TECHNICAL REPORT

ISO/IEC TR 24772-2

First edition
2020-04

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 2: Ada

Langages de programmation — Conduite pour éviter les vulnérabilités dans les langages de programmation —

Partie 2: Ada

Reference number
ISO/IEC TR 24772-2:2020(E)

ISO copyright office
CP 401 • Ch. de Blandonnet 8
CH-1214 Vernier, Geneva
Phone: +41 22 749 01 11
Fax: +41 22 749 09 47
Email: copyright@iso.org
Website: www.iso.org

Published in Switzerland

Contents

Page

Foreword

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

The procedures used to develop this document and those intended for its further maintenance are described in the ISO/IEC Directives, Part 1. In particular, the different approval criteria needed for the different types of ISO documents should be noted. This document was drafted in accordance with the editorial rules of the ISO/IEC Directives, Part 2 (see www.iso.org/directives).

Attention is drawn to the possibility that some of the elements of this document may be the subject of patent rights. ISO shall not be held responsible for identifying any or all such patent rights. Details of any patent rights identified during the development of the document will be in the Introduction and/or on the ISO list of patent declarations received (see www.iso.org/patents).

Any trade name used in this document is information given for the convenience of users and does not constitute an endorsement.

For an explanation of the voluntary nature of standards, the meaning of ISO specific terms and expressions related to conformity assessment, as well as information about ISO's adherence to the World Trade Organization (WTO) principles in the Technical Barriers to Trade (TBT), see www.iso.org/iso/foreword.html.

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

This first edition cancels and replaces ISO/IEC TR 24772:2013, which has been split into several parts.

This document is intended to be used with ISO/IEC TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

A list of all parts in the ISO/IEC 24772 series can be found on the ISO website.

Any feedback or questions on this document should be directed to the user's national standards body. A complete listing of these bodies can be found at www.iso.org/members.html.

Introduction

This document provides guidance for the programming language Ada so that application developers considering Ada or using Ada can better avoid the programming constructs that lead to vulnerabilities in software written in the Ada language and their attendant consequences. This guidance can also be used by developers to select source code evaluation tools that can discover and eliminate some constructs that can lead to vulnerabilities in their software. This document can also be used in comparison with companion documents and with the language-independent ISO/IEC TR 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

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

Programming languages — Guidance to avoiding vulnerabilities in programming languages —

Part 2: Ada

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 document is applicable to the software developed, reviewed or maintained for any application.

Vulnerabilities described in this document present the way that the vulnerability described in ISO/IEC TR 24772-1 are manifested in Ada.

2 Normative references

The following documents are referred to in the text in such a way that some or all of their content constitutes requirements of this document. For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments) applies.

ISO/IEC 2382, *Information technology — Vocabulary*

ISO/IEC 8652, *Information technology — Programming languages — Ada*

ISO/IEC TR 24772-1, *Programming languages — Guidance to avoiding vulnerabilities in programming languages — Part 1: Language-independent guidance*

3 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, ISO/IEC 8652, ISO/IEC TR 24772-1 and the following apply.

ISO and IEC maintain terminological databases for use in standardization at the following addresses:

— ISO Online browsing platform: available at https://www.iso.org/obp

— IEC Electropedia: available at http://www.electropedia.org/

3.1
abnormal representation
representation of an object that is incomplete or that does not represent any valid value of the object's subtype

3.2
access-to-object
pointer to an object

3.3
access-to-subprogram
pointer to a subprogram (function or procedure)

3.4
access type
type for objects that designate (point to) objects or subprograms

Note 1 to entry: This is often called a *pointer* (3.38) type in other languages.

3.5
access value
value of an *access type* (3.4) that is either null or designates another object or subprogram

3.6
allocator
construct that allocates storage from the heap or from a *storage pool* (3.45)

3.7
aspect specification
mechanism used to specify assertions about the behaviour of subprograms, types and objects as well as operational and representational *attributes* (3.9) of various kinds of entities

3.8
atomic
characteristic of an object that guarantees that every access to an object is an indivisible access to the entity in memory instead of possibly partial, repeated manipulation of a local or register copy

3.9
attribute
characteristic of a declaration that can be queried by special syntax to return a value corresponding to the requested attribute

