ISO/IEC JTC 1/SC 22/WG23 N1029

Date: 2021-01-18

ISO/IEC 24772–6

Notes on this document

This document is posted for analysis by SPARK experts to determine the scope of changes needed in the document for compatibility with the latest published SPARK specification.

Participants 4 Jan 2021:

Stephen Michell – Convenor

Rod Chapman – lead contributor

Paul Butcher – AdaCore

Erhard Ploedereder

Joyce Tokar – USA

Tullio Vardanega – Italy

Results of the meeting are captured in the tracked changes and Commented regions in this document.

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

Information Technology — Programming languages — Guidance to avoiding vulnerabilities in programming languages – Part 6 – Vulnerability descriptions for the programming language SPARK

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

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

Warning

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

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

**Copyright notice**

This ISO document is a working draft or committee draft and is copyright-protected by ISO. While the reproduction of working drafts or committee drafts in any form for use by participants in the ISO standards development process is permitted without prior permission from ISO, neither this document nor any extract from it may be reproduced, stored or transmitted in any form for any other purpose without prior written permission from ISO.

Requests for permission to reproduce this document for the purpose of selling it should be addressed as shown below or to ISO’s member body in the country of the requester:

*ISO copyright office*

*Case postale 56, CH-1211 Geneva 20*

*Tel. + 41 22 749 01 11*

*Fax + 41 22 749 09 47*

*E-mail copyright@iso.org*

*Web www.iso.org*

Reproduction for sales purposes may be subject to royalty payments or a licensing agreement.

Violators may be prosecuted.

Contents

[Foreword viii](#_Toc61769461)

[Introduction 10](#_Toc61769462)

[1. Scope 11](#_Toc61769463)

[2. Normative references 11](#_Toc61769464)

[3. Terms and definitions, symbols and conventions 12](#_Toc61769465)

[3.1 Terms and definitions 12](#_Toc61769466)

[4. Language concepts 12](#_Toc61769467)

[5. General guidance for SPARK 15](#_Toc61769468)

[6. Specific Guidance for SPARK Vulnerabilities 17](#_Toc61769469)

[6.1 General 17](#_Toc61769470)

[6.2 Type System [IHN] 17](#_Toc61769471)

[6.3 Bit Representations [STR] 18](#_Toc61769472)

[6.4 Floating-point Arithmetic [PLF] 19](#_Toc61769473)

[6.5 Enumerator Issues[CCB] 19](#_Toc61769474)

[6.6 Conversion Errors [FLC] 20](#_Toc61769475)

[6.7 String Termination [CJM] 20](#_Toc61769476)

[6.8 Buffer Boundary Violation [HCB] 20](#_Toc61769477)

[6.9 Unchecked Array Indexing [XYZ] 20](#_Toc61769478)

[6.10 Unchecked Array Copying [XYW] 21](#_Toc61769479)

[6.11 Pointer Type Conversions [HFC] 21](#_Toc61769480)

[6.12 Pointer Arithmetic [RVG] 21](#_Toc61769481)

[6.13 NULL Pointer Dereference [XYH] 21](#_Toc61769482)

[6.14 Dangling Reference to Heap [XYK] 21](#_Toc61769483)

[6.15 Arithmetic Wrap-around Error [FIF] 21](#_Toc61769484)

[6.16 Using Shift Operations for Multiplication and Division [PIK] 22](#_Toc61769485)

[6.17 Choice of Clear Names [NAI] 22](#_Toc61769486)

[6.18 Dead Store [WXQ] 23](#_Toc61769487)

[6.19 Unused Variable [YZS] 23](#_Toc61769488)

[6.20 Identifier Name Reuse [YOW] 24](#_Toc61769489)

[6.21 Namespace Issues [BJL] 24](#_Toc61769490)

[6.22 Initialization of Variables [LAV] 24](#_Toc61769491)

[6.23 Operator Precedence and Associativity [JCW] 24](#_Toc61769492)

[6.24 Side-effects and Order of Evaluation of Operands [SAM] 25](#_Toc61769493)

[6.25 Likely Incorrect Expression [KOA] 25](#_Toc61769494)

[6.26 Dead and Deactivated Code [XYQ] 26](#_Toc61769495)

[6.27 Switch Statements and Static Analysis [CLL] 27](#_Toc61769496)

[6.28 Demarcation of Control Flow [EOJ] 28](#_Toc61769497)

[6.29 Loop Control Variables [TEX] 28](#_Toc61769498)

[6.30 Off-by-one Error [XZH] 28](#_Toc61769499)

[6.31 Unstructured Programming [EWD] 29](#_Toc61769500)

[6.32 Passing Parameters and Return Values [CSJ] 30](#_Toc61769501)

[6.33 Dangling References to Stack Frames [DCM] 30](#_Toc61769502)

[6.34 Subprogram Signature Mismatch [OTR] 30](#_Toc61769503)

[6.35 Recursion [GDL] 31](#_Toc61769504)

[6.36 Ignored Error Status and Unhandled Exceptions [OYB] 31](#_Toc61769505)

[6.37 Type-breaking Reinterpretation of Data [AMV] 32](#_Toc61769506)

[6.38 Deep vs. Shallow Copying [YAN] 33](#_Toc61769507)

[6.39 Memory Leak and Heap Fragmentation [XYL] 33](#_Toc61769508)

[6.40 Templates and Generics [SYM] 34](#_Toc61769509)

[6.41 Inheritance [RIP] 34](#_Toc61769510)

