated to recover from the exceptional situation

Note: Exceptions are raised explicitly by user code or implicitly by language-defined checks.

**3.1.27 expanded name**
name that is disambiguated from other identical names by prepending the name with the name of the enclosing scope

Note: For example, the name of an entity E within a **`package`** (or any other named enclosing entity) P is expanded or disambiguated by using the alternate name P.E instead of the simple name E

**3.1.28 fixed-point types**
real-valued types with a specified error bound (called the 'delta' of the type) that provide arithmetic operations carried out with fixed precision rather than the relative precision of floating-point types

**3.1.29 generic formal subprogram**
parameter to a generic package used to specify a subprogram or operator

**3.1.30 hiding**
process where a declaration can be *hidden*, either from direct visibility, or from all visibility, within certain parts of its scope

**3.1.31 homograph**
property of two declarations such that they have the same name, and do not overload each other according to the rules of the language

**3.1.32 identifier**
simplest form of a name.

**3.1.33 idempotent behaviour**
behaviour that is a property of an operation that has the same effect whether applied just once or multiple times

**3.1.34 implementation defined**
defined by a set of possible effects of a construct where the implementation can choose to implement any effect in the set of effects

Deleted: may

**3.1.35 invalid representation**
representation of an object that does not represent any valid value of the object's subtype

**3.1.36 modular type**
integer type with values in the range 0.. modulus – 1 with wrap-around semantics for arithmetic operations, bit-wise "and" and "or" operations, and when defined in package Interfaces, arithmetic and logical shift operations

**3.1.37 obsolescent feature**
language feature that has been declared to be obsolescent or deprecated and which is documented in Annex J of ISO/IEC 8652

**3.1.38 operational and representation attributes**
values of certain implementation-dependent characteristics obtained by querying the applicable attributes and possibly specified by the user

**3.1.39 overriding indicator**
indicator that specifies the intent that an operation does or does not override ancestor operations by the same name, and used by the compiler to verify that the operation does (or does not) override an ancestor operation

**3.1.40 partition**
part of a program that consists of a set of library units such that each partition is permitted to execute in a separate address space, possibly on a separate computer, and can execute concurrently with and communicate with other partitions

Deleted: may

**3.1.41 pointer**
access object or access value

**3.1.42 pragma**
a directive to the compiler

**3.1.43 range check**
run-time check that ensures the result of an operation is contained within the range of allowable values for a given type or subtype, such as the check done on the operand of a type conversion.

**3.1.44 record representation clause**
a mechanism to specify the layout of components within records, that is, their order, position, and size

**3.1.45 scalar type**
any one of numeric, Boolean, enumeration, character and access types

**3.1.46 selecting expression**
expression that is part of a case statement or a case expression and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of a discrete type

**3.1.47 static expression**
expression with statically known operands that are computed with exact precision by the compiler

**3.1.48 storage place attribute**
integer attributes that specify, for a component of a record, the component position and size within the record

Note: The storage place attributes are: `Position`, `First_Bit` and `Last_Bit`.

**3.1.49 storage pool**
named location in an Ada program where all objects of a single access type will be allocated

**3.1.50 storage subpool**
separately reclaimable subdivision of a storage pool that is identified by a subpool handle

**3.1.51 subtype declaration**
construct that allows programmers to declare a named entity that defines a possibly restricted subset of values of an existing type or subtype, typically by imposing a constraint, such as specifying a smaller range of values

**3.1.52 task**
separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks from the same program

**3.1.53 unused variable**
variable that is declared but neither read nor written to in the program

**3.1.54 volatile**
characteristic of an object that guarantees that updates to the object are always seen in the same order by all tasks, and all reads are directly from memory

Note: all atomic objects are volatile.

## 4 Using this document

ISO/IEC 24772-1:2022 subclause 4.2 documents the process of creating software that is safe, secure and trusted within the context of the system in which it is fielded. The Ada programming language was explicitly designed for safety, security and the early elimination of errors from Ada programs. Nevertheless, as this document shows, vulnerabilities exist in the Ada programming environment, and organizations are responsible for understanding and addressing the programming language issues that arise in the context of the real-world environment in which the program will be fielded.

Organizations following this document, in addition to meeting the requirements of subclause 4.2 of ISO/IEC 24772-1:

1. Identify and analyze weaknesses in the product or system, including systems, subsystems, modules, and individual components;

2. Identify and analyze sources of programming errors;
3. Determine acceptable programming paradigms and practices to avoid vulnerabilities using guidance drawn from clauses 5.3 and 6 in this document;
4. Determine avoidance and mitigation mechanisms using clause 6 of this document as well as other technical documentation;
5. Map the identified acceptable programming practices into coding standards;
6. Select and deploy tooling and processes to enforce coding rules or practices;
7. Implement controls (in keeping with the requirements of the safety, security and general requirements of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

Tool vendors follow this document by providing tools that diagnose the vulnerabilities described in this document. Tool vendors also document to their users those vulnerabilities that cannot be diagnosed by the tool.

Programmers and software designers follow to this document by following the architectural and coding guidelines of their organization, and by choosing appropriate mitigation techniques when a vulnerability is not avoidable.

# 5 General language concepts and primary avoidance mechanisms

## 5.1 General Ada language concepts

### 5.1.1 Ada language design

Ada has been designed with emphasis on software engineering principles that support the development of high-integrity applications. For example, Ada is strongly typed thereby preventing vulnerabilities associated with type mismatch. Similarly, Ada includes boundary checking on arrays as part of the standard language which prevents buffer overflow vulnerabilities. Most of the language can be used to develop applications without known vulnerabilities. Other views of avoiding programming mistakes and design flaws are addressed by [1], [2], [4], [24], [26] and [29]. For specific guidance regarding programming in safety and/or security environments see [5][6][11][12][25][28].

### 5.1.2 Enumeration type

The defining identifiers and defining character literals of an enumeration type are required to be distinct. The predefined order relations between values of the enumeration type follow the order of corresponding position numbers.

### 5.1.3 Exception

There is a set of predefined exceptions in Ada in **`package`** `Standard`: `Constraint_Error`, `Program_Error`, `Storage_Error`, and `Tasking_Error`; one of them is raised when certain

language-defined checks fail. The standard libraries also define several exceptions that are raised when checks in the libraries fail. User code can define, raise and handle exceptions explicitly.

### 5.1.4 Hiding

Where *hidden from all visibility*, a declaration is not visible at all (neither using a direct_name nor a selector_name). Where *hidden from direct visibility*, only direct visibility is lost; visibility using an expanded name is still possible.

### 5.1.5 Implementation defined

Implementations are required to document their behaviour in implementation-defined situations.

### 5.1.6 Type conversions

Ada uses a strong type system based on name equivalence rules. It distinguishes types, which embody statically checkable equivalence rules, and subtypes, which associate static or dynamic properties with types, for example, index ranges for array subtypes or value ranges for numeric subtypes. Subtypes are not types and their values are implicitly convertible to all other subtypes of the same type. All subtype and type conversions ensure by static or dynamic checks that the converted value is within the value range of the target type or subtype. If a static check fails, then the program is rejected by the compiler. If a dynamic check fails, then an exception `Constraint_Error` is raised.

To affect a transition of a value from one type to another, three kinds of conversions can be applied in Ada:

a) Implicit conversions: there are few situations in Ada that allow for implicit type conversions. An example is the assignment of a value of a type to a polymorphic variable of an encompassing class. In all cases where implicit type conversions are permitted, neither static nor dynamic type safety or application type semantics (see below) are endangered by the conversion.

b) Explicit conversions: various explicit conversions between related types are allowed in Ada. All such conversions ensure by static or dynamic rules that the converted value is a valid value of the target type. Violations of subtype properties cause an exception to be raised by the conversion.

c) Unchecked conversions: Conversions that are obtained by instantiating the generic subprogram `Unchecked_Conversion` are unsafe and enable all vulnerabilities mentioned in subclause 6.3 as the result of a breach in a strong type system. `Unchecked_Conversion` is occasionally needed to interface with type-less data structures, for example, hardware registers.

A guiding principle in Ada is that, with the exception of using instances of `Unchecked_Conversion`, no undefined semantics can arise from conversions and the converted value is a valid value of the target type.

### 5.1.7 Operational and Representation Attributes

Some attributes can be specified by the user; for example:

- `X'Alignment`: allows the alignment of objects on a storage unit boundary at an integral multiple of a specified value.
- `X'Size`: denotes the size in bits of the representation of the object.
- `X'Component_Size`: denotes the size in bits of components of the array type X.

### 5.1.8 User-defined types

Ada allows the usual user-defined types such as records, classes (called tagged records), or access types. In addition Ada allows for user-defined scalar types which permit specification of value ranges, value constraints, and for floating-point and fixed-point types, precision. More advanced typing capabilities allow the user to specify types for communicating concurrently executing entities (tasks) and for synchronized data structures (protected objects).

The typing rules of the language prevent intermixing of objects and values of distinct types.

### 5.1.9 Compiler directives

Note: Ada supports compiler directives in the form of aspect specifications, aspect clauses, and configuration pragmas. As an obsolescent feature, certain aspects can be specified by similarly named pragmas as well. We summarize below the aspects and configuration pragmas that are relevant to this document.

#### 5.1.9.1 **Aspect** `Atomic`

Specifies that all reads and updates of an object are indivisible.

#### 5.1.9.2 **Aspect** `Atomic_Components`

Specifies that all reads and updates of an element of an array are indivisible.

#### 5.1.9.3 **Aspect** `Convention`

Specifies that an Ada entity should use the conventions of another language.

#### 5.1.9.4 **Pragma** `Detect_Blocking`

A configuration pragma that specifies that all potentially blocking operations within a protected operation shall be detected, resulting in the `Program_Error` exception being raised.

#### 5.1.9.5 **Pragma** `Discard_Names`

Specifies that storage used at run-time for the names of certain entities, particularly exceptions and enumeration literals, can be reduced by removing name information from the executable image.

Deleted: may

#### 5.1.9.6 **Aspect** `Export`

Specifies an Ada entity to be accessed by a foreign language, thus allowing an Ada subprogram to be called from a foreign language, or an Ada object to be accessed from a foreign language.

#### 5.1.9.7 **Aspect** Import

Specifies an entity defined in a foreign language that then can be accessed from an Ada program, thus allowing a foreign-language subprogram to be called from Ada, or a foreign-language variable to be accessed from Ada.

#### 5.1.9.8 **Pragma** Normalize_Scalars:

A configuration pragma that specifies that an otherwise uninitialized scalar object is set to a predictable value, but out of range if possible.

#### 5.1.9.9 **Aspect** Pack

Specifies that storage minimization should be the main criterion when selecting the representation of a composite type.

#### 5.1.9.10 **Pragma** Restrictions

Specifies that certain language features are not to be used in a given application. For example, the **pragma** Restrictions (No_Obsolescent_Features) prohibits the use of any deprecated features. This **pragma** is a configuration pragma which means that all program units compiled into the library shall obey the restriction.

#### 5.1.9.11 **Pragma** Suppress

Specifies that a run-time check need not be performed because the programmer asserts it will always succeed.

#### 5.1.9.12 **Aspect** Unchecked_Union

Specifies an interface correspondence between a given discriminated type and some C union. The aspect, if True, specifies that the associated type will be given a representation that leaves no space for its discriminant(s).

#### 5.1.9.13 **Aspect** Volatile

Applicable to a type, an object, or a component, and specifies that the associated objects are volatile.

#### 5.1.9.14 **Aspect** Volatile_Components

Applicable to an array type or an array object, and specifies that the associated components are volatile.

### 5.1.10 Separate Compilation

Ada requires that calls on libraries are checked for invalid situations as if the called routine were part of the current compilation.

### 5.1.11 Storage Pool

A storage pool can be sized exactly to the requirements of the application by allocating only what is needed for all objects of a single type without using the centrally managed heap. Exceptions raised due to memory failures in a storage pool will not adversely affect storage allocation from other storage pools or from the heap. Storage pools for types whose values are of equal length do not suffer from fragmentation. Storage pools can be divided into subpools, to allow efficient reclamation of a portion of a storage pool.

The following Ada restrictions prevent the application from using allocators in various contexts:

**pragma** `Restrictions(No_Allocators)`**:** prevents the use of all allocators.

**pragma** `Restrictions(No_Standard_Allocators_After_Elaboration)`: prevents the use of allocators after the main program has commenced.

**pragma** `Restrictions(No_Local_Allocators)`: prevents the use of allocators except within expressions that are evaluated as part of library-unit elaboration.