3.10
bit ordering
implementation defined (3.32) value that is either `High_Order_First` or `Low_Order_First` that permits the specification or query of the way that bits are represented in memory within a single memory unit

3.11
bounded error
error that does not need to be detected either prior to or during run time, but if not detected falls within a bounded range of possible effects

3.12
case statement
statement that provides multiple paths of execution dependent on the value of the selecting expression, but which have only one of the alternative sequences selected

Note 1 to entry: A selecting expression is understood to be an expression that is part of a case statement or *case expression* (3.13) and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.13
case expression
expression that provides multiple paths of execution dependent on the value of the selecting expression, but which have only one of the alternative dependent expressions evaluated

Note 1 to entry: A selecting expression is understood to be an expression that is part of a *case statement* (3.12) or case expression and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.14
case choice
alternative defined in the *case statement* (3.12) or *case expression* (3.13) 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 are required to be covered

Note 1 to entry: A selecting expression is understood to be an expression that is part of a case statement or case expression and that determines which choice is taken in executing the case statement or evaluating the case expression; it is of *discrete type* (3.19).

3.15
configuration pragma
directive to the compiler that is used to select *partition* (3.37)-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

Note 1 to entry: A compilation unit is understood to be the smallest Ada syntactic construct that can be submitted to the compiler, and that is usually held in a single compilation file.

3.16
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

3.17
dead store
assignment to a variable that is not used in subsequent instructions

3.18
default expression
expression of the formal object type that is used to initialize the formal object if an actual object is not provided

3.19
discrete type
integer type or *enumeration type* (3.23)

3.20
discriminant
parameter for a composite type that is used at elaboration of each object of the type to configure the object

3.21
endianness
byte ordering

3.22
enumeration representation clause
clause used to specify the internal codes for enumeration literals

3.23
enumeration type
discrete type (3.19) defined by an enumeration of its values, which are named by *identifiers* (3.31) or character literals, including the types character and boolean

3.24
erroneous execution
unpredictable result of an execution arising from an error that is not bounded by the language, but that does not need to be detected by the implementation either prior to or during run time

3.25
exception
mechanism to detect an exceptional situation and to initiate processing dedicated to recover from the exceptional situation

Note 1 to entry: Exceptions are raised explicitly by user code or implicitly by language-defined checks.

3.26
expanded name
name that is disambiguated from other identical names by prepending the name with the name of the enclosing scope

Note 1 to entry: 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.

3.27
fixed-point type
real-valued type 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

3.28
generic formal subprogram
parameter to a generic package used to specify a subprogram or operator

3.29
hiding
process where a declaration can be hidden, either from direct visibility, or from all visibility, within certain parts of its scope

3.30
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

3.31
identifier
simplest form of a name

3.32
implementation defined
defined by a set of possible effects of a construct where the implementation may choose to implement any effect in the set of possible effects

3.33
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

3.34
obsolescent feature
language feature that has been declared to be obsolescent or deprecated and documented in ISO/IEC 8652:2012, Annex J

3.35
operational and representation attribute
value of certain implementation-dependent characteristics obtained by querying the applicable *attributes* (3.9) and possibly specified by the user

3.36
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

3.37
partition
part of a program that consists of a set of library units such that each partition executes in a separate address space, possibly on a separate computer, and can execute concurrently with and communicate with other partitions

3.38
pointer
access object or *access value* (3.5)

3.39
pragma
directive to the compiler

3.40
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

3.41
record representation clause
mechanism to specify the layout of components within records, that is, their order, position, and size

3.42
scalar type
any one of numeric, Boolean, enumeration, character and *access types* (3.4)

3.43
static expression
expression with statically known operands that are computed with exact precision by the compiler

3.44
storage place attribute
integer *attribute* (3.9) that specify, for a component of a record, the component position and size within the record

Note 1 to entry: The storage place attributes are: `Position`, `First_Bit` and `Last_Bit`.

3.45
storage pool
named location in an Ada program where all objects of a single *access type* (3.4) are allocated

3.46
storage subpool
separately reclaimable subdivision of a storage pool that is identified by a subpool handle

3.47
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

3.48
task
separate thread of control that proceeds independently and concurrently between the points where it interacts with other tasks from the same program

3.49
unused variable
variable that is declared but neither read nor written to in the program

3.50
volatile
characteristic of an object that guarantees that updates to the object are always seen in the same order by all *tasks* (3.48)

Note 1 to entry: All *atomic* (3.8) objects are volatile.