[6.42 Violations of the Liskov Substitution Principle or the Contract Model [BLP] 35](#_Toc61769511)

[6.43 Redispatching [PPH] 35](#_Toc61769512)

[6.44 Polymorphic variables [BKK] 36](#_Toc61769513)

[6.45 Extra Intrinsics [LRM] 37](#_Toc61769514)

[6.46 Argument Passing to Library Functions [TRJ] 37](#_Toc61769515)

[6.47 Inter-language Calling [DJS] 38](#_Toc61769516)

[6.48 Dynamically-linked Code and Self-modifying Code [NYY] 38](#_Toc61769517)

[6.49 Library Signature [NSQ] 38](#_Toc61769518)

[6.50 Unanticipated Exceptions from Library Routines [HJW] 39](#_Toc61769519)

[6.51 Pre-processor Directives [NMP] 40](#_Toc61769520)

[6.52 Suppression of Language-defined Run-time Checking [MXB] 40](#_Toc61769521)

[6.53 Provision of Inherently Unsafe Operations [SKL] 40](#_Toc61769522)

[6.54 Obscure Language Features [BRS] 41](#_Toc61769523)

[6.55 Unspecified Behaviour [BQF] 42](#_Toc61769524)

[6.56 Undefined Behaviour [EWF] 42](#_Toc61769525)

[6.57 Implementation–defined Behaviour [FAB] 42](#_Toc61769526)

[6.58 Deprecated Language Features [MEM] 43](#_Toc61769527)

[6.59 Concurrency – Activation [CGA] 44](#_Toc61769528)

[6.60 Concurrency – Directed termination [CGT] 44](#_Toc61769529)

[6.61 Concurrent Data Access [CGX] 44](#_Toc61769530)

[6.62 Concurrency – Premature Termination [CGS] 45](#_Toc61769531)

[6.63 Lock Protocol Errors [CGM] 45](#_Toc61769532)

[6.64 Uncontrolled Format String [SHL] 46](#_Toc61769533)

[6.65 Modifying Constants [UJO] 46](#_Toc61769534)

[7. Language specific vulnerabilities for SPARK 46](#_Toc61769535)

[8. Implications for standardization 46](#_Toc61769536)

[Bibliography 47](#_Toc61769537)

[Index 50](#_Toc61769538)

# Foreword

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

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

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

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

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

ISO/IEC 24772-6 was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

With the cancellation of ISO/IEC TR 24772:2013, this document replaces ISO/IEC TR 24772:2013 Annex G. The main changes between this document and the previous version are:

* This document has been brought up to date with respect to the most recent (June 2020) release of the SPARK Language Reference Manual.
* Recommendations to avoid vulnerabilities are ranked and the top 12 are placed in a table in clause 5, together with the vulnerabilities in clauses 6 that contain each recommendation.
* The following vulnerabilities that were documented in clause 8 of ISO/IEC TR 24772:2013 are now addressed in this document in clause 6.
  + [CGA] *Concurrency – Activation*
  + [CGT] *Concurrency – Directed termination*
  + [CGX] *Concurrent data access*
  + [CGS] *Concurrency – Premature termination*
  + [CGM] *Protocol lock errors is now Lock protocol errors*
  + [CGY] *Inadequately secure communication of shared resource.*
* Clauses 6.2 *Terminology* is integrated into clause 3, and all subclauses in clause 6 are renumbered.
* The following vulnerabilities were removed:
  + [XZI] *Sign extension error* was integrated into [XTR] *Type system*.
  + [REU] *Termination strategy*, 6.39, is placed in clause 7 in Part 1, and hence is not documented for SPARK herein.
* The following vulnerabilities were renamed to track the changes made in Part 1:
  + [HFC] *Pointer casting and pointer type changes* was renamed to *Pointer type conversion*;
  + [JCW] *Operator precedence/Order of evaluation*, was renamed to *Operator precedence and associativity*;
  + [XYL] *Memory leak* is renamed to *Memory leaks and heap fragmentation*;
  + [XYP] *Hard coded password* is renamed *Hard coded credentials*;
* New vulnerabilities are added, to match the additions of Part 1:
  + [YAN] *Deep vs shallow copying*;
  + [BLP] *Violations of the Liskov substitution principle or the contract model*;
  + [PPH] *Redispatching*;
  + [BKK] *Polymorphic Variables*;
  + [SHL] *Reliance on external format strings*;
* Guidance material for each vulnerability given in subclause 6.X.2 is reworded to be more explicit and directive.

Addition material has been added for some vulnerabilities to reflect addition knowledge gained since the publication of ISO/IEC 24772-2

# Introduction

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

This technical report part is intended to be used with ISO/IEC 24772–1, which discusses programming language vulnerabilities in a language independent fashion. It is also intended to be used with ISO/IEC 24772-2, Ada which discusses how the vulnerabilities introduced in ISO/IEC 24772-1 are manifested in Ada, which is a superset of SPARK.

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

**Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages — Vulnerability descriptions for the programming language SPARK**

# 1. Scope

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

Vulnerabilities described in this Technical Report document the way that the vulnerability described in the language-independent ISO/IEC 24772–1 are manifested in SPARK.

This report is based on the publicly available “Community 2020” release of the SPARK, which is itself based on Ada 2012. Earlier versions of SPARK (those based on Ada83 through Ada2005), are *not* covered by this report.