**pragma** `Restrictions(No_Implicit_Heap_Allocations)`: prevents the implicit use of heap allocation by the Ada implementation, but allows explicit allocators.

**pragma** `Restrictions(No_Anonymous_Allocators)`: prevents the use of allocators having an anonymous type.

**pragma** `Restrictions(No_Access_Parameter_Allocators`): prevents the use of allocators as the actual parameter for an access parameter.

**pragma** `Restrictions(No_Coextensions`): prevents the use of allocators as the initial value for an access discriminant.

**pragma** `Default_Storage_Pool(`**`null`**`)`: specifies that no allocators are permitted for access types that do not specify their own Storage_Pool or Storage_Size.

**pragma** `Restrictions(No_Unchecked_Deallocations)`: prevents allocated storage from being deallocated and hence effectively enforces storage pool memory approaches or a completely static approach to access types. Storage pools are not affected by this restriction as explicit routines to free memory for a storage pool can be created

### 5.1.12 Unsafe programming

In recognition of the occasional need to step outside the type system or to perform "risky" operations, Ada provides clearly identified language features to do so. Examples include the generic `Unchecked_Conversion` for unsafe type conversions, `Unchecked_Deallocation` for the deallocation of heap objects regardless of the existence of surviving references to the object, and `Address_To_Access_Conversions` for converting addresses into access values. If unsafe programming is employed in a unit, then the unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units. Similarly, there are ways to create a potentially unsafe global pointer to a local object, using the `Unchecked_Access` attribute. A Restriction **`pragma`** can be used to disallow uses of these language-defined generic units, as well as `Unchecked_Access`. The **`pragma`** `Suppress` allows an implementation to omit certain run-time checks.

## 5.2 Primary avoidance mechanisms

The recommendations of this subclause are restatements of recommendations from clause 6 that have been identified as the most frequent or noteworthy recommendations from clause 6. Table 5.1 identifies the most relevant avoidance mechanisms to be used to prevent vulnerabilities in Ada.

In addition to the generic programming rules from ISO/IEC 24772-1:2022 subclause 5.4, additional rules from this subclause apply specifically to the Ada programming language. Clause 6 of this document provides Avoidance mechanisms to mitigate against known vulnerabilities in Ada.

Table 1: Primary avoidance mechanisms for software developers

| Number | Ada language users can … | Reference |
|---|---|---|
| 1 | Specify pre- and postconditions on subprograms. | 6.32 Passing parameters and return values [CSI], 6.34 Subprogram signature mismatch [OTR], 6.46 Argument passing to library functions [TRJ] |
| 2 | Avoid the use of the **`abort`** statement. | 6.56 Undefined behaviour [EWF], 6.60 Concurrency – Directed termination [CGT], 6.62 Concurrency – Premature termination [CGS] |
| 3 | Prohibit the use of features explicitly identified as unsafe, such as `Unchecked_Deallocation`, | 6.2 Type system [IHN], 6.3 Bit representation [STR], 6.11 Pointer type conversions [HFC], 6.14 Dangling reference |

Deleted: guidance to

Formatted: Subtitle, Centered

Deleted: Avoidance Mechanism

Deleted: 6.32 [CSJ]

Deleted: 6.34 [OTR]

Deleted: 6.46 [TRJ]

Deleted: 6.56 [EWF]

Deleted: 6.60 [CGT]

Deleted: 6.62 [CGS]

Deleted: Do not

Deleted: 6.2 [IHN]

Deleted: 6.3 [STR]

Deleted: 6.11 [HFC]

| | | |
|---|---|---|
| | `Unchecked_Conversion`, or `Unchecked_Access`, unless absolutely necessary and then with extreme caution. | to heap [XYK], 6.33 Dangling references to stack frames [DCM], 6.53 Provision of inherently unsafe operations [SKL], 6.56 Undefined behaviour [EWF], 6.3 Bit representation [STR], |
| 4 | Use user-defined types in preference to predefined types, including range and precision as needed. | 6.2 Type system [IHN], 6.4 Floating-point arithmetic [PLF], 6.6 Conversion errors [FLC], 6.57 Implementation-defined behaviour [FAB], |
| 5 | Protect all data shared between tasks within a protected object, or use Atomic operations to synchronize cooperating tasks | 6.3 Bit representation [STR], 6.56 Undefined behaviour [EWF], 6.61 Concurrent data access [CGX], |
| 6 | Exploit the type and subtype system of Ada, and pre- and postconditions, to express constraints on the values of parameters. | 6.46 Argument passing to library functions [TRI], |
| 7 | Whenever possible, the `'First`, `'Last`, and `'Range` attributes should be used for loop termination. If the `'Length` attribute has to be used, then extra care should be taken to ensure that the length expression considers the starting index value for the array. | 6.14 Dangling reference to heap [XYK], 6.30 Off-by-one error [XZH], |
| 8 | Use objects of controlled types to ensure that resources are properly released if a scope is exited prematurely. | 6.14 Dangling reference to heap [XYK], 6.22 Missing initialization of variables [LAV], 6.39 Memory leak and heap fragmentation [XYL], 6.60 Concurrency – Directed termination [CGT], 6.62 Concurrency – Premature termination [CGS], |

Deleted: 6.14 [XYK]

Deleted: 6.33 [DCM]

Deleted: 6.53 [SKL]

Deleted: 6.56 [EWF]

Deleted: 6.3 [STR]

Deleted: 6.2 [IHN]

Deleted: 6.4 [PLF]

Deleted: 6.6 [FLC]

Deleted: 6.57 [FAB]

Deleted: 6.3 [STR]

Deleted: 6.56 [EWF]

Deleted: 6.61 [CGX]

Deleted: 6.46 [TRJ]

Deleted: 6.14 [XYK]

Deleted: 6.30 [XZH]

Deleted: 6.14 [XYK]

Deleted: 6.22 [LAV]

Deleted: 6.39 [XYL]

Deleted: 6.60 [CGT]

Deleted: 6.62 [CGS]

| | | |
|---|---|---|
| 9 | Specify type invariants. | 6.44 Polymorphic variables [BKK], 6.46 Argument passing to library functions [TRJ] |
| 10 | Prohibit the suppression of checks provided by the language unless the absence of the errors checked against has been verified by static analysis tools. | 6.6 Conversion errors [FLC], 6.9 Unchecked array indexing [XYZ], 6.33 Dangling references to stack frames [DCM], 6.52 Suppression of language-defined run-time checking [MXB], 6.56 Undefined behaviour [EWF] |
| 11 | Use static analysis tools to detect erroneous or undefined behaviours and to preclude the raising of implicit exceptions. | 6.6 Conversion errors [FLC], 6.18 Dead store [WXQ], 6.19 Unused variable [YZS], 6.20 Identifier name reuse [YOW], 6.24 Side-effects and order of evaluation of operands [SAM], 6.25 Likely incorrect expression [KOA], 6.52 Suppression of language-defined run-time checking [MXB], 6.56 Undefined behaviour [EWF] |
| 12 | Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing. | 6.9 [XYZ], 6.10 Unchecked array copying [XYW], 6.30 Off-by-one error [XZH] |
| 13 | Include exception handlers for every task, so that their unexpected termination can be handled and possibly communicated to the execution environment. | 6.36 Ignored error status and unhandled exceptions [OYB], 6.60 Concurrency – Directed termination [CGT], 6.62 Concurrency – Premature termination [CGS] |

Deleted: 6.44 [BKK]
Deleted: 6.46 [TRJ]
Deleted: Do not
Deleted: the
Deleted: 6.6 [FLC]
Deleted: 6.9 [XYZ]
Deleted: 6.33 [DCM]
Deleted: 6.52 [MXB]
Deleted: 6.56 [EWF]
Deleted: 6.6 [FLC]
Deleted: 6.18 [WXQ]
Deleted: 6.19 [YZS]
Deleted: 6.20 [YOW]
Deleted: 6.24 [SAM]
Deleted: 6.25 [KOA]
Deleted: 6.52 [MXB]
Deleted: 6.56 [EWF]
Deleted: 6.10 [XYW]
Deleted: 6.30 [XZH]
Deleted: 6.36 [OYB]
Deleted: 6.60 [CGT]
Deleted: 6.62 [CGS]

| | | |
|---|---|---|
| 14 | For `case` statements and aggregates, do not use the `others` choice. | 6.5 Enumerator issues [CCB]<br>6.27 Switch statements and static analysis [CLL] |

**Table 5-1 Most relevant avoidance mechanisms to be used to prevent vulnerabilities**

These vulnerability guidelines can be categorized into several functional groups. Items 3, 10 and 11 are applicable to Exceptional and Erroneous Behaviours. Mitigation methods associated with Types, Subtypes, and Contracts include Items 1, 4, 6, and 9. Those techniques appropriate for Statements and Operations consist of Items 7, 12, and 14. Finally, Items 2, 5, and 8 are pertinent to Concurrency in applications.

# 6 Specific analysis for Ada

## 6.1 General

This subclause contains specific advice for Ada about the possible presence of vulnerabilities as described in ISO/IEC 24772-1:2022 [20] and provides specific guidance on how to avoid them in Ada code. This subclause mirrors ISO/IEC 24772-1:2022 clause 6 in that the vulnerability "Type System [IHN]" is found in subclause 6.2 of [20], and Ada specific guidance is found in subclause 6.2 in this document.

## 6.2 Type system [IHN]

### 6.2.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.2 applies to Ada.

Implicit conversions cause no application vulnerability, as long as the resulting exceptions are properly handled.

Assignment between types cannot be performed except by using an explicit conversion.

Failure to apply correct unit conversion factors when explicitly converting among types for different units will result in application failures due to incorrect values.

Failure to handle the exceptions raised by failed checks of dynamic subtype properties causes the execution of the whole system, a task, or an inner nested scope to halt abruptly.

Unchecked conversions circumvent the type system and therefore can cause unspecified behaviour (see 6.37 Type-breaking reinterpretation of data [AMV]).

### 6.2.2 Avoidance mechanisms for language users

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

Deleted: 6.5 [CCB]

Deleted: 6.27 [CLL]

Deleted: guidance

Deleted: 6.37 Type-breaking reinterpretation of data [AMV]

Formatted: hyper Char, Font: +Headings (Cambria), Font color: Blue

Deleted: Guidance

Deleted: to

Formatted: NormBull

They can:

- Apply the mitigation mechanisms of subclause 6.2.5 of ISO/IEC 24772-1:2022.
- Apply the predefined `'Valid` attribute for a given subtype to any value as needed to ascertain if the value is a valid value of the subtype. This is especially useful when interfacing with type-less systems or after `Unchecked_Conversion`.
- Consider restricting explicit conversions to the bodies of user-provided conversion functions that are then used as the only means to effect the transition between unit systems. Review these bodies critically for proper conversion factors.
- Handle exceptions raised by type and subtype conversions.
- Consider using the restriction `No_Dependence(Ada.Unchecked_Conversion)` to prevent circumventing the type system.

## 6.3 Bit representation [STR]

### 6.3.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerabilities described in ISO/IEC 24772-1 subclause 6.3 are mitigated by the type system in Ada.

The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in subclause 6.3 of ISO/IEC 24772-1.

Ada provides mechanism to individually access individual bits without having to individually count or mask neighbouring bits.

For the traditional approach to bit level programming, Ada provides modular types and literal representations in arbitrary base from 2 to 16 to deal with numeric entities and correct handling of the sign bit. The use of **`pragma`** `Pack` on arrays of Booleans provides a type-safe way of manipulating bit strings and eliminates the use of error-prone arithmetic operations.

### 6.3.2 Avoidance mechanisms for language users

Ada software developers can avoid the vulnerabilities associated with the complexity of bit level programming or mitigate its ill effects in the following ways. They can:

- Apply the mitigation mechanisms of subclause 6.3.5 of ISO/IEC 24772-1:2022.
- Use record and array types with the appropriate representation specifications added so that the objects are accessed by their logical structure rather than their physical representation. These representation specifications address order, position, and size of data components and fields.
- Query the default object layout chosen by the compiler to determine the expected behaviour of the final representation.
- Use the restriction `No_Dependence (Ada.Unchecked_Conversion)` to prevent circumventing the type system.

Formatted: Font: Cambria

Deleted: Follow

Deleted: Guidance to

Deleted:

Formatted: Font: Cambria

Formatted: NormBull

Deleted: In order to mitigate the vulnerabilities associated with the complexity of bit level programming¶
Follow

## 6.4 Floating-point arithmetic [PLF]

### 6.4.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1:2022 subclause 6.4 applies to Ada. Accuracy of data representation can be specified independently of any implementation characteristics. Ada provides binary and decimal fixed-point arithmetic as an alternative to floating points. Attributes are provided to access mantissa and exponents of values, thus reducing the need for bit manipulations. An implementation that conforms to the (optional) Annex G of the Ada standard provides guarantees on the accuracy of arithmetic operations and of the standard mathematical functions.