4 Language concepts

4.1 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.

4.2 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. User code can define and raise exceptions explicitly.

4.3 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.

4.4 Implementation defined

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

4.5 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 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 effect 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 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 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 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.

4.6 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.

4.7 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.

4.8 Pragma compiler directives

NOTE Each of these pragmas specifies that the similarly named aspect of the type, object, or component denoted by its argument is either True (for parameterless pragmas) or is the value of the pragma parameter.

4.8.1 **Pragma** `Atomic`

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

4.8.2 **Pragma** `Atomic_Components`

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

4.8.3 **Pragma** `Convention`

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

4.8.4 **Pragma** `Detect_Blocking`

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

4.8.5 **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.

4.8.6 **Pragma** 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.

4.8.7 **Pragma** 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.

4.8.8 **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.

4.8.9 **Pragma** Pack

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

4.8.10 **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 need to obey the restriction.

4.8.11 **Pragma** Suppress

Specifies that a run-time check does not need to be performed because the programmer asserts it always succeeds.

4.8.12 **Pragma** Unchecked_Union

Specifies an interface correspondence between a given discriminated type and some C union. The pragma specifies that the associated type is given a representation that leaves no space for its discriminant(s).

4.8.13 **Pragma** Volatile

Applicable to a type, an object, or a component, and specifies that the associated objects are volatile, meaning that all updates to the objects are seen in the same order by all tasks.

4.8.14 **Pragma** Volatile_Components

Applicable to an array type or an array object, and specifies that the associated components are volatile, meaning that all updates to the components are seen in the same order by all tasks.

4.9 Separate compilation

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

4.10 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 do 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

4.11 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 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. A restriction pragma can be used to disallow uses of `Unchecked_Access`. The **pragma** `Suppress` allows an implementation to omit certain run-time checks.

5 General guidance for Ada

5.1 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.

5.2 Top 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 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 TR 24772-1:2019, 5.4, additional rules from this subclause apply specifically to the Ada programming language. Clause 6 provides guidance to mitigate against known vulnerabilities in Ada.

Table 1 — Most relevant avoidance mechanisms to be used to prevent vulnerabilities

Index	Avoidance mechanism	Subclause in this document
1	Specify pre- and postconditions on subprograms.	6.32 [CSJ], 6.34 [OTR], 6.46 [TRJ]
2	Avoid the use of the abort statement.	6.56 [EWF], 6.60 [CGT], 6.62 [CGS]
3	Do not use features explicitly identified as unsafe, such as `Unchecked_Deallocation`, `Unchecked_Conversion`, or `Unchecked_Access`, unless absolutely necessary and then with extreme caution.	6.2 [IHN], 6.3 [STR], 6.11 [HFC], 6.14 [XYK], 6.33 [DCM], 6.53 [SKL], 6.56 [EWF]
4	Use user-defined types in preference to predefined types, including range and precision as needed.	6.2 [IHN], 6.4 [PLF], 6.6 [FLC], 6.57 [FAB]
5	Protect all data shared between tasks within a protected object or mark the data Atomic.	6.3 [STR], 6.56 [EWF], 6.61 [CGX]
6	Exploit the type and subtype system of Ada to express (and post-conditions) on the values of parameters.	6.46 [TRJ]
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.	6.29 [TEX], 6.30 [XZH]
8	Use objects of controlled types to ensure that resources are properly released if a task terminates unexpectedly.	6.60 [CGT], 6.62 [CGS]
9	Specify type invariants.	6.44 [BKK], 6.46 [TRJ]
10	Do not suppress the checks provided by the language unless the absence of the errors checked against has been verified by static analysis tools.	6.6 [FLC], 6.9 [XYZ], 6.33 [DCM], 6.52 [MXB], 6.56 [EWF]
11	Use static analysis tools to detect erroneous or undefined behaviours and to preclude the raising of implicit exceptions.	6.6 [FLC], 6.18 [WXQ], 6.19 [YZS], 6.20 [YOW], 6.24 [SAM], 6.25 [KOA], 6.52 [MXB], 6.56 [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 [XYW], 6.30 [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 [OYB], 6.60 [CGT], 6.62 [CGS]
14	For **`case`** statements and aggregates, do not use the **`others`** choice.	6.5 [CCB], 6.27 [CLL]

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, 8, and 12 are pertinent to concurrency in applications.