# 2. Normative references

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

ISO 80000–2:2009, *Quantities and units* — *Part 2: Mathematical signs and symbols to be use in the natural sciences and technology*

ISO/IEC 2382–1:1993, *Information technology — Vocabulary — Part 1: Fundamental terms*

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

ISO/IEC 24772-2, *Programming Languages— Guidance to avoiding vulnerabilities in programming languages – Part 2: Ada*

ISO/IEC 8652:2012, *Information Technology – Programming Languages—Ada*

# 3. Terms and definitions, symbols and conventions

## 3.1 Terms and definitions

For the purposes of this document, the terms and definitions given in ISO/IEC 2382, in ISO/IEC 24772–1, in ISO/IEC 24772–2, in 14882:2014 and the following apply. Other terms are defined where they appear in *italic* type.

See clause 2. Normative references, plus the bibliography. In the body of this annex, the following documents are referenced using the short abbreviation that introduces each document, optionally followed by a specific section number. For example “[SRM 5.2]” refers to section 5.2 of the SPARK Reference Manual.

[SRM] *SPARK 2014 Reference Manual Release 2020*. AdaCore and Altran UK, April 2020 [1]. Available from <https://www.adacore.com/papers/spark-2014-reference-manual-release-2020>

# 4. Language concepts

The SPARK language is a subset of Ada, specifically designed for high-assurance systems. SPARK is designed to be amenable to various forms of static analysis that prevent or mitigate the vulnerabilities described in this Document.

Many terms and concepts applicable to Ada also apply to SPARK. See clauses 3 and 4 of ISO/IEC 24772-2.

This clause introduces concepts and terminology which are specific to SPARK and/or relate to the use of static analysis tools.

**Soundness**

This concept relates to the absence of false-negative results from a static analysis tool. A false negative is when a tool is posed the question “Does this program exhibit vulnerability X?” but incorrectly responds “no.” Such a tool is said to be *unsound* for vulnerability X. A *sound* tool effectively finds all the vulnerabilities of a particular class, whereas an unsound tool only finds some of them.

The provision of soundness in static analysis is problematic, mainly owing to the presence of unspecified and undefined features in programming languages. Claims of soundness made by tool vendors should be carefully evaluated to verify that they are reasonable for a particular language, compilers and target machines. Soundness claims are always underpinned by assumptions (for example, regarding the reliability of memory, the correctness of compiled code and so on) that should also be validated by users for their appropriateness.

Static analysis techniques can also be *sound in theory* – where the mathematical model for the language semantics and analysis techniques have been formally stated, proved, and reviewed – *but unsound in practice* owing to defects in the implementation of analysis tools. Again, users should seek evidence to support any soundness claim made by language designers and tool vendors. A language which is *unsound in theory* can never be sound in practice.

The single overriding design goal of SPARK is the provision of a static analysis framework which is sound in theory**,** and as sound in practice as is reasonably possible.

In the subclauses below, we say that SPARK *prevents* a vulnerability if supported by a mandatory form of static analysis which is sound in theory. Otherwise, we say that SPARK *mitigates* a particular vulnerability.

**SPARK Analyzer**

We define a “SPARK Analyzer” to be a tool that implements the various forms of static analysis required by the SPARK language definition. Without a SPARK Analyzer, a program cannot reasonably be claimed to be SPARK at all, much in the same way as a compiler checks the static semantic rules of a standard programming language.

In SPARK, certain forms of analysis are *mandatory*– they are required to be implemented and programs must pass these checks to be valid SPARK. Examples of mandatory analyses are

* Enforcement of the SPARK language subset.
* Verification of the absence of aliasing.
* Verification of the absence of function side-effects.
* Verification that every variable is initialized before use.
* Verification of the absence of undefined or erroneous behaviour.
* Verification that there is no dependence on unspecified behaviour.
* Verification of the absence of “runtime errors” that would normally raise a predefined exception in Ada, such as buffer overflow, division-by-zero, and arithmetic overflow.

Some analyses are said to be *optional* – users may choose to enable these additional analyses at their discretion. The most notable example of an optional analysis in SPARK is the generation and proof of verification conditions for user-defined contracts. Sometimes default SPARK proofs will be assisted by adding contracts such as type invariants, assertions, loop invariants and subprogram preconditions and postconditions. Optional analyses may provide greater depth of analysis, protection from additional vulnerabilities, and functional proofs of correctness.

**Failure modes for static analysis**

Unlike a language compiler, a user can always choose not to run a static analysis tool. Therefore, there are two modes of failure that apply to all vulnerabilities:

1. The user fails to apply the appropriate static analysis tool to their code.
2. The user fails to review or mis-interprets the output of static analysis.

In the discussion of specific vulnerabilities in clause 6, this report assumes that a user has sufficient expertise to apply a SPARK Analyzer and interpret the results correctly.

**Unsafe Programming**

In recognition of the occasional need to step outside the type system or to perform “risky” operations, SPARK provides clearly identified language features to do so. These are:

* Using the generic Unchecked\_Conversion for type-conversions. See subclause 6.37.
* Use of pragma Assume, which allows a general Boolean expression to be asserted for the purposed of program verification.
* Hiding a unit from a SPARK Analyzer, by NOT providing the aspect “SPARK\_Mode” on a unit or on its body. This means that the unit body is written in Ada, but not SPARK. For such units, the advice of ISO/IEC 24772-2 applies.
* The pragma Suppress allows an implementation to omit run-time checks. A SPARK Analyzer justifies the use of this pragma by verifying that those checks will never fail at run-time. See subclause 6.52 Suppression of Language-defined Run-time Checking [MXB].