### 6.4.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull

Formatted: English (UK)

- Apply the mitigation mechanisms of subclause 6.4.5 of ISO/IEC 24772-1:2022.
- Rather than using predefined types, such as `Float` and `Long_Float`, whose precision can vary according to the target system, declare floating-point types that specify the required precision (for example, `digits 10`). Additionally, specifying ranges of a floating-point type enables constraint checks which prevents the propagation of infinities and NaNs.
- Forbid comparing floating-point values for equality. Instead, use comparisons that account for the approximate results of computations. Consult a numeric analyst when appropriate.
- Make use of static arithmetic expressions and static constant declarations when possible, since static expressions in Ada are computed at compile time with exact precision.
- Use Ada's standardized numeric libraries (for example, `Generic_Elementary_Functions`) for common mathematical operations (trigonometric operations, logarithms, and others).
- Use an Ada implementation that supports the Numerics Annex of ISO/IEC 8652 and employ the "strict mode" of that Annex in cases where additional accuracy requirements shall be met by floating-point arithmetic and the operations of predefined numerics packages, as defined and guaranteed by the Annex.
- Forbid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as `'Exponent`).
- In cases where absolute precision is needed, consider replacement of floating-point types and operations with fixed-point types and operations.

Deleted: Follow

Deleted: may

Deleted: Avoid

Deleted: Avoid

## 6.5 Enumerator issues [CCB]

### 6.5.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.5 applies to Ada.

Enumeration representation specifications are used to specify non-default representations of an enumeration type, for example when interfacing with external systems, or to confirm the default representation of a type. Ada specifies that all of the values in the enumeration type shall be defined

in the enumeration representation specification and that the numeric values of the representation shall preserve the original order. For example:

```
type IO_Types is (Null_Op, Open, Close, Read, Write, Sync);
for IO_Types use (Null_Op => 0, Open => 1, Close => 2,
 Read => 4, Write => 8, Sync => 16);
```

An array can be indexed by such a type. Ada does not prescribe the implementation model for arrays indexed by an enumeration type with non-contiguous values. Two options exist: Either the array is represented "with holes" and indexed by the values of the enumeration type, or the array is represented contiguously and indexed by the position of the enumeration value rather than the value itself. In the former case, the vulnerability described in subclause 6.5 of ISO/IEC 24772-1:2022 exists only if unsafe programming is applied to access the array or its components outside of (?) the protection of the type system. Within the type system, the semantics are well defined and safe. The vulnerability of unexpected but well-defined program behaviour upon extending an enumeration type exists in Ada. In particular, subranges or `others` choices in aggregates and case statements are susceptible to unintentionally capturing newly added enumeration values.

### 6.5.2 Avoidance mechanisms for language users

Deleted: Guidance to

Formatted: NormBull

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

- Apply the mitigation mechanisms of subclause 6.5.5 of ISO/IEC 24772-1:2022;
- For `case` statements and aggregates, do not use the `others` choice;
- For `case` statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.

Deleted: Follow

Deleted: .

Deleted: .

### 6.6 Conversion errors [FLC]

### 6.6.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.6 is mitigated by Ada.

Ada does not permit implicit conversions between different numeric types, hence cases of implicit loss of data due to truncation cannot occur as they can in languages that allow type coercion between types of different sizes.

- Ada permits the definition of subtypes of existing types that can impose a restricted range of values, and implicit conversions can occur for values of different subtypes belonging to the same type, but such conversions still involve range checks that prevent any loss of data or violation of the bounds of the target subtype.

In the case of explicit conversions, Ada language rules prevent numeric conversion errors by applying

- Range bound checks, which raise an exception if the operand of the conversion exceeds the bounds of the target type or subtype.

Precision is lost only on explicit conversion from a real type to an integer type or a real type of less precision.

As Ada permits a type distinction to be made among numeric or composite values in different unit systems, e.g., meters and feet, complex numbers or intervals of real numbers, explicit conversions between such types may not be consistent with application semantics for the types, unless accompanied with conversion factors.

On structured data, implicit conversions preserve all values. Explicit value conversions omit components not present in the target type where such differences are allowed in conversions. See in particular (implicit) upcasts and (explicit) downcasts for OOP in subclause 6.44 Polymorphic variables [BKK].

### 6.6.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.6.5 of ISO/IEC 24772-1:2022;
- Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets;
- Forbid range check suppression on conversions involving scalar types and subtypes to prevent generation of invalid data;
- Use static analysis tools during program development to verify that conversions cannot violate the range of their target.

Deleted: Guidance to

Formatted: NormBull

Formatted: English (UK)

Deleted: Follow

Deleted: .

Deleted: .

Deleted: Do not

Deleted: range checks

Deleted: .

### 6.7 String termination [CJM]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.7 does not apply to Ada.

Strings in Ada are not delimited by a termination character. Ada programs that interface to languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

### 6.8 Buffer boundary violation (buffer overflow) [HCB]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.8 does not apply to Ada (see 6.9 Unchecked array indexing [XYZ] and 6.10 Unchecked array copying [XYW]).

Formatted: Underline, Font color: Blue

Deleted: 6.9 Unchecked array indexing [XYZ]

Deleted: 6.10 Unchecked array copying [XYW]

Formatted: Underline, Font color: Blue

### 6.9 Unchecked array indexing [XYZ]

#### 6.9.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.9 does not apply to Ada.

All array indexing is checked automatically in Ada, and an Ada program raises an exception when indexes are out of bounds. This is checked in all cases of indexing, including when arrays are passed to subprograms.

An explicit suppression of the run-time checks can be requested by use of **`pragma`** `Suppress`, in which case the vulnerability would apply; however, such suppression is easily detected, and generally reserved for tight time-critical loops, even in production code.

#### 6.9.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull

Formatted: Font: Cambria, English (UK)

- Apply the mitigation mechanisms of subclause 6.9.5 of ISO/IEC 24772-1:2022;
- Use Ada's support for whole-array operations, such as for assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing;
- Write explicit bounds tests to prevent exceptions for indexing out of bounds.

Deleted: Follow

Deleted: .

Deleted: .

### 6.10 Unchecked array copying [XYW]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.10 does not apply to Ada.

Ada allows arrays to be copied by simple assignment (":="). The rules of the language ensure that no overflow can happen; instead, the exception `Constraint_Error` is raised if the target of the assignment is not able to contain the value assigned to it. The rules also ensure that overlapping source and target slices are handled correctly, i.e., the target slice receives the original value of the source slice. Since array copy is provided by the language, Ada does not provide unsafe functions to copy structures by address and length.

### 6.11 Pointer type conversions [HFC]

#### 6.11.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.11 does not apply to Ada. The mechanisms available in Ada to alter the type of a pointer value are unchecked type conversions and type conversions involving pointer types derived from a common root type. In addition, uses of the unchecked

address taking capabilities can create pointer values that misrepresent the true type of the designated entity (see subclause 13.10 of ISO/IEC 8652).

Checked type conversions that affect the application semantics adversely are possible. For example, when a pointer to a class-wide type is changed to a pointer to a specific type in the class, a run-time check is required.

### 6.11.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.11.5 of ISO/IEC 24772-1:2022;
- Forbid the use of features explicitly identified as unsafe.;
- Use `'Access` which is always type safe;
- Consider using the restriction `No_Dependence(Ada.Unchecked_Conversion)`, `No_Use_Of_Attribute(Address)`, `No_Specification_of_Aspect(Address)`, and `No_Unchecked_Access` to prevent circumventing the type system.

### 6.12 Pointer arithmetic [RVG]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.12 does not apply to Ada.

### 6.13 Null pointer dereference [XYH]

#### 6.13.1 Applicability to the language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.13 is mitigated by Ada. The vulnerability is mitigated by compile-time or run-time checks that ensure that no null value can be dereferenced. Any attempt to dereference a null pointer results in the `Constraint_Error` exception being implicitly raised. Vulnerabilities associated with unhandled exceptions are addressed in subclause 6.36.

#### 6.13.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.13.5 of ISO/IEC 24772-1:2022;
- Use non-null access types where possible;
- Handle exceptions raised by attempts to dereference null values.

### 6.14 Dangling reference to heap [XYK]

#### 6.14.1 Applicability to language

Deleted: Guidance to
Formatted: Font: Cambria
Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate
Formatted: Font: Cambria
Deleted: Follow
Deleted: .
Deleted: Do not
Deleted: the
Deleted:
Deleted: .
Deleted: Guidance to
Formatted: NormBull
Deleted: Follow
Deleted: .
Deleted: .

The vulnerability as described in ISO/IEC 24772-1 subclause 6.14 applies to Ada. Use of Unchecked_Deallocation can cause dangling references to the heap when this feature is used, since `Unchecked_Deallocation` can be applied even though there are outstanding references to the deallocated object.

Ada provides a model in which whole collections of heap-allocated objects can be deallocated safely, automatically and collectively when the scope of the root access type or the scope of any associated storage pool object ends.

For global access types, unless storage pools are used, allocated objects can only be deallocated through an instantiation of the generic procedure `Unchecked_Deallocation`.

### 6.14.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.14.5 of ISO/IEC 24772-1:2022;
- Use local access types where possible;
- Avoid `Unchecked_Deallocation` and apply the restriction `No_Unchecked_Deallocation` to enforce this;
- Use controlled types and reference counting;
- Consider the use of storage pools and subpools.

Deleted: Guidance to

Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate

Formatted: Font: Cambria

Deleted: Follow

Deleted: .

Deleted: .

Deleted: Consider not using

Deleted: ing

Deleted: .

Deleted: .

### 6.15 Arithmetic wrap-around error [FIF]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.15 does not apply to Ada as wrap-around arithmetic in Ada is limited to modular types. Arithmetic operations on such types use modulo arithmetic, and thus no such operation can create an invalid value of the type.

For non-modular arithmetic, Ada raises the predefined exception `Constraint_Error` whenever a wrap-around occurs but implementations are allowed to refrain from doing so when a correct final value is obtained. In Ada there is no confusion between logical and arithmetic shifts.

### 6.16 Using shift operations for multiplication and division [PIK]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.16 does not apply to Ada as shift operations in Ada are limited to the modular types declared in the standard package Interfaces, which are not signed entities.

### 6.17 Choice of clear names [NAI]

### 6.17.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.17 applies to Ada. There are two possible issues: the use of the identical name for different purposes (overloading) and the use of similar names for different purposes.

This vulnerability does not address overloading, which is covered in 6.20 Identifier name reuse [YOW].

Deleted: 6.20 Identifier name reuse [YOW]

The risk of confusion by the use of similar names can occur through:

- Mixed casing. Ada treats upper case and lower-case letters in names as identical. Thus, no confusion can arise through an attempt to use Item and ITEM as distinct identifiers with different meanings.
- Underscores and periods. Ada permits single underscores in identifiers and they are significant. Thus, `BigDog` and `Big_Dog` are different identifiers. But multiple underscores (which can be confused with a single underscore) are forbidden, thus `Big__Dog` is forbidden. Leading and trailing underscores are also forbidden. Periods are not permitted in identifiers at all.
- Singular/plural forms. Ada does permit the use of identifiers which differ solely in this manner such as `Item` and `Items`. However, Ada lets the programmer use the identifier Item for a single object of a `type T` and the identifier Items for an object denoting an array of items that is of a `type array (…) of T`. The use of `Item` where `Items` was intended or vice versa will be detected by the compiler because of the type violation and the program rejected so no vulnerability would arise.
- International character sets. Ada compilers strictly conform to the appropriate International Standard for character sets.
- Identifier length. All characters in an identifier in Ada are significant. Thus `Long_IdentifierA` and `Long_IdentifierB` are always different. An identifier cannot be split over the end of a line. The only restriction on the length of an identifier is that enforced by the line length and this is guaranteed by the language standard to be no less than 200.

Ada permits the use of names such as `X`, `XX`, and `XXX` (which can all be declared as integers) and a programmer can easily, by mistake, write `XX` where `X` (or `XXX`) was intended. Ada does not attempt to catch such errors.

Deleted: may

The use of the wrong name will typically result in a failure to compile so no vulnerability will arise. But, if the wrong name has the same type as the intended name, then an incorrect executable program will be generated.