6 Specific guidance for Ada

6.1 General

This clause contains specific advice for Ada about the possible presence of vulnerabilities as described in ISO/IEC TR 24772-1 and provides specific guidance on how to avoid them in Ada code. This subclause mirrors ISO/IEC TR 24772-1:2019, Clause 6 in that the vulnerability "Type System [IHN]" is found in ISO/IEC TR 24772-1:2019, 6.2, and Ada specific guidance is found in 6.2.

6.2 Type system [IHN]

6.2.1 Applicability to language

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 results 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 thread, or an inner nested scope to halt abruptly.

Unchecked conversions circumvent the type system and therefore can cause unspecified behaviour (see 6.37).

6.2.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.2.5.

— 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_Unchecked_Conversion` to prevent circumventing the type system.

6.3 Bit representation [STR]

6.3.1 Applicability to language

In general, the type system of Ada protects against the vulnerabilities outlined in ISO/IEC TR 24772-1:2019, 6.3. The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in ISO/IEC TR 24772-1:2019, 6.3.

6.3.2 Guidance to language users

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

— follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.3.5;

— use record and array types with the appropriate representation specifications (record representation clause, bit ordering, storage place attribute or pragma pack) added so that the objects are accessed by

their logical structure rather than their physical representation. These representation specifications address endianness, 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_Unchecked_Conversion` to prevent circumventing the type system.

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.

6.4 Floating-point arithmetic [PLF]

6.4.1 Applicability to language

Ada specifies adherence to the IEEE Floating Point Standards (IEEE 754, IEEE 854).

The vulnerability in Ada is as described in ISO/IEC TR 24772-1:2019, 6.4.2.

6.4.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.4.5.

— Rather than using predefined types, such as `Float` and `Long_Float`, whose precision may 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.

— Avoid 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 ISO/IEC 8652:2012, Annex G, and employ the "strict mode" of that annex in cases where additional accuracy requirements need to be met by floating-point arithmetic and the operations of predefined numerics packages, as defined and guaranteed by the annex.

— Avoid 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.

6.5 Enumerator issues [CCB]

6.5.1 Applicability to language

Enumeration representation clauses (specifications) are used to either 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 need to be defined in the enumeration representation specification and that the numeric values of the representation need to 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 ISO/IEC TR 24772-1:2019 6.5, exists only if unsafe programming is applied to access the array or its components outside 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 on 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.5.5.

— 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.

6.6 Conversion errors [FLC]

6.6.1 Applicability to language

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 the following.

— 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 6.44.

6.6.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.6.5.

— Use Ada's capabilities for user-defined scalar types and subtypes to avoid accidental mixing of logically incompatible value sets.

— Do not suppress range checks 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.

6.7 String termination [CJM]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable to Ada as 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 Clasue 4), this vulnerability is not applicable to Ada (see 6.9 and 6.10).

6.9 Unchecked array indexing [XYZ]

6.9.1 Applicability to language

All array indexing is checked automatically in Ada, and 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 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.9.5.

— 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.

6.10 Unchecked array copying [XYW]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable to Ada as 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

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 types that misrepresent the true type of the designated entity (see ISO/IEC 8652:2012, 13.10).

The vulnerabilities described in ISO/IEC TR 24772-1:2019, 6.11, exist in Ada only if unchecked type-conversions or unsafe taking of addresses are applied (see Clause 4). Other permitted type-conversions can never misrepresent the type of the designated entity.

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 leaf type, a run-time check is required.

6.11.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.11.5.

— Do not use the features explicitly identified as unsafe.

— Use `'Access` which is always type safe.

— Consider using the restriction `No_Unchecked_Conversion`, `No_Unchecked_Access` and `No_Use_Of_Attribute(Address)` to prevent circumventing the type system.

6.12 Pointer arithmetic [RVG]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable to Ada as Ada does not allow pointer arithmetic.

6.13 Null pointer dereference [XYH]

6.13.1 Applicability to the language

In Ada, this vulnerability is mitigated by compile-time or run-time checks that ensure that no null-value can be dereferenced. Of course, the `Constraint_Error` exception implicitly raised on such dereferencing needs to be handled or else the vulnerability of a failing system or components prevails.