**Access Types in SPARK**

Over and above the mechanisms inherited from Ada, SPARK requires additional protections from vulnerabilities associated with the use of access types and values.

Several vulnerabilities listed in clause 6 concern access types, so this clause contains an introductory description of how access types are managed in SPARK, in order to avoid repetition of that material in clause 6.

Firstly, avoid the use of access types if possible. In SPARK, many common programming idioms can be implemented without the explicit use of access types. Parameter passing, including mutable parameters and functions returning composite types do not require the use of access types in SPARK. Similarly, the use of array types and low-level programming (such as mapping a variable to a specific memory location) are achieved in SPARK without recourse to access types.

In SPARK, only simple “access-to-variable” and “access-to-constant” types are permitted which allocate memory from a single, global storage pool. User-defined storage pools are not permitted. “General” access types which can reference global memory or memory on the stack are not permitted. Access-to-subprograms are not permitted.

An access value in SPARK can either be an “Owner” or an “Observer” of the designated memory. At any point in the execution of a SPARK program, any allocated area of memory can only have a single access value that owns it. Assignment of access values transfers ownership, leaving the original value unable to access the designated memory for reading or writing.

An “Observing” access value has read-only permission on a constant object, but several such observers are allowed to exist.

Any one area of allocated memory has exactly one owner, one or more observers, but not both.

Additionally, the ownership of an access value can be “borrowed” by a locally declared access value, with the ownership automatically returning to the original value at the end of the borrowing value’s scope. This “borrowing” allows for subprograms that traverse or modify linked and recursive data structures before returning ownership to an enclosing scope or calling subprogram.

A SPARK Analyzer is required to keep track of the ownership relationship between access values and allocated memory, and to enforce legality rules which are designed to prevent defects and vulnerabilities. See clause 6 for further information on how these rules apply to the vulnerabilities identified by ISO/IEC 24772-1.

Full details of the ownership and legality rules for access types and values are in [SRM 3.10].

# 5. General guidance for SPARK

In addition to the generic programming rules from ISO/IEC 24772-1 clause 5.4, additional rules from this clause apply specifically to the SPARK programming language. The recommendations of this clause are restatements of recommendations from clause 6, but represent ones stated frequently, or that are considered as particularly noteworthy by the authors. Clause 6 of this document contains the full set of recommendations, as well as explanations of the problems that led to the recommendations made.

|  |  |  |
| --- | --- | --- |
| **Index** | **Avoidance Mechanism** | **Reference** |
| 1 | Do not use features explicitly identified as unsafe, such as Unchecked\_Conversion, unless absolutely necessary and then with extreme caution. Apply the ‘Valid attribute on the value of a scalar object that results from a call to Unchecked\_Conversion. | 6.2 [IHN], 6.3 [STR], 6.11 [HFC], 6.14 [XYK],  6.33 [DCM], 6.53 [SKL],  6.56 [EWF] |
| 2 | Do not suppress the checks provided by the language unless the absence of the errors checked against has been verified by a SPARK Analyzer | 6.6 [FLC], 6.9 [XYZ],  6.33 [DCM], 6.52 [MXB],  6.56 [EWF] |
| 3 | Use a SPARK Analyzer 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] |
| 4 | Removed |  |
| 5 | Specify preconditions and postconditions on subprograms. | 6.32 [CSJ], 6.34 [OTR], 6.46 [TRJ] |
| 6 | 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] |
| 7 | Exploit the type and subtype system to express constraints as well as preconditions and postconditions on the values of parameters. | 6.46 [TRJ] |
| 8 | Specify subtype predicates and type invariants. | 6.44 [BKK], 6.46 [TRJ] |
| 9 | Whenever possible, the 'First, 'Last, and 'Range attributes should be used for loop termination. If the 'Length attribute must 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] |
| 10 | Use SPARK'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] |
| 11 | For **case** statements and aggregates, do not use the **others** choice. | 6.5 [CCB], 6.27 [CLL] |
| 12 | Protect all data shared between tasks within a protected object or a suspension object, or mark the data Atomic. | 6.3 [STR], 6.56 [EWF], 6.61 [CGX] |

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

These vulnerability guidelines may be categorized into several functional groups.

* Items 1, 2, and 3 are applicable to Exceptional and Erroneous Behaviours.
* Mitigation methods associated with Types, Subtypes, and Contracts include Items 5, 6, 7, and 8.
* Those techniques appropriate for Statements and Operations consist of Items 9, 10, and 11
* Finally, Item 12 is pertinent to Concurrency in applications.

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

# 6. Specific Guidance for SPARK Vulnerabilities

## 6.1 General

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

For the remainder of this clause 6, the following assumptions apply:

* A user applies a SPARK Analyzer (in addition to a compiler) and has the necessary skills and expertise to understand and act on its output.
* A SPARK Analyzer is used that implements the mandatory analyses required by the SPARK language design, including all of those analyses listed in clause 4.
* Unsafe programming and, and in particular the use of Unchecked\_Conversion and pragma Assume, is *not* used. The use of unsafe programming techniques subverts the prevention of many classes of vulnerability, so must be strictly controlled.