The "incorrect executable" can also happen when the two confused names have different types, but occur in a context where the type does not matter, for example `X'Address` or `X'Size`, or in a context where the type matters but only leads to the selection of a different overloaded entity, for example `Foo(X)` can be legal for both Integer `X` and Boolean `X`, if `Foo` is overloaded for both types.

### 6.17.2 Avoidance mechanisms for language users

Deleted: Guidance to

Deleted:

Formatted: Font: Cambria

Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

They can:

- Apply the mitigation mechanisms of subclause 6.17.5 of ISO/IEC 24772-1:2022;
- Avoid the use of similar names to denote different objects of the same type.;
- Adopt a project convention for dealing with similar names;
- See the Ada Quality and Style Guide [1].

### 6.18 Dead store [WXQ]

#### 6.18.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.18 applies to Ada.

Ada compilers do exist that detect and generate compiler warnings for dead stores.

The error in ISO/IEC 24772-1 subclause 6.18.3 that the planned reader misspells the name of the store is possible but highly unlikely in Ada since the language specifies that all objects shall be declared and typed, and the existence of two objects with almost identical names and compatible types (for assignment) in the same scope would be readily detectable.

#### 6.18.2 Avoidance mechanisms for Language Users

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

- Apply the mitigation mechanisms of subclause 6.18.5 of ISO/IEC 24772-1;
- Use Ada compilers that detect and generate compiler warnings for dead stores;
- Use static analysis tools to detect such problems.

### 6.19 Unused variable [YZS]

#### 6.19.1 Applicability to language

The vulnerability as described in subclause 6.19 of ISO/IEC 24772-1 applies to Ada. Ada compilers exist that detect and generate compiler warnings for unused variables.

#### 6.19.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.19.5 of ISO/IEC 24772-1:2022;
- Avoid the declaration of variables of the same type with similar names; instead use distinctive identifiers and the strong typing of Ada (for example through declaring specific types as in

```
type Pig_Counter is range 0 .. 1000;
Pig : Pig_Counter;
```

rather than just

Formatted: Font: Cambria
Deleted: Follow
Deleted: .
Deleted:
Deleted: Guidance to
Formatted: NormBull
Deleted: Follow
Deleted: :2022
Deleted: .
Deleted: .
Deleted: Guidance to
Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate
Formatted
Deleted: Follow
Deleted: .
Deleted: Do not
Deleted: e
Deleted: . U
Deleted: se

```
   Pig: Integer;)
```
to reduce the number of variables of the same type;

- Use Ada compilers that detect and generate compiler warnings for unused variables.

## 6.20 Identifier name reuse [YOW]

### 6.20.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.20 applies to Ada. Ada is a language that permits local scope, and names within nested scopes can hide identical names declared in an outer scope. As such it is susceptible to the vulnerability. For subprograms and other overloaded entities the problem is reduced by the fact that hiding also takes the signatures of the entities into account. Entities with different signatures, therefore, do not hide each other.

Name collisions with keywords cannot happen in Ada because keywords are reserved.

The mechanism of failure identified in subclause 6.20.3 of ISO/IEC 24772-1:2022 regarding the declaration of non-unique identifiers in the same scope cannot occur in Ada because all characters in an identifier are significant.

### 6.20.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.20.5 of ISO/IEC 24772-1:2022;
- Use *expanded names* whenever confusion is possible;
- Use Ada compilers or static analysis tools that generate warnings for declarations in inner scopes that hide declarations in outer scopes.

## 6.21 Namespace issues [BJL]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.21 does not apply to Ada, since Ada does not attempt to disambiguate conflicting names imported from different packages. Instead, use of a name with conflicting imported declarations causes a compile-time error. The programmer can disambiguate the name usage by using an expanded name that identifies the exporting package.

## 6.22 Missing initialization of variables [LAV]

### 6.22.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.22 applies to Ada. As in many languages, it is possible in Ada to make the mistake of using the value of an uninitialized variable. However, as described below, Ada prevents some of the most harmful possible effects of using the value.

Deleted: .

Deleted: Guidance to

Deleted:

Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate, Tab stops: Not at 0 cm

Deleted: Follow

Deleted: .

Deleted: .

The vulnerability does not exist for pointer variables (or constants). Pointer variables are initialized to **`null`** by default, and every dereference of a pointer that is not null-excluding is checked for a **`null`** value.

The checks mandated by the type-system apply to the use of uninitialized variables as well. When the context for using a value imposes a subtype with a restricted set of values, then values of the type that are outside of the subtype will fail the check required in such contexts.
Use of an out-of-bounds value in most contexts raises an exception, regardless of the origin of the faulty value. (See 6.36 Ignored error status and unhandled exceptions [OYB] regarding exception handling.) In the case of values originating from an uninitialized variable that are not detected by such a subtype check (such as when the context does not impose a subtype constraint, the value is within the subtype's set of values, or the value does not belong to the type itself), execution can proceed with that value, but use of such values will not lead to out-of-bounds memory modifications. In particular, use of uninitialized values will not result in writing outside of the bounds of array objects, and will not lead to wild jumps when used as the selecting value of a **`case`** statement or **`case`** expression.

Deleted: may

For scalar types, the `Default_Value` aspect can be specified to provide a default initial value for otherwise uninitialized objects of the type.

For record types, default initializations can be specified as part of the type definition. For record types, aggregate values can be used to initialize an object to ensure that all components of the object have been initialized with a value.

For controlled types (those descended from the language-defined type `Controlled` or `Limited_Controlled`), the user can also specify an Initialize procedure which is invoked on all default-initialized objects of the type.

The **`pragma`** `Normalize_Scalars` can be used to ensure that scalar variables are always initialized by the compiler in a repeatable fashion. This **`pragma`** is designed to initialize variables to an out-of-range value if there is one, to avoid hiding errors.

Lastly, the user can query the validity of a given value. The expression `X'Valid` yields true if the value of the scalar variable `X` conforms to the subtype of `X` and false otherwise. Thus, the user can protect against the use of out-of-bounds uninitialized or otherwise corrupted scalar values.

### 6.22.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull

- Apply the mitigation mechanisms of subclause 6.22.5 of ISO/IEC 24772-1:2022.
- If the compiler has a mode that detects use before initialization, then enable this mode and treat any such warnings as errors.
- Where appropriate, specify explict initializations or default initializations.

Deleted: Follow
Deleted: .
Deleted: .
Deleted: i
Deleted: .

- Use the **pragma** `Normalize_Scalars` to cause out-of-range default initializations for scalar variables.
- Use the `'Valid` attribute to identify out-of-range scalar values caused by the use of uninitialized variables, without incurring the raising of an exception. Note that an implementation is permitted to raise an exception for an `Unchecked_Conversion` in this case.
- Consider avoiding "junk initialization" of variables, as initializing a variable with an inappropriate default value such as zero can result in hiding underlying problems, because the compiler or other static analysis tools will then be unable to detect that the variable has been used prior to receiving a correctly computed value.

Deleted: .

Deleted: ¶

Deleted: Common advice that should be

Deleted: avoided is to perform a

Deleted: .

Deleted: I

Formatted: List Paragraph, Space Before: 6 pt, After: 6 pt, Bulleted + Level: 1 + Aligned at: 0.63 cm + Indent at: 1.27 cm

## 6.23 Operator precedence and associativity [JCW]

### 6.23.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.23 applies to Ada. Since this vulnerability is about "incorrect beliefs" of programmers, there is no way to establish a limit to how far incorrect beliefs can go. However, Ada is less susceptible to that vulnerability than many other languages, since

- Ada only has six levels of precedence and associativity is closer to common expectations. For example, an expression like `A = B or C = D` will be parsed as expected, as `(A = B) or (C = D)`.
- Mixed logical operators are not allowed without parentheses, for example, "`A or B or C`" is valid, as well as "`A and B and C`", but "`A and B or C`" is not; the user must write "`(A and B) or C`" or "`A and (B or C)`".
- Assignment is not an operator in Ada.

### 6.23.2 Avoidance mechanisms for language users

Deleted: Guidance to

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can follow the mitigation mechanisms of subclause 6.23.5 of ISO/IEC 24772-1:2022.

Formatted: NormBull

Deleted: ¶

Deleted: F

## 6.24 Side-effects and order of evaluation of operands [SAM]

### 6.24.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.24 applies to Ada. There are no operators in Ada with direct side effects on their operands using the language-defined operations, especially not the increment and decrement operation. Ada does not permit multiple assignments in a single expression or statement, except in the case of initialization of multiple variables by a single expression. In this case, the declaration is equivalent to a sequence of initializing declarations placed in the order of the variables in the list.

There is the possibility though to have side effects through function calls in expressions where the function modifies globally visible variables or "**in out**" or "**out**" parameters. Ada disallows multiple uses of the same variable within a single expression if one or more of the uses are as "**in out**" or "**out**" parameters. Operators in Ada are functions with only "**in**" parameters, so, when defined by

the user, although they cannot modify their own operands, they can modify global state and therefore have side effects.

Ada allows the implementation to choose the order of evaluation of expressions with operands of the same precedence level, the order of association is left-to-right. The operands of a binary operation are also evaluated in an arbitrary order, as happens for the parameters of any function call. In the case of user-defined operators with side effects on global state, this implementation dependency can cause unpredictability of the side effects.

### 6.24.2 Avoidance mechanisms for language users

Deleted: Guidance to

Formatted: NormBull

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

- Apply the mitigation mechanisms of subclause 6.24.5 of ISO/IEC 24772-1:2022;
- Make use of one or more programming guidelines which prohibit functions that modify global state, and can be enforced by static analysis;
- Minimize use of "`in out`" and "`out`" parameters for functions;
- Always use brackets to indicate order of evaluation of operators of the same precedence level.

Deleted: Follow

Deleted: .

Deleted: .

Deleted: .

### 6.25 Likely incorrect expression [KOA]

### 6.25.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.25 applies to Ada. An instance of this vulnerability consists of two syntactically similar constructs such that the inadvertent substitution of one for the other can result in a program which is accepted by the compiler but does not reflect the intent of the author.

The examples given in subclause 6.25 of ISO/IEC 24772-1:2022 are not problems in Ada because of Ada's strong typing and because an assignment is not an expression in Ada.

In Ada, a type conversion and a qualified expression are syntactically similar, differing only in the presence or absence of a single character:

```
Type_Name (Expression) -- a type conversion
```
vs
```
Type_Name'(Expression) -- a qualified expression
```

Typically, the inadvertent substitution of one for the other results in either a semantically incorrect program which is rejected by the compiler or in a program which behaves in the same way as if the intended construct had been written. In the case of a constrained array subtype, the two constructs differ in their treatment of sliding (conversion of an array value with bounds `100 .. 103` to a subtype with bounds `200 .. 203` will succeed; qualification will fail a run-time check).

Similarly, a timed entry call and a conditional entry call with an else-part that happens to begin with a **delay** statement differ only in the use of "**else**" vs "**or**" (or even "**then abort**" in the case of an asynchronous_select statement).

Probably the most common correctness problem resulting from the use of one kind of expression where a syntactically similar expression should have been used has to do with the use of short-circuit vs. non-short-circuit Boolean-valued operations (for example, "**and then**" and "**or else**" vs "**and**" and "**or**"), as in

```
if (P /= null) and (P.all.Count > 0) then ... end if;
-- should have used "and then" to avoid dereferencing null
```

### 6.25.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull, Tab stops: Not at 7.94 cm

- Apply the mitigation mechanisms of subclause 6.25.5 of ISO/IEC 24772-1:2022;
- Consider using short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), though this can make it more difficult to express the distinction between the cases where short-circuited evaluation is known to be needed (either for correctness or for performance) and those where it is not.

Deleted: Follow

Deleted: .

## 6.26 Dead and deactivated code [XYQ]

### 6.26.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.26 applies to Ada. Ada allows the usual sources of dead code as described in subclause 6.26 of ISO/IEC 24772-1 and [22] that are common to most conventional programming languages.

In some cases, use `pragma`s such as `Restrictions`, `Suppress`, or `Discard_Names` to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example:

Moved (insertion) [1]

Formatted: Normal, No bullets or numbering

```
package Pkg is
   type Enum is (Aaa, Bbb, Ccc);
   pragma Discard_Names( Enum );
end Pkg;
```

If `Pkg.Enum'Image` and related attributes (e.g., `Value`, `Wide_Image`) of the type `Enum` are never used, and if the implementation normally builds a table of the enumeration literals, then the **pragma** allows the elimination of the table.

### 6.26.2 Avoidance mechanisms for language users

Deleted: Guidance to

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

Formatted: NormBull

They can:

- Apply the mitigation mechanisms of subclause 6.26.5 of ISO/IEC 24772-1:2022;
- Use implementation-specific mechanisms, if provided, to support the elimination of dead code.