6.13.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.13.5.

— Use non-null access-to-object 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

Use of `Unchecked_Deallocation` can cause dangling references to the heap. The vulnerabilities described in ISO/IEC TR 24772-1:2019, 6.14, exist in Ada 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-to-object types, unless storage pools are used, allocated objects can only be deallocated through an instantiation of the generic procedure `Unchecked_Deallocation`.

6.14.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.14.5.

— Use local access-to-object types where possible.

— Consider not using `Unchecked_Deallocation` and applying the restriction `No_Unchecked_Deallocation` to enforce this.

— Use controlled types and reference counting.

— Consider the use of storage pools and storage subpools.

6.15 Arithmetic wrap-around error [FIF]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable 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 Clause 4), this vulnerability is not applicable 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

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.

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 `item` that is of a `type array (…) of T`. The use of `Item` where `Items` was intended or vice versa is 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` (possibly declared to be of the same type) and a programmer can easily, by mistake, write `xx` where `x` (or `xxx`) was intended. Ada does not attempt to catch such errors.

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

6.17.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.17.5.

— 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[11].

6.18 Dead store [WXQ]

6.18.1 Applicability to language

This vulnerability exists in Ada as described in ISO/IEC TR 24772-1:2019, 6.18, with the exception that, in Ada, if a variable is read by a different thread (task) than the thread that wrote a value to the variable, it is not a dead store. Simply marking a variable as being `Volatile` is usually considered to be too error-prone for inter-thread (task) communication by the Ada community, and Ada has numerous facilities for safer inter thread communication.

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

The error in ISO/IEC TR 24772-1:2019, 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 need to 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.18.5.

— 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

This vulnerability exists in Ada as described in ISO/IEC TR 24772-1:2019, 6.19, although Ada compilers do exist that detect and generate compiler warnings for unused variables.

6.19.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.19.5.

— Do not declare variables of the same type with similar names. 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

```
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

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. Therefore, entities with different signatures do not hide each other.

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

The mechanism of failure identified in ISO/IEC TR 24772-1:2019, 6.20.3, 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.20.5.

— 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]

This vulnerability is not applicable to Ada because 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 Initialization of variables [LAV]

6.22.1 Applicability to language

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.

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. Use of an out-of-bounds value in relevant contexts causes an exception, regardless of the origin of the faulty value (see 6.36 regarding exception handling). Thus, the only remaining vulnerability is the potential use of a faulty but subtype-conformant value of an uninitialized variable, since it is technically indistinguishable from a value legitimately computed by the application.

For scalar types, the aspect specification `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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.22.5.

— 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 explicit initializations or default initializations.

— Use the **`pragma`** `Normalize_Scalars`**Error! Reference source not found.** 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 an implementation is permitted to raise an exception for an `Unchecked_Conversion` in this case.

Common advice that should be avoided is to perform a “junk initialization” of variables. 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 is then unable to detect that the variable has been used prior to receiving a correctly computed value.

6.23 Operator precedence/order of evaluation [JCW]

6.23.1 Applicability to language

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` is 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 has to write `"(A and B) or C"` or `"A and (B or C)"`;

— assignment is not an operator in Ada.

6.23.2 Guidance to language users

Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.23.5.

6.24 Side-effects and order of evaluation [SAM]

6.24.1 Applicability to language

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.

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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.24.5.

— 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.

6.25 Likely incorrect expression [KOA]

6.25.1 Applicability to language

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 ISO/IEC TR 24772-1:2019, 6.25, of 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` succeeds; qualification fails 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 (Ptr /= null) and (Ptr.all.Count > 0) then ... end if;
-- should have used "and then" to avoid dereferencing null
```

6.25.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.25.5.

— Consider using short-circuit forms by default (errors resulting from the incorrect use of short-circuit forms are much less common), thought 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.

6.26 Dead and deactivated code [XYQ]

6.26.1 Applicability to language

Ada allows the usual sources of dead code (described in ISO/IEC TR 24772-1:2019, 6.26, and Reference [22]) that are common to most conventional programming languages.

6.26.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.26.5.