## 6.2 Type System [IHN]

### 6.2.1 Applicability to language

A design goal of SPARK is the provision of *strong* *static type safety,* meaning that programs can be shown to be free from all run-time type failures using entirely static analysis. This depth of verification is mandatory in SPARK. Even so, verification of type safety can be confounded in the following ways:

* The use of unsafe programming techniques, specifically the use of Unchecked\_Conversion and pragma Assume, can introduce vulnerabilities that will not always be detected by a SPARK Analyzer. See subclause 6.37 Type-breaking Reinterpretation of Data [AMV].
* Mixed language programming can defeat the type system of a SPARK program. See subclause 6.47 Inter-language Calling [DJS].
* A SPARK Analyzer may not be able to verify *all* the type safety checks, although these failed verifications may be a *false alarm*.
* A program which fails full type safety verification with a SPARK Analyzer may nonetheless still be a legal Ada program, and so can still be compiled, linked, and deployed.

### 6.2.2 Guidance to language users

* Follow the guidance of ISO/IEC 24772-1 subclause 6.2.2.
* Use a SPARK Analyzer to verify the absence of runtime type errors.
* Document and justify a process for dealing with false alarms arising from static verification.
* Develop processes and tooling that prevent the compilation and linking of SPARK executables that do not meet the required depth of static verification.

Note 1: SPARK programs that have been subject to this depth of analysis can be compiled with run-time checks suppressed, supported by a body of evidence that such checks could never fail, and thus removing the possibility of erroneous execution.

6.3 Bit Representations [STR]

### 6.3.1 Applicability to language

SPARK prevents these vulnerabilities within a single self-contained SPARK program.

These vulnerabilities may occur when passing data to or from another programming language, or to another program or network, where data representation may differ.

Note: Bit-wise operations in SPARK are permitted for modular types only, where the semantics is defined mathematically, and does not depend on the chosen representation. Bit-wise operations on signed integer types, floating point types, or enumeration types are not permitted.

### 6.3.2 Guidance to language users

* Explicitly document any reliance on bit ordering or usage using SPARK’s representation clauses.
* Where bit ordering can change either between the development host and the target, or between interfaced targets, provide compatible types with derived types that document each system’s mapping and explicitly convert between them. For example:

**type** Reversible\_Integer **is** **new** Integer;

**type** Big\_E\_Integer **is** **new** Reversible\_Integer;

**for** Big\_E\_Integer’Bit\_Order **use** High\_Order\_First;

**type** Little\_E\_Integer **is** **new** Reversible\_Integer;

**for** Little\_E\_Integer’Bit\_Order **use** Low\_Order\_First;

BI : Big\_E\_Integer := <Some\_Value>;

LI : Little\_E\_Integer := Little\_E\_Integer(BI);

* Localize and document the code associated with explicit manipulation of bits and bit fields.
* Use a SPARK Analyzer to verify the correct conversion between types.

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

See ISO/IEC 24772-2 subclause 6.4.

Additionally, SPARK mitigates floating-point vulnerabilities through mandatory static verification of *type safety* for all floating-point operations and conversions.

### 6.4.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.4.5 of ISO/IEC 24772-1 and subclause 6.4.2 of ISO/IEC 24772-2.
* Use a SPARK Analyzer to verify type safety of all floating point operations and conversions.
* If a specific compiler and target system implement a particular model of floating-point arithmetic, such as ISO/IEC 60559[3], then document any implementation-defined choices (e.g. rounding mode) made by that implementation.
* Verify and document that a SPARK Analyzer makes the same implementation-defined choices for verification as the target compiler and system.
* Check the validity of floating-point values received from another programming language or as inputs using the ‘Valid attribute. In particular, Ada requires that ‘Valid returns False for bit patterns which represent NaN.

## 6.5 Enumerator Issues[CCB]

### 6.5.1 Applicability to language

The vulnerability of unexpected but well-defined program behaviour upon extending an enumeration type exists in SPARK. In particular, subranges or **others** choices in aggregates and case statements are susceptible to unintentionally capturing newly added enumeration values.

Vulnerabilities relating the use of non-standard representation clauses with enumerated types do not apply to SPARK, since SPARK’s semantics are independent of representation values.

Vulnerabilities relating to Unchecked\_Conversion of enumerated types do not apply to SPARK, since SPARK limits the use of Unchecked\_Conversion to types which have exactly the same number of valid values [SRM 13.9].

### 6.5.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.5.5 of ISO/IEC 24772-1 and subclause 6.5.2 of ISO/IEC 24772-2.

## 6.6 Conversion Errors [FLC]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.6 does not apply to SPARK, because SPARK requires mandatory static verification of type safety for all conversions.

## 6.7 String Termination [CJM]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.7 does not apply to SPARK, because strings are not delimited by a termination character. SPARK programs that interface to any language that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

## 6.8 Buffer Boundary Violation [HCB]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.8 does not apply to SPARK (see 6.9 Unchecked Array Indexing [XYZ] and 6.10 Unchecked Array Copying [XYW]).

## 6.9 Unchecked Array Indexing [XYZ]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.9 does not apply to SPARK, since SPARK requires mandatory static verification of type safety for all array indexing operations.

Note 1: Use SPARK’s support for whole array operations, such as assignment and comparison, plus aggregates for whole-array initialization, to reduce the use of indexing.

## 6.10 Unchecked Array Copying [XYW]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.10 does not apply to SPARK, since SPARK requires mandatory static analysis verification that both the source and the target of an array assignment have matching lengths.

## 6.11 Pointer Type Conversions [HFC]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.11 does not apply to SPARK, because SPARK forbids type conversion of access values.

## 6.12 Pointer Arithmetic [RVG]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.12 does not apply to SPARK, because SPARK forbids pointer arithmetic.

## 6.13 NULL Pointer Dereference [XYH]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.13 does not apply to SPARK, because SPARK requires mandatory static verification that a null value can never be dereferenced.

Note: Use non-null access types where possible since it simplifies verification.