## 6.27 Switch statements and static analysis [CLL]

### 6.27.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts) and the use of default cases, the vulnerability as described in ISO/IEC 24772-1 subclause 6.27 does not apply to Ada.

Ada ensures that a case statement or a case expression provide exactly one alternative for each value of the selecting expression's subtype. This restriction is enforced at compile time. An **others** choice can be used in the last alternative of a case statement or case expression to capture any remaining values of the selecting_expression subtype that are not covered by the preceding case choices. If the value of the expression is outside of the range of this subtype (e.g., due to an uninitialized variable), then the resulting behaviour is well-defined (if an others choice is present, that alternative is permitted to be be selected, otherwise `Constraint_Error` is raised). Control does not flow from one alternative to the next. Upon reaching the end of an alternative, control is transferred to the end of the **case** statement.

The remaining vulnerability is that unexpected values are captured by the **others** clause or a subrange as case choice. For example, when the range of the type `Character` was extended from 128 characters to the 256 characters in the Latin-1 character type, an **others** clause for a **case** statement with a Character type case expression originally written to capture cases associated with the 128 characters type now also captures the 128 additional cases introduced by the extension of the type `Character`. Some of the new characters needed to be covered by the existing case choices or new case choices.

### 6.27.2 Avoidance mechanisms for language users

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

- For **case** statements, **case** expressions and aggregates, avoid the use of the **others** choice;
- For **case** statements, **case** expressions and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition. [1]

[1] This case is somewhat specialized but is important, since enumerations are the one case where subranges turn *bad* on the user.

Deleted: Follow

Deleted: .

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

**Moved up [1]:** In some cases, use `pragma`s such as `Restrictions`, `Suppress`, or `Discard_Names` to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example:¶

```
package Pkg is¶
  type Enum is (Aaa, Bbb, Ccc);¶
  pragma Discard_Names( Enum );¶
end Pkg;¶
¶
```

If `Pkg.Enum'Image` and related attributes (e.g., `Value`, `Wide_Image`) of the type `Enum` are never used, and if the implementation normally builds a table of the enumeration literals, then the **pragma** allows the elimination of the table.

Deleted: may

Deleted: Guidance to

Formatted: NormBull

Deleted: .

### 6.28 Non-demarcation of control flow [EOJ]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.28 does not apply to Ada. The Ada syntax describes several types of compound statements that are associated with control flow including **`if`** statements, **`loop`** statements, **`case`** statements, **`select`** statements, and extended **`return`** statements. Each of these forms of compound statements require unique syntax that marks the end of the compound statement.

### 6.29 Loop control variable abuse [TEX]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.29 does not apply to Ada. Ada defines a **`for loop`** where the number of iterations is controlled by a loop control variable (called a loop parameter). This value has a constant view and cannot be updated within the sequence of statements of the body of the loop.

### 6.30 Off-by-one error [XZH]

#### 6.30.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.30 is mitigated by Ada.

**Confusion between the need for < and <= or > and >= in a test.**

A **`for`** … **`loop`** in Ada does not require the programmer to specify a conditional test for loop termination. Instead, the starting and ending value of the loop are specified which eliminates this source of off-by-one errors. There are also special **`for`** … **`loop`** structures that iterate through an entire array or container. These avoid the need to specify any bounds for the iteration. A **`while`** … **`loop`** however, lets the programmer specify the loop termination expression, which can be susceptible to an off-by-one error.

**Confusion as to the index range of an algorithm.**

Although there are language defined attributes to symbolically reference the start and end values for a loop iteration, the language does allow the use of explicit values and loop termination tests. Off-by-one errors can result in these circumstances.

Care should be taken when using the `'Length` attribute in the loop termination expression. The expression should generally be relative to the `'First` value.

The strong typing of Ada eliminates the potential for buffer overflow associated with this vulnerability. If the error is not statically caught at compile time, then a run-time check generates an exception if an attempt is made to access an element outside the bounds of an array.

**Failing to allow for storage of a sentinel value.**

Ada does not use language-defined sentinel values to terminate arrays. There is no need to account for the storage of a sentinel value, therefore this particular vulnerability concern does not apply to Ada.

### 6.30.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.30.5 of ISO/IEC 24772-1;
- Whenever possible, use a **`for`** … **`loop`** instead of a **`while`** … **`loop`**;
- Whenever possible, use Ada constructs that eliminate the need for loop statements, such as array aggregates, qualified expressions, and reduction expressions;
- Whenever possible, use the form of iteration that takes the name of the array or container and nothing more;
- When indices are necessary, use the `'First`, `'Last`, and `'Range` attributes for loop termination, e.g. **`for`** `I` **`in`** `MyArray'Range` **`loop`**…;
- If the `'Length` attribute is required to be used, take extra care to ensure that the index computation considers the starting index value for the array.

## 6.31 Unstructured programming [EWD]

### 6.31.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.31 applies to Ada. Ada programs can exhibit many of the vulnerabilities documented, such as leaving a **`loop`** at an arbitrary point, local jumps (**`goto`**), and multiple exit points from subprograms.

Ada however does not suffer from non-local jumps and multiple entries to subprograms.

### 6.31.2 Avoidance mechanisms for language users

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can follow the mitigation mechanisms of subclause 6.31.5 of ISO/IEC 24772-1:2022.

## 6.32 Passing parameters and return values [CSJ]

### 6.32.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.32 does not apply to Ada, except when parameter passing by reference is used. Ada employs the mechanisms (for example, modes **`in`**, **`out`** and **`in out`**) that are recommended in subclause 6.32 of ISO/IEC 24772-1:2022. These mode definitions are not optional, mode `in` being the default.

### 6.32.2 Avoidance mechanisms for language users

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways.

Deleted: Guidance to
Formatted: NormBull
Deleted: Follow
Deleted: :2022
Deleted: .
Deleted: .
Deleted: .
Deleted: .
Deleted: .
Deleted: Guidance to
Formatted: NormBull
Deleted: ¶
F
Formatted: Font: (Default) Cambria, 12 pt, Not Bold
Deleted: Guidance to
Formatted: Font: +Headings (Cambria)
Formatted: NormBull

They can follow the mitigation mechanisms of subclause 6.32.5 of ISO/IEC 24772-1:2022.

Deleted: Follow

### 6.33 Dangling references to stack frames [DCM]

#### 6.33.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.35 does not apply to Ada, except when the ‘`Address` or ‘`Unchecked_Access` attributes are used.

In Ada, the attribute `'Address` yields a value of some system-specific type that is not equivalent to a pointer. The attribute `'Access` provides an access value (what other languages call a pointer). Addresses and access values are not automatically convertible, although a predefined set of generic functions can be used to convert one into the other. Access values are typed, that is to say, they can only designate objects of a particular type or class of types.

As in other languages, it is possible to apply the `'Address` attribute to a local variable, and to make use of the resulting value outside of the lifetime of the variable. However, `'Address` is very rarely used in this fashion in Ada. Most commonly, programs use `'Access` to designate objects and subprograms, and the language enforces accessibility checks whenever code attempts to use this attribute to provide access to a local object outside of its scope. These accessibility checks eliminate the possibility of dangling references.

As for all other language-defined checks, accessibility checks can be disabled over any portion of a program by using **`pragma`** `Suppress`. The attribute `Unchecked_Access` produces values that are exempt from accessibility checks.

#### 6.33.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull

- Apply the mitigation mechanisms of subclause 6.33.5 of ISO/IEC 24772-1:2022;
- Only use the `'Address` attribute on static objects (for example, a register address);
- Forbid the use of `'Address` to provide indirect untyped access to an object;
- Forbid the conversion between `'Address` and access types;
- Use access types in all circumstances when indirect access is needed.
- Forbid the suppression of accessibility checks;
- Avoid use of the attribute `'Unchecked_Access`;
- Use 'Access attribute in preference to `'Address`;
- Consider applying the restriction `No_Use_Of_Attribute(Address)` to prohibit use of `'Address`;
- Consider applying the restriction `No_Unchecked_Access` to enforce that `'Unchecked_Access` is not used;

Deleted: Follow
Deleted: .
Deleted: .
Deleted: Do not
Deleted: .
Deleted: Do not
Deleted: convert
Deleted: .
Deleted: Do not
Deleted: .
Deleted: .
Deleted: .
Deleted: .
Deleted: .

### 6.34 Subprogram signature mismatch [OTR]

#### 6.34.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.34 applies to Ada.

There are two concerns identified with this vulnerability. The first is the corruption of the execution stack due to the incorrect number or type of actual parameters. The second is the corruption of the execution stack due to calls to externally compiled modules. Ada does not support variadic subprograms, which eliminates a common source for this vulnerability. The case of calls to libraries written in other languages is covered in 6.47.

In Ada, at compilation time, the parameter association is checked to ensure that the type of each actual parameter matches the type of the corresponding formal parameter. In addition, the formal parameter specification can include default expressions for a parameter. Hence, the procedure can be called with some actual parameters missing. In this case, if there is a default expression for the missing parameter, then the call will be compiled without any errors. If default expressions are not specified, then the procedure call with insufficient actual parameters will be flagged as an error at compilation time.

Caution is advised when specifying default expressions for formal parameters, as their use can result in successful compilation of subprogram calls with an unintended signature. The execution stack will not be corrupted in this event, but the program can be executing with unexpected values. The most appropriate use of default expressions is when, without them, there would end up being an overloading of the same name with fewer parameters that performed essentially the same operation. When calling externally compiled modules that are Ada program units, the type matching and subprogram interface signatures are monitored and checked as part of the compilation and linking of the full application. When calling externally compiled modules in other programming languages, additional steps are needed to ensure that the number and types of the parameters for these external modules are correct.

### 6.34.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate

Formatted

- Apply the mitigation mechanisms of subclause 6.34.5 of ISO/IEC 24772-1:2022;
- Minimize the use of default expressions for formal parameters.

Deleted: Follow

Deleted: .

### 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.35 is mitigated by Ada as the exception `Storage_Error` is raised when the recursive execution results in insufficient storage. It is also possible to use a recursion-depth counter to control recursive behavior.

### 6.35.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

- Apply the mitigation mechanisms of subclause 6.35.5 of ISO/IEC 24772-1:2022.
- If recursion is used, then use a `Storage_Error` exception handler to handle insufficient storage due to recursive execution.
- Use a recursion-depth counter to put a limit on recursion depth (for example raising an exception if the check fails).
- Consider using the asynchronous control construct to time the execution of a recursive call and to terminate the call if the time limit is exceeded.

## 6.36 Ignored error status and unhandled exceptions [OYB]

### 6.36.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.36 applies to Ada. Ada offers a set of predefined exceptions for error conditions that are detected by checks that are compiled into a program. In addition, the programmer can define exceptions that are appropriate for their application.

Exceptions are handled using an exception handler. Exceptions can be handled in the scope where the exception occurs, or they are propagated to an enclosing scope or the caller. However, exceptions that are not handled by a task body result in silent task termination. Similarly, exceptions that occur during the elaboration of a library-level package result in program termination.

### 6.36.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.36.5 of ISO/IEC 24772-1:2022.
- Use the result of the `'Valid` attribute to check for the validity of values delivered to an Ada program from an external device prior to use.
- Consider using the call
  `Ada.Task_Termination.Set_Dependents_Fallback_Handler`
  to install a handler that will be invoked whenever a task terminates, including due to exception propagation.
- Consider including an exception handler in the outermost block of the main subprogram and each task body to use as a last-chance exception handler.
- Document any exceptions that are expected to be propagated out of a given subprogram or the elaboration of a library-level package.

## 6.37 Type-breaking reinterpretation of data [AMV]

### 6.37.1 Applicability to language

Formatted: NormBull
Formatted
Deleted: Follow
Deleted: .
Deleted: .
Deleted: .
Deleted: Guidance to
Formatted: NormBull, Space After: 0 pt, Widow/Orphan control, Hyphenate
Formatted
Deleted: Follow
Deleted: .
Deleted: .
Deleted: .
Deleted: .

The vulnerability as described in ISO/IEC 24772-1 subclause 6.37 applies to Ada but only if the mechanisms of Unsafe Programming (5.1 Language concepts) are used.

`Unchecked_Conversion` can be used to bypass the type-checking rules, and its use is thus unsafe, as is its equivalent in any other language. The same applies to the use of `Unchecked_Union`, even though the language specifies various inference rules that the compiler shall use to catch statically detectable constraint violations. The fact that `Unchecked_Conversion` is a generic function that must be instantiated explicitly (and given a meaningful name) hinders its undisciplined use and places a loud marker in the code wherever it is used. Well-written Ada code will have a small set of instantiations of `Unchecked_Conversion`. Most implementations require the source and target types to have the same size in bits, to prevent accidental truncation or missing sign extensions.