— Use implementation-specific mechanisms, if provided, to support the elimination of dead code. In some cases, use `pragmas` 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 does not need to 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.

6.27 Switch statements and static analysis [CLL]

6.27.1 Applicability to language

With the exception of unsafe programming (see Clause 4) and the use of default cases, this vulnerability is not applicable to Ada as Ada ensures that a case statement, which only operates on discrete types, provides exactly one alternative for each value of the expression's subtype. This restriction is enforced at compile time. The `others` clause can be used as the last choice of a case statement to capture any remaining values of the case expression type that are not covered by the preceding case choices. If the value of the expression is outside of the range of this discrete subtype (e.g. due to an uninitialized variable), then the resulting behaviour is well-defined (`Constraint_Error` is raised). Control does not flow from one alternative to the next. On 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 Guidance to language users

— For **`case`** statements and aggregates, avoid the use of 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[1)].

6.28 Demarcation of control flow [EOJ]

This vulnerability is not applicable to Ada as 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 variables [TEX]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable to Ada as 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

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 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.

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

6.30.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.30.5.

— Whenever possible, use a **`for`** … **`loop`** instead of a **`while`** … **`loop`**.

— 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

Ada programs can exhibit many of the vulnerabilities noted in ISO/IEC TR 24772-1:2019, 6.31, leaving a **`loop`** at an arbitrary point, local jumps (**`goto`**), and multiple exit points from subprograms.

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

6.31.2 Guidance to language users

Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.31.5.

6.32 Passing parameters and return values [CSJ]

6.32.1 Applicability to language

Ada employs the mechanisms (for example, modes **`in`**, **`out`** and **`in out`**) that are recommended in ISO/IEC TR 24772-1:2019, 6.32. These mode definitions are not optional, mode **`in`** being the default. The remaining vulnerability is aliasing when a large object is passed by reference. In addition, Ada requires that a function result type be specified and the return value be assigned to the same type variable, making it much more obvious to the reader if a function result is not being used (see ISO/IEC 8652:2012, 6.5).

6.32.2 Guidance to language users

Follow avoidance advice in ISO/IEC TR 24772-1:2019, 6.32.5.

6.33 Dangling references to stack frames [DCM]

6.33.1 Applicability to language

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-to-object 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-to-object 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-object 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.33.5.

— Only use the `'Address` attribute on static objects (for example, a register address).

— Do not use `'Address` to provide indirect untyped access-to-object to an object.

— Do not convert between `'Address` and access-to-object or access-to-subprogram types.

— Use access types in all circumstances when indirect access is needed.

— Do not suppress 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.

6.34 Subprogram signature mismatch [OTR]

6.34.1 Applicability to language

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.

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 is compiled without any errors. If default expressions are not specified, then the procedure call with insufficient actual parameters is 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 incorrect signature. The execution stack is not 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.34.5.

— Minimize the use of default expressions for formal parameters.

— Manage interfaces between Ada program units and program units in other languages by using **pragma** `Import` to specify subprograms that are defined externally and **pragma** `Export` to specify subprograms that are used externally. These pragmas specify the imported and exported aspect

specifications of the subprograms, this includes the calling convention. All parameters need to be specified when using **pragma** `Import` and **pragma** `Export`.

— Use **pragma** `Convention` to identify when an Ada entity should use the calling conventions of a different programming language facilitating the correct usage of the execution stack when interfacing with other programming languages.

— Use the `'Valid` attribute to check if an object that is part of an interface with another language has a valid value for its type.

6.35 Recursion [GDL]

6.35.1 Applicability to language

Ada permits recursion. The exception `Storage_Error` is raised when the recurring execution results in insufficient storage.

6.35.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.35.5.

— If recursion is used, then use a `Storage_Error` exception handler to handle insufficient storage due to recurring execution.

— Use the asynchronous control construct to time the execution of a recurring call and to terminate the call if the time limit is exceeded.

— Consider applying
```
pragma Restriction (No_Recursion); or
pragma Restriction No_Reentrancy;
```
to eliminate this vulnerability.

6.36 Ignored error status and unhandled exceptions [OYB]

6.36.1 Applicability to language

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. These exceptions are handled using an exception handler. Exceptions can be handled in the environment where the exception occurs or they are propagated out to an enclosing scope.

6.36.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.36.5.