## 6.14 Dangling Reference to Heap [XYK]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.14 does not apply to SPARK, because SPARK requires mandatory static verification of ownership of access values. In particular:

* SPARK’s ownership model for access values, and transfer of that ownership on assignments, mean that dangling access values cannot exist.
* Allocated memory must be deallocated before its owner goes out of scope. Failure to do so will be reported by the static analysis tool as a memory leak.

## 6.15 Arithmetic Wrap-around Error [FIF]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.15 does not apply to SPARK, because:

* Modular integer types exhibit modular arithmetic, which is well-defined in all circumstances, and can never generate an unexpected value, a negative value, or an exception.
* Arithmetic for signed integer types never exhibits wrap-around, and is subject to mandatory static verification of type safety in SPARK.

## 6.16 Using Shift Operations for Multiplication and Division [PIK]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.16 does not apply to SPARK, because:

* Shift operations are limited to the modular types declared in the predefined package Interfaces.
* Modular types do not permit negative values.

## 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 Identifier Name Reuse [YOW].

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

* Mixed casing. This is not an issue since SPARK treats upper-case and lower-case letters in names as identical. Confusion for the programmer may arise through an attempt to use Item and ITEM as distinct identifiers with different meanings, but the language system and strong type checking will verify appropriate and correct usage.
* Underscores and periods. SPARK permits single underscores in identifiers and they are significant. Thus, BigDog and Big\_Dog are different identifiers and the language system and strong type checking will ensure appropriate and correct usage. Multiple underscores (which might be confused with a single underscore), leading underscores, and trailing underscores are forbidden.
* Periods in SPARK denote substructures and hence are meaningful.
* Singular/plural forms. SPARK permits the use of identifiers which differ solely in this manner such as Item and Items. The programmer may create plural and singular forms to identify single items or collections, and the language system and strong type checking will ensure appropriate and correct usage.
* International character sets. SPARK strictly conforms to the appropriate International Standard for character sets.
* Identifier length. All characters in an identifier in SPARK are significant and 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.

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

1. Declares such similar names to have different types in which case the type system will guarantee safe usage; or
2. Creates contracts that define the functional behaviour of the code module and uses the analysis and proof tools to verify correct usage.

### 6.17.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.17.5 of ISO/IEC 24772-1.
* Avoid the use of similar names to denote different objects of the same type.
* Adopt a project convention for dealing with similar names.

## 6.18 Dead Store [WXQ]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.18 does not apply to SPARK, because SPARK requires mandatory static verification of information flow which detects and reports all dead stores. Additionally, SPARK requires variables that are used for output to the environment, where multiple writes to a variable without intervening reads could be confused as dead store, to be specifically identified. In this case, the information flow analysis for such variables is modified since it is known that consecutive writes to such variables might not constitute a dead store.

## 6.19 Unused Variable [YZS]

### 6.19.1 Applicability to language

SPARK is designed to permit static verification of the following cases:

* Variables which are declared but not used at all.
* Variables which are assigned to, but the resulting value is not used in any way that affects an output of the enclosing subprogram.
* Subprogram formal parameters of mode “in” which are never used in a way that affects an output of that subprogram.

### 6.19.2 Guidance to language users

* Apply a SPARK Analyzer to verify the absence of unused variables and parameters.
* Enable and act on compiler warnings for other unused entities.

## 6.20 Identifier Name Reuse [YOW]

### 6.20.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.20 is mitigated by SPARK.

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

Name collisions with keywords cannot happen in SPARK since keywords are reserved.

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

### 6.20.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.20.5 of ISO/IEC 24772-1.
* Use *expanded names* whenever confusion may arise*.*
* Use compiler warnings or other static analysis tools to detect declarations in inner scopes that hide declarations in outer scopes, and to prevent the reuse of predefined identifiers, such as Integer or Boolean.

## 6.21 Namespace Issues [BJL]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.21. does not apply to SPARK, because the language does not attempt to disambiguate conflicting names imported from different packages. 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]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.22. does not apply to SPARK, because SPARK requires mandatory static verification of information flow.

## 6.23 Operator Precedence and Associativity [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, SPARK is less susceptible to that vulnerability than many other languages, since

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

### 6.23.2 Guidance to language users

* Follow the guidance provided in ISO/IEC 24772-1 subclause 6.23.5
* Use parentheses whenever arithmetic operators, logical operators, mixed logical operators such as “and” and “and then” and shift operators are mixed in an expression.
* Create contracts that specify the expressions in mathematical terms and verify using a SPARK Analyzer.

## 6.24 Side-effects and Order of Evaluation of Operands [SAM]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.24 does not apply to SPARK, because

* SPARK does not include operators that have side-effects.
* In SPARK, all functions (and hence function calls) are free from side-effects.
* Assignment is a statement, not an expression.
* In SPARK, expression evaluation order is unspecified, but the language design requires mandatory static verification that any possible evaluation order always yields the same result, and that all intermediate expressions are type-safe.

## 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 may result in a program which is accepted by the compiler but does not reflect the intent of the author.

The examples given in subclause 6.25 of ISO/IEC 24772-1 do not apply to SPARK because of strong typing and because an assignment is not an expression in SPARK.

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

Type\_Name (Expression) -- a type-conversion

vs.