Type reinterpretation is a universal programming need, and no usable programming language can exist without some mechanism that bypasses the type model. Ada provides these mechanisms with some additional safeguards, and makes their use purposely verbose, to alert the writer and the reader of a program to the presence of an unchecked operation.

### 6.37.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.37.5 of ISO/IEC 24772-1;
- Use `Unchecked_Union` only in multi-language programs that need to communicate data between Ada and C or C++. ;therwise the use of discriminated types prevents "punning" between values of two distinct types that happen to share storage;
- Forbid the use of the `Address` aspect or address clauses to obtain overlays, including avoiding `Address_to_Access_Conversions`. If the types of the objects are the same, then a renaming declaration is preferable. Otherwise, the `pragma Import` can be used to inhibit the initialization of one of the entities so that it does not interfere with the initialization of the other one.;
- Consider applying

```
pragma Restrictions (No_Specification_Of_Aspect => Unchecked_Union),
pragma Restrictions (No_Use_Of_Attribute => Address), and
pragma Restrictions (No_Dependence => System.Address_to_Access_Conversions)
```

to ensure this vulnerability cannot arise.

Deleted: Guidance to

Formatted: NormBull

Deleted: Follow

Deleted: :2022

Deleted: .

Deleted: O

Deleted: .

Deleted: Avoid

Deleted: using

Deleted: use

Deleted:

## 6.38 Deep vs. shallow copying [YAN]

### 6.38.1 Applicability to language

The vulnerability described in subclause 6.38 of ISO/IEC 24772-1 applies to Ada. It can be mitigated somewhat by language constructs that allow the creation of abstractions and the addition of user-defined copying operations, such that inadvertent aliasing problems can be contained within the abstraction. The default semantics of assignment create a shallow copy, when applied to the root of a graph structure.

### 6.38.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.38.5 of ISO/IEC 24772-1;
- Use controlled types and appropriate redefinitions of the `Initialize`, `Adjust`, and `Finalize` operation to create deep copies when needed;
- Use a pre-existing Container type for trees.

### 6.39 Memory leak and heap fragmentation [XYL]

### 6.39.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.39 applies to Ada. For objects that are allocated from the heap without the use of reference counting, the memory leak vulnerability is possible in Ada. For objects that allocate from a storage pool, the vulnerability is present but is restricted to this single pool, which makes it easier to detect memory leaks by verification. Subpools can be used to further reduce the possibility for memory leaks. For objects of a controlled type that uses referencing counting and that are not part of a cyclic reference structure, the vulnerability does not exist.

Ada ensures that objects designated by an access type declared in a nested scope are finalized when execution leaves the nested scope, however, it is implementation defined whether storage is reclaimed for this case. Associating an access type with a storage pool can ensure that the storage reclamation takes place.

Ada does not mandate the use of a garbage collector, but Ada implementations are free to provide such memory reclamation. For applications that use and return memory on an implementation that provides garbage collection, the issues associated with garbage collection exist in Ada.

### 6.39.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.39.5 of ISO/IEC 24772-1:2022;
- Use controlled types and reference counting to implement explicit storage management systems that cannot have storage leaks;
- Declare access types in a nested scope where possible;
- Consider the use of predefined container libraries where possible;
- Consider the use of user-defined storage pools and subpools;
- Use a completely static model where all storage is allocated from global memory and explicitly managed under program control.

Deleted: Guidance to
Formatted: NormBull
Deleted: Follow
Deleted: :2022
Deleted: .
Deleted: .
Deleted: Guidance to
Formatted: NormBull
Deleted: Follow
Deleted: .
Deleted: .
Deleted: .
Deleted: .
Deleted: .

### 6.40 Templates and generics [SYM]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.40 does not apply to Ada as the Ada generics model is based on imposing a contract on the structure and operations of the types that can be used for instantiation. Also, explicit instantiation of the generic is required for each particular type.

Therefore, the compiler is able to check the generic body for programming errors, independently of actual instantiations. At each actual instantiation, the compiler will also check that the instantiated type meets all the requirements of the generic contract.

Ada also does not allow for 'special case' generics for a particular type, therefore behaviour is consistent for all instantiations.

### 6.41 Inheritance [RIP]

#### 6.41.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1 subclause 6.41 applies to Ada.

Ada allows only a restricted form of multiple inheritance, where only one of the multiple ancestors (the parent) is permitted to implement operations. All other ancestors (interfaces) can only specify the operations' signature, and whether the operation is required to be overridden, or can simply do nothing if never explicitly defined. Therefore, Ada does not suffer from multiple inheritance related vulnerabilities.

Ada has no preference rules to resolve ambiguities of calls on primitive operations of tagged types. Hence the related vulnerability documented in ISO/IEC 24772-1:2022 subclause 6.41 does not apply to Ada.

#### 6.41.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: NormBull

- Apply the mitigation mechanisms of subclause 6.41.5 of ISO/IEC 24772-1;
- Use the overriding indicators on potentially inherited subprograms to ensure that the intended set of operations are overridden, thus preventing the accidental redefinition or failure to redefine an operation of the parent;
- Specify **`aspect`** `Pre'Class` and **`aspect`** `Post'Class` aspects when a primitive operation is initially defined, to indicate the properties of inputs that any overridings shall accept, and the properties of outputs that any overridings shall produce.

Deleted: Follow

Deleted: :2022.

Deleted: .

### 6.42 Violations of the Liskov substitution principle or the contract model [BLP]

#### 6.42.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.42 applies to Ada. The vulnerabilities can be mitigated by the use of language concepts of specified/enforced pre- and postconditions of methods.

When defining one type as a descendant of another and overriding existing primitive operations of the ancestor type, the Liskov Substitution Principle (LSP) argues for ensuring that the important properties of the operations are preserved in the descendant types, according to the rules of *behavioural subtyping*. In Ada, this can be enforced by specifying these properties using the `Pre'Class` and `Post'Class` aspects when the operation is first defined, to define the relevant pre- and postconditions (respectively) which are to apply to the operations and any overridings. Run-time checks will be provided by the Ada implementation on all calls of these operations and their overridings, to verify that the inputs provided by the caller satisfy the required preconditions, and that the outputs produced by the operation satisfy the required postconditions. Ada allows these aspects to be refined in overridings, but only in ways that are consistent with LSP, meaning that the effective class-wide preconditions can only be relaxed in overridings, never made more stringent, and the effective class-wide postconditions can only be tightened, never made looser. This ensures that if a caller is reaching an operation of a descendant type while being only aware of the `Pre'Class` and `Post'Class` aspects of an ancestor operation, any input that satisfies the ancestor `Pre'Class` will still satisfy the descendant effective `Pre'Class`, and any output that satisfies the descendant effective `Post'Class` will also satisfy the ancestor's `Post'Class`.

### 6.42.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.42.5 of ISO/IEC 24772-1:2022.
- Specify `Pre'Class` and `Post'Class` for all primitive operations of tagged types.

## 6.43 Redispatching [PPH]

### 6.43.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.43 applies to Ada. The default behaviour of the relevant calls is non-dispatching in Ada, which is not subject to this vulnerability, but upon explicitly coding a redispatching call, this vulnerability can occur.

Ada distinguishes between a specific `type T` and a class-wide `type T'Class`. If dispatching is being performed within a routine on a particular formal parameter, it is preferable that the parameter be declared as class-wide to document this internal use of dispatching. Ada permits an explicit conversion from a specific type to a class-wide type to perform re-dispatching, but this should be avoided when possible, and documented explicitly when necessary.

### 6.43.2 Avoidance mechanisms for language users

Deleted: may
Deleted: Guidance to
Deleted: L
Deleted: Users
Formatted: NormBull
Deleted: Follow
Deleted: may
Deleted: Guidance to
Deleted: L
Deleted: Users

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

- Apply the mitigation mechanisms of subclause 6.43.5 of ISO/IEC 24772-1:2022.
- If redispatching is necessary, document the behaviour explicitly.

### 6.44 Polymorphic variables [BKK]

#### 6.44.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability described in ISO/IEC 24772-1 subclause 6.44 is mitigated by Ada as run-times checks identify faulty uses.

Ada checks all conversions to descendant tagged types (*downward* conversions) to be sure the run-time *tag* of the object being converted matches that of the target type, or one of its descendants. To avoid the failure of such a *tag check*, the programmer should use a class-wide membership test ("`Obj in Target'Class`") or rely on a dispatching call to perform the appropriate *downward* conversion implicitly.

Although conversions *up* to ancestors are always *structurally* safe (*upward* conversions), in that the ancestor has a subset of the data components of any descendant, a conversion to a *specific* (as opposed to *class-wide*) ancestor type can violate semantic requirements of the descendant type, particularly if the descendant type is a private extension of the ancestor and has certain desired relationships between components of the extension and those inherited from the ancestor. By specifying a Type_Invariant aspect on a private extension, the programmer can ensure that the semantic requirements of the private extension, as captured by the type invariant, are preserved across such conversions to an ancestor specific type, in that they are re-checked after the construct manipulating the upward conversion is complete.

#### 6.44.2 Avoidance mechanisms for language Users

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can follow the mitigation mechanisms of subclause 6.44.5 of ISO/IEC 24772-1:2022.

### 6.45 Extra intrinsics [LRM]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.45 does not apply to Ada. In Ada, all subprograms, whether intrinsic or not, belong to the same name space. Ada specifies that all subprograms shall be explicitly declared, and the same name resolution rules apply to all of them, whether they are predefined or user defined. If two subprograms with the same name and signature are visible (that is to say nameable) at the same place in a program, then a call using that name will be rejected as ambiguous by the compiler, and the programmer will have to specify (for example, by means of an expanded name) which subprogram is meant.

Formatted: NormBull

Deleted: Follow

Deleted: may

Deleted: Guidance to

Deleted: L

Deleted: Follow

## 6.46 Argument passing to library functions [TRJ]

### 6.46.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 applies to Ada. Ada parameters can have values precluded by preconditions of the called routine.

To the extent that the preclusion of values can be expressed as part of the type system of Ada, however, the preconditions are checked by the compiler statically or dynamically and thus are no longer vulnerabilities. For example, any range constraint on values of a parameter can be expressed in Ada by means of type or subtype declarations. Type violations are detected at compile time, subtype violations cause run-time exceptions. In addition, preconditions, postconditions, type invariants, and subtype predicates can be specified explicitly to express more complex restrictions to be observed by callers. These are checked at run-time depending on the Assertion_Policy in effect, and can be recognized by other static analysis tools as part of program verification.

### 6.46.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.46.5 of ISO/IEC 24772-1:2022.
- Exploit the type and subtype system of Ada to express restrictions on the values of parameters and results.
- Specify explicit preconditions and postconditions for subprograms wherever practical.
- Specify subtype predicates and type invariants for subtypes and private types when appropriate.
- Specify the exception raised or other response to values that do not satisfy the precondition.

## 6.47 Inter-language calling [DJS]

### 6.47.1 Applicability to Language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.47 is mitigated by Ada as Ada provides mechanisms to interface with common languages, such as C, C++, Fortran and COBOL, so that vulnerabilities associated with interfacing with these languages can be avoided.

### 6.47.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.47.5 of ISO/IEC 24772-1:2022.
- Use the inter-language methods and syntax specified by ISO/IEC 8652 when the routines to be called are written in languages that ISO/IEC 8652 specifies an interface with, including aspects `Import`, `Export`, and `Convention`.

Deleted: may

Deleted: Guidance to

Formatted: Normal

Deleted: Follow

Deleted: Guidance to

Deleted: L

Deleted: Users

Formatted: Normal

Deleted: Follow

- Use interfaces to the C programming language where the other language system(s) are not covered by ISO/IEC 8652, but the other language systems have interfacing to C.
- Make explicit checks on all return values from foreign system code artifacts, for example by using the `'Valid` attribute or by performing explicit tests to ensure that values returned by inter-language calls conform to the expected representation and semantics of the Ada application.
- Consider handling any exceptions raised in Ada code before returning to a routine from a foreign language, to prevent possible stack corruption if the foreign language cannot handle exceptions raised in Ada code.

Deleted: that may be

### 6.48 Dynamically-linked code and self-modifying code [NYY]

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability described in ISO/IEC 24772-1 subclause 6.48 does not apply to Ada as Ada does not support either dynamic linking or self-modifying code. The latter is possible only by exploiting other vulnerabilities of the language in the most malicious ways and even then it is still very difficult to achieve.