— 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 is invoked whenever a task terminates.

6.37 Type-breaking reinterpretation of data [AMV]

6.37.1 Applicability to language

`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 needs to use to catch statically detectable constraint violations. The fact that `Unchecked_Conversion` is a generic function that needs to 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 has 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.37.5.

— Use `Unchecked_Union` only in multi-language programs that need to communicate data between Ada and C or C++. Otherwise, the use of discriminated types prevents "punning" between values of two distinct types that happen to share storage.

— Avoid using address clauses to obtain overlays. If the types of the objects are the same, then a renaming declaration is preferable. Otherwise, use the `pragma Import` 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_Use_Of_Pragma(Unchecked_Union)),
pragma Restrictions (No_Use_Of_Aspect(Unchecked_Union)),
pragma Restrictions (No_Use_Of_Attribute(Address)), and
pragma Restrictions ()
```

to ensure this vulnerability cannot arise.

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

The vulnerability described in ISO/IEC TR 24772-1:2019, 6.38, of 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.38.5.

— 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

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-to-object 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-to-object 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.39.5.

— Use controlled types and reference counting to implement explicit storage management systems that cannot have storage leaks.

— Declare access-to-object types in a nested scope where possible.

— Use a completely static model where all storage is allocated from global memory and explicitly managed under program control.

6.40 Templates and generics [SYM]

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable 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 also checks 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 TR 24772-1:2019, 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 TR 24772-1:2019, 6.41, does not apply to Ada.

6.41.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.41.5.

— 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 the aspect specifications **aspect** `Pre'Class` and **aspect** `Post'Class` aspects when a primitive operation is initially defined, to indicate the properties of inputs that any overridings need to accept, and the properties of outputs that any overridings need to produce, as specified by ISO/IEC 8652:2012, 6.1.1.

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

6.42.1 Applicability to language

This vulnerability generally does apply to Ada but is mitigated by the language concepts of specified and 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 aspect specifications `Pre'Class` and `Post'Class` 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 are 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 aspect specifications 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` still satisfies the descendant effective `Pre'Class`, and any output that satisfies the descendant effective `Post'Class` also satisfies the ancestor's `Post'Class`.

6.42.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.42.5.

— Specify `Pre'Class` and `Post'Class` for all primitive operations of tagged types.

6.43 Redispatching [PPH]

6.43.1 Applicability to language

The default behaviour of the relevant calls is non-dispatching in Ada. But, on explicitly coding a redispatching call, this vulnerability applies.

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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.43.5.

— If redispatching is necessary, document the behaviour explicitly.

6.44 Polymorphic variables [BKK]

6.44.1 Applicability to language

The vulnerabilities related to upcasts apply to Ada.

The vulnerabilities related to unsafe casts do not apply to Ada, except when unsafe programming (see Clause 4) is used. The vulnerabilities related to downcasts are mitigated, 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 may 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 specification 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 Guidance to language users

Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.44.5.

6.45 Extra intrinsics [LRM]

The vulnerability does not apply to Ada, because all subprograms, whether intrinsic or not, belong to the same name space. Ada requires that all subprograms are 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 is rejected as ambiguous by the compiler, and the programmer needs to specify (for example, by means of an expanded name) which subprogram is meant (see ISO/IEC 8652:2012, 6.1).

6.46 Argument passing to library functions [TRJ]

6.46.1 Applicability to language

The general vulnerability from ISO/IEC TR 24772-1:2019 that parameters may have values precluded by preconditions of the called routine applies to Ada as well.

However, to the extent that the preclusion of values can be expressed as part of the type system of Ada, 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.46.5.

— Exploit the type and subtype system of Ada to express restrictions on the values of parameters and results.

— Specify explicit preconditions and postconditions**Error! Reference source not found.** for subprograms wherever practical.

— Specify subtype predicates and type invariants**Error! Reference source not found.** 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 applies to Ada, however 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.47.5.

— 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.

— 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 possibly 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.

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

With the exception of unsafe programming (see Clause 4), this vulnerability is not applicable to Ada as Ada supports neither dynamic linking nor 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

Ada provides mechanisms to explicitly interface to modules written in other languages. **`pragma`** `Import`, **`pragma`** `Export` and **`pragma`** `Convention` permit the name of the external unit and the interfacing convention to be specified.