Type\_Name'(Expression) -- a qualified expression

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

Problems arising from a failure to use short-circuit Boolean forms are less frequent in SPARK programs because static verification will reveal failure to verify the right-hand side of such an expression if its successful evaluation depends on the value of the left-hand side. For example, if a user correctly writes:

**if** (A /= **null**) **and** **then** (A.**all** = 0) **then** ...

then a SPARK analyzer is required to verify that A cannot be null on the right-hand side, so the expression will evaluate successfully. If the user mistakenly uses the non-short-circuit form:

**if** (A /= **null**) **and** (A.**all** = 0) **then** ...

then a SPARK Analyzer will report a potential null dereference on the right-hand side.

### 6.25.2 Guidance to language users

* Use short-circuit Boolean operators where the expression on the right-hand side includes a call to a function that has an explicit precondition, or uses an operator (such as division or pointer dereference) that has an implicit precondition, and establish that precondition on the left-hand side. For example:

**if** (Y /= 0) **and** **then** ((X / Y) > 1) **then** ...

## 6.26 Dead and Deactivated Code [XYQ]

### 6.26.1 Applicability to language

These vulnerabilities can apply to SPARK in much same way as in Ada.

Although it is not strictly required by the language design, a SPARK Analyzer may offer facilities to detect dead code through static verification:

* A dead *path* in a subprogram can be detected because the logical condition that guarantees its execution is equivalent to “False”.
* Analysis of the “closure” of a complete program partition can reveal subprograms that are never called and/or packages and other entities that are never referenced.

### 6.26.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.26.5 of ISO/IEC 24772-1.
* Use implementation-specific mechanisms, if provided, to support the elimination of dead code. In some cases, use pragmas such as Restrictions, Suppress, or the Discard\_Names aspect to inform the compiler that some code whose generation would normally be required for certain constructs would be dead because of properties of the overall system, and that therefore the code need not be generated. For example:

**package** Pkg **is**

**type** Enum **is** (Aaa, Bbb, Ccc)

**with** Discard\_Names;

**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 aspect allows the elimination of the table.

## 6.27 Switch Statements and Static Analysis [CLL]

### 6.27.1 Applicability to language

, which requires that a case statement provides exactly one alternative for each value of the expression's subtype. The others clause may 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. Control does not flow from one alternative to the next. Upon reaching the end of an alternative, control is transferred to the end of the case statement.

The sole remaining vulnerability is that unexpected values can be captured by the others clause or a subrange as case choice. The introduction of additional values may have been intended to have their own case alternatives but instead fall into the others category. Another example is the inclusion of additional values internal to a range (usually done by adding an enumeration value to an enumeration type but not at the first or last of that type), and some case statements choices hide the addition in a range of 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.
* When adding enumeration values to an enumeration type, review all of the places where if statements or case choices are used to ensure that the position of the added value does not create logic errors.

## 6.28 Demarcation of Control Flow [EOJ]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.28 does not apply to SPARK, because SPARK enforces a clear demarcation of all branching control flows, if statements, case statements, loops, and blocks.

## 6.29 Loop Control Variables [TEX]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.29 does not apply to SPARK, because “for” loops in SPARK define a loop control variable that has a constant view in the loop body and cannot be modified by the sequence of statements therein.

For more general loops, SPARK also supports the specification and verification of a loop *variant* contract that can be used to verify termination of loops in simple cases.

## 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 SPARK **for loop** does not require the programmer to specify a conditional test for loop termination. Instead, the starting and ending value of the loop can be specified (in terms of using a subrange expression to define the object being iterated over or using ‘First and ‘Last to eliminate this source of off-by-one errors. SPARK also provides 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 could be susceptible to an off-by-one error. Any off-by-one error that gives rise to the potential for a buffer-overflow, range violation, or any other construct that could give rise to a predefined exception, will be prevented by mandatory static verification of type safety in SPARK.

#### 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. Again, any off-by-one error that gives rise to a type-safety vulnerability will be prevented by a SPARK Analyzer.

SPARK does not use sentinel values to terminate arrays (such as strings). Therefore, the vulnerability documented in ISO/IEC 24772-1 subclause 6.30 related to space required for implicit sentinel values does not apply to SPARK.

### 6.30.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.30.5 of ISO/IEC 24772-1.
* Whenever possible, use a **for** **loop** instead of a **while loop**.
* Whenever possible, use 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** My\_Array'Range **loop**….
* If the 'Length attribute must be used, ensure that the index computation considers the starting index value for the array.
* Use a SPARK Analyzer to verify type safety of all code.

## 6.31 Unstructured Programming [EWD]

### 6.31.1 Applicability to language

SPARK programs can exhibit some of the vulnerabilities noted in subclause 6.31 of ISO/IEC 24772-1: leaving a **loop** at an arbitrary point, and multiple exit points from subprograms, but these are mitigated by mandatory static verification of control- and information-flow.

SPARK forbids the **goto** statemement, exception handlers, non-local jumps and subprograms with multiple entries.

### 6.31.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.31.5 of ISO/IEC 24772-1.

## 6.32 Passing Parameters and Return Values [CSJ]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.32 does not apply to SPARK, because:

* SPARK uses parameter modes in, out and in out to specify the desired direction of information flow for each formal parameter of a subprogram.
* Functions in SPARK are expressions and never have a side-effect.
* SPARK allows the programmer to specify a Global Contract for each subprogram that specifies exactly the global variables (and their modes) that are accessed by that subprogram. If it is given, then Global Contract is verified by static verification.
* SPARK requires mandatory static verification of the absence of aliasing [SRM 6.4.2] between actual parameters and global variables at each procedure call statement. This means that the semantics of a program are not affected by a compiler’s choice of parameter passing mechanism for each parameter.
* SPARK requires static verification of information flow to verify that the value returned from a function call is never ignored.