### 6.49 Library signature [NSQ]

#### 6.49.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.49 applies to Ada. Ada provides mechanisms to explicitly interface to modules written in other languages. Aspects `Import`, `Export`, and `Convention` permit the name of the external unit and the interfacing convention to be specified.

Even with the use of aspects `Import`, `Export`, and `Convention`, the vulnerabilities stated in subclause 6.49 of ISO/IEC 24772-1:2022 are possible. Names and number of parameters change under maintenance; calling conventions change as compilers are updated or replaced, or languages are used for which Ada does not specify a calling convention.

#### 6.49.2 Avoidance mechanisms for language users

Deleted: Guidance to

Deleted:

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can follow the mitigation mechanisms of subclause 6.49.5 of ISO/IEC 24772-1:2022.

Formatted: Space Before: 0 pt, After: 10 pt, Line spacing: Multiple 1.15 li

Deleted: ¶
F

### 6.50 Unanticipated exceptions from library routines [HJW]

#### 6.50.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.50 applies to Ada. Ada programs are capable of handling exceptions at any level in the program, as long as any exception naming and delivery mechanisms are compatible between the Ada program and the library components. In such cases the normal Ada exception handling processes will apply, and either the calling unit or some subprogram or task in its call chain will catch the exception and take appropriate programmed action. If no action is taken to handle the exception, the task or program where the exception occurred will terminate.

If the library convention is to report errors by means of error codes and not by exceptions, then if the library components themselves are written in Ada, then Ada's exception handling mechanisms let all called units trap any exceptions that are generated and return error codes instead.

If the interface between the Ada units and the library routine being called does not adequately address the issue of naming, generation and delivery of exceptions across the interface, then the vulnerabilities as expressed in subclause 6.50 of ISO/IEC 24772-1:2022 apply.

### 6.50.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.50.5 of ISO/IEC 24772-1:2022.
- Ensure that the interfaces with libraries written in other languages are compatible in the naming and generation of exceptions.
- Put appropriate exception handlers in all routines that call library routines, including the catch-all exception handler `when others =>`.
- Put appropriate exception handlers in all routines that are called by library routines, including the catch-all exception handler `when others =>`.
- Document any exceptions raised by any Ada units being used as library routines.

### 6.51 Pre-processor directives [NMP]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.51 does not apply to Ada as Ada does not have a pre-processor.

### 6.52 Suppression of language-defined run-time checking [MXB]

### 6.52.1 Applicability to Language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.52 applies to Ada. The Ada `pragma Suppress()` permits explicit suppression of language-defined checks on a unit-by-unit basis or on partitions or programs as a whole. (The language-defined default, however, is to perform the run-time checks that prevent run-time vulnerabilities.) `Pragma Suppress` can suppress all language-defined checks or individual categories of checks (see subclause 11.5 of ISO/IEC 8652).

### 6.52.2 Avoidance mechanisms for language users

Ada software developers can avoid the vulnerability or mitigate its ill effects in the following ways. They can apply the mitigation mechanisms of subclause 6.52.5 of ISO/IEC 24772-1:2022.

### 6.53 Provision of inherently unsafe operations [SKL]

### 6.53.1 Applicability to Language

Deleted: Guidance to
Deleted:
Formatted: Normal
Deleted: Follow
Deleted: that may be
Deleted: Guidance to
Deleted: L
Deleted: Users
Formatted: Space Before: 0 pt, After: 10 pt, Line spacing: Multiple 1.15 li
Deleted: ¶
F
Deleted: ollow

The vulnerability as described in ISO/IEC 24772-1 subclause 6.53 applies to Ada. In recognition of the occasional need to step outside the type-system or to perform "risky" operations, Ada provides clearly identified language features to do so. Examples include the generic `Unchecked_Conversion` for unsafe type conversions or `Unchecked_Deallocation` for the deallocation of heap objects regardless of the existence of surviving references to the object. If unsafe programming is employed in a unit, then the unit needs to specify the respective generic unit in its context clause, thus identifying potentially unsafe units. Similarly, there are ways to create a potentially unsafe global pointer to a local object, using the `Unchecked_Access` attribute.

### 6.53.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: Normal, Space After: 0 pt, Widow/Orphan control, Hyphenate, Tab stops: Not at 0 cm

Formatted

- Apply the mitigation mechanisms of subclause 6.53.5 of ISO/IEC 24772-1:2022.
- Forbid the use of unsafe programming practices and use the **`pragma`** `Restrictions()` to prevent the inadvertent use of unsafe language constructs.
- Carefully scrutinize any code that refers to a program unit explicitly designated to provide unchecked operations.

Deleted: Follow

Deleted: Avoid

Deleted: ,

### 6.54 Obscure language features [BRS]

### 6.54.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.54 applies to Ada. Ada is a rich language and provides facilities for a wide range of application areas. Because some areas are specialized, it is likely that a programmer not versed in a special area can misuse features for that area. For example, the use of tasking features for concurrent programming requires knowledge of this domain. Similarly, the use of exceptions and exception propagation and handling requires a deeper understanding of control flow issues than some programmers possess.

### 6.54.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

Formatted: Normal, Space After: 0 pt, Widow/Orphan control, Hyphenate, Tab stops: Not at 0 cm

Formatted

- Apply the mitigation mechanisms of subclause 6.54.5 of ISO/IEC 24772-1:2022.
- Use the **`pragma`** `Restriction()` to prevent the use of obscure features of the language.
- Avoid features in a Specialized Needs Annex of ISO/IEC 8652 unless the application area concerned is well-understood.
- Apply the restriction `No_Dependence` to prevent the use of specified pre-defined or user-defined libraries.

Deleted: Follow

Deleted: Similarly, a

Deleted: The

Deleted: s

### 6.55 Unspecified behaviour [BQF]

### 6.55.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.55 applies to Ada. In Ada, there are two main categories of unspecified behaviour, one having to do with unspecified aspects of normal run-time behaviour, and one having to do with *bounded errors*, errors that need not be detected at run-time but for which there is a limited number of possible run-time effects (though always including the possibility of raising `Program_Error` exception).

For the normal behaviour category, there are several distinct aspects of run-time behaviour that can be unspecified, including:

- Order in which certain actions are performed at run-time;
- Number of times a given element operation is performed within an operation invoked on a composite or container object;
- Results of certain operations within a language-defined generic package if the actual associated with a particular formal subprogram does not meet stated expectations (such as "<" providing a strict weak ordering relationship);
- Whether distinct instantiations of a generic or distinct invocations of an operation produce distinct values for tags or access-to-subprogram values.

The index entry in the ISO/IEC 8652 for *unspecified* provides the full list. Similarly, the index entry for *bounded error* provides the full list of references to places in ISO/IEC 8652 where a bounded error is described.

Failure can occur due to unspecified behaviour when the programmer did not fully account for the possible outcomes, and the program is executed in a context where the actual outcome was not one of those handled, resulting in the program producing an unintended result.

### 6.55.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.55.5 of ISO/IEC 24772-1:2022.
- For situations involving generic formal subprograms, ensure that the actual subprogram satisfies all of the stated expectations.
- For situations involving unspecified values, avoid depending on equality between potentially distinct values.
- For situations involving bounded errors, avoid the problem completely by ensuring in other ways that all requirements for correct operation are satisfied before invoking an operation that can result in a bounded error. See subclause 6.22 Initialization of variables [LAV] for a discussion of uninitialized variables in Ada, a common cause of a bounded error.

### 6.56 Undefined behaviour [EWF]

### 6.56.1 Applicability to language

Deleted: may

Deleted: Guidance to

Deleted:

Formatted: Normal

Deleted: Follow

Deleted: ,

The vulnerability as described in ISO/IEC 24772-1 subclause 6.56 applies to Ada. In Ada, undefined behaviour is called *erroneous execution*, and can arise from certain errors that are not required to be detected by the implementation, and whose effects are not in general predictable.

There are various kinds of errors that can lead to erroneous execution, including:

- Changing a discriminant of a record (by assigning to the record as a whole) while there remain active references to subcomponents of the record that depend on the discriminant;
- Referring via an access value, task id, or tag, to an object, **`task`**, or **`type`** that no longer exists at the time of the reference;
- Referring to an object whose assignment was disrupted by an **`abort`** statement, prior to invoking a new assignment to the object;
- Sharing an object between multiple tasks without adequate synchronization;
- Suppressing a language-defined check that is in fact violated at run-time;
- Specifying the address or alignment of an object in an inappropriate way;
- Using `Unchecked_Conversion`, `Address_To_Access_Conversions`, or calling an imported subprogram to create a value, or reference to a value, that has an invalid representation.

The full list is given in the index of ISO/IEC 8652 under *erroneous execution*.

Any occurrence of erroneous execution represents a failure situation, as the results are unpredictable, such as overwriting of memory, jumping to unintended locations within memory, and other uncontrolled events.

Deleted: and may involve

### 6.56.2 Avoidance mechanisms for language users

Deleted: Guidance to

Formatted: Normal

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

- Apply the mitigation mechanisms of subclause 6.56.5 of ISO/IEC 24772-1:2022.
- Ensure that all data shared between tasks are either contained within a protected object or marked **`atomic`**;
- Upon any use `of Unchecked_Deallocation`, carefully check to be sure that there are no remaining references to the object;
- Use **`pragma`** `Suppress` sparingly, and only after the code has undergone extensive verification.
- Be aware of other errors that can lead to erroneous execution are less common, but clearly in any given Ada application such as:
    - **`abort`**;
    - `Unchecked_Conversion`;
    - `Address_To_Access_Conversions`;
    - The results of imported subprograms;
    - Discriminant-changing assignments to global variables.

Deleted: Follow

Deleted: private

Deleted: The

Deleted: , care is required when using features

### 6.57 Implementation-defined behaviour [FAB]

### 6.57.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.57 applies to Ada. There are a number of situations in Ada where the language semantics are implementation defined, to allow the implementation to choose an efficient mechanism, or to match the capabilities of the target environment. Each of these situations is identified in Annex M of ISO/IEC 8652, and implementations are required to provide documentation associated with each item in Annex M to provide the programmer with guidance on the implementation choices.

A failure can occur in an Ada application due to implementation-defined behaviour if the programmer presumed the implementation made one choice, when in fact it made a different choice that affected the results of the execution. In many cases, a compile time message or a run-time exception will indicate the presence of such a problem. For example, the range of integers supported by a given compiler is implementation defined. However, if the programmer specifies a range for an integer type that exceeds that supported by the implementation, then a compile time error will be indicated, and if at run-time a computation exceeds the base range of an integer type, then `Constraint_Error` is raised.

As indicated above, many such failures are indicated by compile time error messages or run-time exceptions. However, there are cases where the implementation-defined behaviour can be silently misconstrued, such as if the implementation presumes `Ada.Exceptions.Exception_Information` returns a string with a particular format, when in fact the implementation does not use the expected format. If a program is attempting to extract information from `Ada.Exceptions.Exception_Information` for the purposes of logging propagated exceptions, then the log can result in misleading or useless information if there is a mismatch between the programmer's expectation and the actual implementation-defined format.

Deleted: may

Deleted: may

Deleted: end up with

Many implementation-defined limits have associated constants declared in language-defined packages, generally `package System`. In particular, the maximum range of integers is given by `System.Min_Int .. System.Max_Int`, and other limits are indicated by constants such as `System.Max_Binary_Modulus`, `System.Memory_Size`, `System.Max_Mantissa`, and similar. Other implementation-defined limits are implicit in normal '`First` and '`Last` attributes of language-defined (sub) types, such as `System.Priority'First` and `System.Priority'Last`. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.

### 6.57.2 Avoidance mechanisms for language users

Deleted: Guidance to

Formatted: Normal

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

- Apply the mitigation mechanisms of subclause 6.57.5 of ISO/IEC 24772-1:2022.
- Be aware of the contents of Annex M of ISO/IEC 8652 and avoid implementation-defined behaviour whenever possible.

Deleted: Follow

- Make use of the constants and subtype attributes provided in **package** `System` and elsewhere to avoid exceeding implementation-defined limits.
- Minimize use of any predefined numeric types, as the ranges and precisions of these are all implementation defined and instead, declare explicit numeric types to match the particular application needs.
- When there are implementation-defined formats for strings, such as `Exception_Information`, localize any necessary processing in packages with implementation-specific variants.

### 6.58 Deprecated language features [MEM]

#### 6.58.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 clause 6.58 applies to Ada. Ada has obsolescent features that can be used but provides a strong mitigation, in the form of the compilation **pragma** `Restrictions (No_Obsolescent_Features)` which prevents the use of any of these features.