Even with the use of **`pragma`** `Import`, **`pragma`** `Export` and **`pragma`** `Convention`, the vulnerabilities stated in ISO/IEC TR 24772-1:2019, 6.49, of 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 Guidance to language users

Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.49.5.

6.50 Unanticipated exceptions from library routines [HJW]

6.50.1 Applicability to language

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 apply, and either the calling unit or some subprogram or task in its call chain catches the exception and take appropriate programmed action. If no action is taken to handle the exception, the task or program where the exception occurred terminates.

If the library convention is to report 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 conditions instead. If such exception handling mechanisms are not put in place, then exceptions can be unexpectedly delivered to a caller.

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 ISO/IEC TR 24772-1:2019, 6.50, apply.

6.50.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.50.5.

— 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 possibly raised by any Ada units being used as library routines.

6.51 Pre-processor directives [NMP]

This vulnerability is not applicable 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 exists in Ada since `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 runtime checks that prevent the runtime vulnerabilities). `Pragma Suppress` can suppress all language-defined checks or 12 individual categories of checks (see ISO/IEC 8652:2012, 11.5).

6.52.2 Guidance to language users

Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.52.5.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to Language

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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.53.5.

— Avoid the use of unsafe programming practices.

— 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.

6.54 Obscure language features [BRS]

6.54.1 Applicability to language

Ada is a rich language and provides facilities for a wide range of application areas. Because some areas are specialized, it is possible that a programmer not versed in a special area misuses 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.54.5.

— Use the **`pragma`** `Restriction()` to prevent the use of obscure features of the language.

— Similarly, avoid features in ISO/IEC 8652:2012, Annexes C to H, unless the application area concerned is well-understood.

— The restriction `No_Dependence` prevents the use of specified pre-defined or user-defined libraries.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability to language

In Ada, there are two main categories of unspecified behaviour. One has to do with unspecified aspects of normal run-time behaviour. The other has to do with bounded errors, errors that do not need to 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 are 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.55.5.

— For situations involving generic formal subprograms, ensure that the actual subprogram satisfies all 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 6.22 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

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-to-object 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 abnormal 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, and may involve overwriting of memory, jumping to unintended locations within memory and other uncontrolled events.

6.56.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.56.5.

— Ensure that all data shared between tasks are either private within a protected object or marked Atomic.

— On 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. The other errors that can lead to erroneous execution are less common, but clearly in any given Ada application, care is required when using features such as:

  — `abort`;

  — `Unchecked_Conversion`;

  — `Address_To_Access_Conversions`;

  — the results of imported subprograms;

  — discriminant-changing assignments to global variables.

6.57 Implementation-defined behaviour [FAB]

6.57.1 Applicability to language

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 ISO/IEC 8652:2012, Annex M, and implementations are required to provide documentation associated with each item in ISO/IEC 8652:2012, 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 indicates 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 is indicated, and if at run time a computation exceeds the base range of an integer type, then `Constraint_Error` is raised.

Failure due to implementation-defined behaviour is generally due to the programmer presuming a particular effect that is not matched by the choice made by the implementation. 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 may 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 may end up with misleading or useless information if there is a mismatch between the programmer's expectation and the actual implementation-defined format.

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/aspect specification 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 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.57.5.

— Be aware of the contents of ISO/IEC 8652:2012, Annex M and avoid implementation-defined behaviour whenever possible.

— 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. Instead, declare your own numeric types to match your 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

If obsolescent language features are used, then the mechanism of failure for the vulnerability is as described in ISO/IEC TR 24772-1:2019, 6.58.3.

6.58.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.58.5.

— Use **pragma** Restrictions (No_Obsolescent_Features) to prevent the use of any obsolescent features.

— Refer to ISO/IEC 8652:2012, Annex J, to determine whether a feature is obsolescent.

6.59 Concurrency — Activation [CGA]

6.59.1 Applicability to language

Ada is open to this vulnerability but provides features for its mitigation. A task failing during activation always raises an exception in the activating task (e.g. 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 is executable (i.e. not terminated).

6.59.2 Guidance to language users

— Follow the mitigation mechanisms of ISO/IEC TR 24772-1:2019, 6.59.5.

— Always have a handler to catch activation failures.

— If possible, declare all tasks statically at the library level.