#### 6.58.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.58.5 of ISO/IEC 24772-1:2022.
- Use **pragma** `Restrictions (No_Obsolescent_Features)` to prevent the use of any obsolescent features.
- Refer to Annex J of the ISO/IEC 8652 to determine whether a feature is obsolescent.

### 6.59 Concurrency – Activation [CGA]

#### 6.59.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.59 applies to Ada. Ada is open to this vulnerability but provides features for its mitigation. A task failing during activation will always raise an exception in the activating task (i.e., `Tasking_Error`). The activating task does not continue executing until all its dependent tasks have completed activation. A task can always check that another task has successfully activated.

#### 6.59.2 Avoidance mechanisms for language users

- Apply the mitigation mechanisms of subclause 6.59.5 of ISO/IEC 24772-1:2022.
- Provide a handler to catch activation failures of local tasks.
- If possible, declare all tasks statically at the library level and use language-provided mechanisms to verify successful activation.

### 6.60 Concurrency – Directed termination [CGT]

#### 6.60.1 Applicability to language

Deleted: .
Deleted: Instead
Deleted: your own
Deleted: your
Deleted: Guidance to
Deleted:
Formatted: Normal, Space After: 0 pt
Deleted: Follow
Deleted: Guidance to
Deleted: Follow
Deleted: ans

The vulnerability as described in ISO/IEC 24772-1 subclause 6.60 applies to Ada. Ada defines abort-deferred regions in which task termination will not occur. On a single processor, abort is defined to be immediate if the task is not in such a region. On multiprocessors, even if the abort is not immediate, it will be before any synchronization (dispatching) point.

Deleted: may not be

Deleted: but

### 6.60.2 Avoidance mechanisms for language users

Deleted: Guidance to

Deleted:

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

Formatted: Normal

- Apply the mitigation mechanisms of subclause 6.60.5 of ISO/IEC 24772-1:2022.
- Use the `'Terminated` and `'Callable` attributes to check that a task has terminated.
- Minimize the size of any abort-deferred region.
- Remove any possibility of unbounded loops in abort-deferred regions.
- Where possible, apply **`pragma`** `Restrictions (No_Abort_Statements)` to eliminate the use of this construct.

Deleted: Follow

### 6.61 Concurrent data access [CGX]

### 6.61.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.61 applies to Ada. Ada does allow tasks to access unprotected shared variables. However, the standard means of programming data that is shared between tasks is to use a protected object that enforces serial access. Atomic accesses on some simple types are supported (if supported by the hardware).

### 6.61.2 Avoidance mechanisms for language users

Deleted: Guidance to

Deleted:

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

Formatted: Normal

- Apply the mitigation mechanisms of subclause 6.61.5 of ISO/IEC 24772-1:2022.
- Prefer protected objects for shared data in preference to atomic, volatile or unmarked data.
- Statically determine that no unprotected data is used directly by more than one task.
- When shared variables are used, employ model checking or equivalent methodologies to prove the absence of race conditions.
- Use **`pragma`** `Atomic` and **`pragma`** `Atomic_Components` to ensure that all accesses to shared objects and components happen atomically.
- Use **`pragma`** `Volatile` and **`pragma`** `Volatile_Components` to ensure that all tasks see writes to the associated objects or array components in the same order.

Deleted: Follow

### 6.62 Concurrency – Premature termination [CGS]

### 6.62.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.62 applies to Ada. An Ada task can terminate silently, however in general the tasking model is robust and a number of features are available to mitigate against this vulnerability – see guidance below.

### 6.62.2 Avoidance mechanisms for language users

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

- Apply the mitigation mechanisms of subclause 6.62.5 of ISO/IEC 24772-1:2022.
- If possible, apply **`pragma`** `Restrictions (No_Abort_Statements)` to eliminate the use of this construct.
- Ensure that all tasks contain an exception handler at the outer level to prevent silent termination due to unhandled exceptions.
- Make use of **`package`** `Ada.Task_Termination` to force a handler to be executed when a task terminates.
- Use attributes `'Terminated` and `'Callable` to confirm that a task has not terminated (although care is needed here as a task can terminate immediately after this call is made).
- Ensure that all accesses and updates to data that are vulnerable to premature task termination are executed in abort-deferred regions (e.g., protected operations).
- Make use of timed task communication that will time-out if the called task does not respond.

### 6.63 Lock protocol errors [CGM]

### 6.63.1 Applicability to language

With the exception of unsafe programming (see 5.1 Language concepts), the vulnerability as described in ISO/IEC 24772-1 subclause 6.63 is mitigated by Ada. Locks are implicit in Ada protected objects, and explicit locks (like semaphores or mutexes) can be implemented by coding protected objects or tasks with explicit "lock" and "unlock" operations. For explicitly coded locks, any of the well-known lock protocol errors can occur. For the locks implicit in protected objects, protocol errors can occur in the following ways:

- By an "external" call, or "external" requeue, to a protected object that is already locked by the caller. A call or requeue to a protected object is "external" when the callee object is not statically known to be the "current object", which means that the call or requeue tries to acquire the implicit lock of the callee object.
- By directly or indirectly invoking any other potentially blocking operation, such as a delay statement, during a protected action (that is, from code executed in a protected object).
- By a call from a task that has a priority higher than the ceiling priority of the callee protected object, when locking is implemented by ceiling priorities (the `Ceiling_Locking` policy).

The first two cases, invoking potentially blocking operations, are by default bounded errors that are not required to be detected, neither at compile time nor at run-time. If not detected, the result can be deadlock or violation of mutual exclusion. Different implementations of Ada can behave

Deleted: Guidance to
Formatted: Normal
Deleted: Follow
Deleted: A
Deleted: should
Deleted: is
Deleted: may

differently and unpredictably. However, using the **pragma** `Detect_Blocking` forces a run-time check, which raises the `Program_Error` exception in case of failure. For the last case, ceiling priority violation, such a run-time check is always performed.

In general, whether an Ada program risks any of these errors can be determined only by a global analysis of the program, including the full caller-callee relationship. Such an analysis becomes much harder, and often impossible, if callees are defined dynamically by access values or if task priorities or ceiling priorities are modified dynamically.

### 6.63.2 Avoidance mechanisms for language users

Deleted: Guidance to

Deleted:

Formatted: Normal

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

- Apply the mitigation mechanisms of subclause 6.63.5 of ISO/IEC 24772-1:2022.
- Make use of loosely coupled communication using protected objects.
- Where possible stay within the constraints defined by the Ravenscar tasking profile [19][24].
- Use **pragma** `Detect_Blocking` to ensure blocking errors are detected.
- If synchronous communication (rendezvous) is employed, use model checking or equivalent to prove that the program is free from deadlocks etc.
- Always handle exceptions that can arrive from rendezvous or protected objects (unless they can be proved to not be raised).
- Guard against protocol failures by using timed communication, watchdog timers (programmed using Ada's timed events) and time-stamped data (using Ada's clock facilities).
- Forbid the use of unprotected shared data for synchronization between tasks.

Deleted: Follow

Deleted: Do not

### 6.64 Reliance on external format strings [SHL]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.63 does not apply to Ada, because Ada does not provide format strings.

### 6.65 Modifying constants [UJO]

#### 6.65.1 Applicability to language

The vulnerability described in ISO/IEC 24772-1 applies to Ada. Certain kinds of types in Ada permit the creation of a self-reference during object initialization, even for a constant. For such types (immutably limited and controlled types), the potential for the errors identified in this vulnerability exists, but there are various ways to mitigate this potential – see guidance below. With the exception of unsafe programming (see 5.1 Language concepts), this vulnerability is prevented in other cases by rules that prevent obtaining a reference with update access given a constant view of an object.

#### 6.65.2 Avoidance mechanisms for language users

Deleted: Guidance to

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

- Apply the mitigation mechanisms of subclause 6.65.5 of ISO/IEC 24772-1:2022.
- Forbid the use of the `Access` attribute to create a self-reference with update access when initializing an immutably limited type.
- Forbid the use of the `Unchecked_Access` attribute when it could create a self-reference with update access during an initialization routine, or the Adjust procedure of a controlled type.
- If a self-reference with update access is important to the functionality of a given (private) type, ensure that all primitive operations of the type use "in out" mode for parameters of the type, if they make any use of this self-reference to potentially update the parameter to ensure that constants are not inadvertently altered by such a primitive operation.

## 7 Language specific vulnerabilities for Ada

There are no language vulnerabilities specific to Ada documented in this document.

## 8 Implications for standardization

Future standardization efforts should consider the following items to address vulnerability issues identified earlier in this Annex:

- **`Pragma`** `Restrictions` is permitted to be extended to statically constrain dubious uses of control structures (see 6.31 Unstructured programming [EWD]).
- When appropriate, language-defined checks should be added to reduce the possibility of multiple outcomes from a single construct, such as by disallowing side-effects in cases where the order of evaluation can affect the result, similar to those specified for use of "**`in out`**" or "**`out`**" parameters of functions (see 6.24 Side-effects and order of evaluation [SAM] and 6.55 Unspecified behaviour [BQF]).
- When appropriate, language-defined checks should be added to reduce the possibility of erroneous execution, such as by disallowing unsynchronized access to shared variables (see 6.56 Undefined behaviour [EWF]).
- Language standards should specify relatively tight boundaries on implementation-defined behaviour whenever possible, and the standard should highlight what levels represent a portable minimum capability on which programmers can rely. For languages like Ada that allow user declaration of numeric types, the number of predefined numeric types should be minimized (for example, strongly discourage or disallow declarations of `Byte_Integer`, `Very_Long_Integer`, and similar, in **`package`** `Standard`) (see 6.57 Implementation-defined behaviour [FAB]).
- Ada can define `a` **`pragma`** `Restrictions` identifier No_Hiding that forbids the use of a declaration that result in a local homograph (see 6.20 Identifier name reuse [YOW]).
- Ada can add the ability to declare in the specification of a function that it is pure, that is, it has no side effects (see 6.24 Side-effects and order of evaluation of operands [SAM]).

Formatted: Normal

Deleted: Follow

Deleted: Do not

Deleted: Do not

Deleted: . This will

Deleted: This clause is intentionally left blank.

Deleted: may

Deleted: 6.31 Unstructured programming [EWD]

Formatted: Underline, Font color: Blue

Deleted: may

Formatted: Underline, Font color: Blue

Deleted: 6.55 Unspecified behaviour [BQF]

Deleted: 6.56 Undefined behaviour [EWF]

Formatted: Underline, Font color: Blue

Formatted: Underline, Font color: Blue

Deleted: 6.57 Implementation-defined behaviour [FAB]

Deleted: 6.20 Identifier name reuse [YOW]

Formatted: Underline, Font color: Blue

Deleted: 6.24 Side-effects and order of evaluation [SAM]

Formatted: Underline, Font color: Blue

- **`Pragma`** `Restrictions` can be extended to restrict the use of `'Address` attribute to library level static objects (see 6.33 Dangling references to stack frames [DCM]).
- Future standardization of Ada should consider implementing a language-provided reference counting storage management mechanism for dynamic objects (see 6.38 Deep vs. shallow copying [YAN]).

Deleted: 6.33 Dangling references to stack frames [DCM]

Formatted: Underline, Font color: Blue

## Bibliography


[1] AQSG, Ada Quality and Style Guide, Guidelines for Professional Programmers. Available from: https://en.wikibooks.org/wiki/Ada_Style_Guide.

[2] Barnes, John, *High Integrity Software - the SPARK Approach to Safety and Security*. Addison-Wesley. 2002.

[3] Barnes, John, Lecture Notes on Computer Science 8338, Ada 2012 Rationale: The Language—The Standard Libraries, Springer, 2013.

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

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

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

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

[8] 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

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

[10] 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.

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

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

[13] ISO 10241 (all parts), *International terminology standards*

[14] ISO/IEC Directives, Part 2, *Rules for the structure and drafting of International Standards*, 2017

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

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

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

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

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

[20] ISO/IEC 24772-1,– *Programming Languages— Avoidance mechanisms for avoiding vulnerabilities in programming languages – Part 1: Language independent guidelines*

[21] IEC/IEEE 60559:2011, *Information technology – Microprocessor Systems – Floating-Point arithmetic (3 parts)*

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

[23] 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.

[24] 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

[25] RTCA SC167 DO178-C/EUROCAE ED-12C, *Software Considerations in Airborne Systems and Equipment Certification*. Issued in the USA by the Requirements and Technical Concepts for Aviation) and in Europe by the European Organization for Civil Aviation Electronics. December 1992.

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

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

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

[29] 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.

Deleted: *Guidance to*

## Index