## 6.33 Dangling References to Stack Frames [DCM]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.33 does not apply to SPARK, because SPARK forbids the use of the ‘Address, ‘Access and ‘Unchecked\_Access attributes, so an access value or address values that denotes a stack-allocated object can never be generated.

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

There are only two cases where this vulnerability can apply to SPARK.

The first case is for calls to/from subprograms where the other side is a foreign language. This case is addressed in subclause 6.47.

The second case is where a subprogram contains formal parameters with default expressions.

In the second case, actual parameters are constructed for the missing formal parameters via the default expression, hence all subprogram expressions will exist and there will be no stack corruption.

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 may include default expressions for a parameter. Hence, a procedure call may be constructed with some actual parameters missing. In this case, if there is a default expression for the missing parameter, then the call will be compiled without any errors. If no default expression exists for missing parameters, then a compilation error is generated.

### 6.34.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.34.5 of ISO/IEC 24772-1.
* Minimize the use of default expressions for formal parameters.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

SPARK permits recursion. The exception Storage\_Error is raised when the recurring execution results in insufficient storage. This will result in program termination unless an exception handler is placed outside the SPARK portion of the program.

### 6.35.2 Guidance to language users

* Apply the guidance described in ISO/IEC 24772-1 subclause 6.35.5.
* Use static analysis to verify worst-case stack usage.
* Consider applying the restriction No\_Recursion to eliminate this vulnerability.

## 6.36 Ignored Error Status and Unhandled Exceptions [OYB]

### 6.36.1 Applicability to language

SPARK permits the declaration of exceptions, and the execution of the **raise** statement. SPARK does not permit exception handlers, which means that all SPARK programs must be verified to be free of all predefined and user defined exceptions. Note however, that exception handlers can be declared in parts of the program explicitly excluded from a SPARK analyzer, for example in the main subprogram to handle exceptions generated by hardware faults and to handle program closeout or restart.

The ’Valid attribute can be used to check the result of Unchecked\_Conversion and to handle resulting error conditions by explicit code such as an if statement.

If a subprogram returns an error status value via a formal parameter, then SPARK requires that the assignment to the corresponding actual parameter is not an *ineffective assignment* (See 6.19).

### 6.36.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.36.5 of ISO/IEC 24772-1.
* Use a SPARK Analyzer to verify the absence of runtime errors.
* Use a SPARK Analyzer to verify that user-defined exceptions can never be raised.
* Use the result of the 'Valid attribute to check for the validity of values delivered to a SPARK program from an external device or from Unchecked\_Conversion prior to use and explicitly handle both True and False cases.
* Consider placing a top-level exception handler in the main program (external to SPARK) and in each task so that notification of failure can be given.

## 6.37 Type-breaking Reinterpretation of Data [AMV]

### 6.37.1 Applicability to language

There are two unsafe programming techniques allowed by SPARK that can lead to this vulnerability: the use of the generic Unchecked\_Conversion function, and the use of the Unchecked\_Union aspect on a discriminated, variant record type.

SPARK permits the instantiation and use of Unchecked\_Conversion as in Ada, but limits instantiation to types that have the same size (in bits) and the same number of valid values. Hence, a call to a legal instantiation of Unchecked\_Conversion cannot generate an invalid value in SPARK.

For example, converting Interfaces.Integer\_16 onto Interfaces.Unsigned\_16 is permitted, since their ‘Size attribute is 16 in both cases, and both have exactly 216 valid values. Conversely, an instantiation of Unchecked\_Conversion from Interfaces.Unsigned\_8 to Boolean is not permitted, since the ‘Size of the former is 8, while the ‘Size of the latter is 1.

Unchecked\_Union allows a discriminated, variant record type to be directly compatible with a matching declaration of a “union” type in C. A SPARK Analyzer is required to verify that access to fields of an Unchecked\_Union object are only legal when the (implicit) discriminant is known because the object is of a constrained subtype.

Language rules prevent the changing of a discriminant of a variable unless the whole object is written, so reinterpreting an objects components is not possible. Record extensions require that the extension components be written or read by subprograms with visibility to the extensions, hence those elements will be correctly interpreted.

### 6.37.2 Guidance to language users

* Follow the guidelines of ISO/IEC 24772-1 subclause 6.37.5.
* Limit the use of Unchecked\_Union to units that must interface directly with C code only.
* Consider applying the restrictions No\_Use\_Of\_Pragma(Unchecked\_Union),  
  No\_Use\_Of\_Aspect(Unchecked\_Union), and No\_Unchecked\_Conversion to ensure this vulnerability cannot arise.
* Apply ‘Valid to the result of Unchecked\_Conversion.

## 6.38 Deep vs. Shallow Copying [YAN]

### 6.38.1 Applicability to language

In SPARK, the default semantics of assignment create a shallow copy, when applied to the root of a graph structure.

Vulnerabilities can be mitigated by limited types (which have no default assignment operator), 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.

### 6.38.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.38.5 of ISO/IEC TR 24772-1:2019.
* Use limited types and/or user-defined copying operations to enforce the correct semantics.
* Use predefined Container packages and types for linked data structures.

## 6.39 Memory Leak and Heap Fragmentation [XYL]

### 6.39.1 Applicability to language

Memory leaks are prevented in SPARK by mandatory static verification of the ownership of access values and associated rules [SRM 3.10]. In particular, SPARK requires that an access value is null before it is Finalized (i.e. goes out of scope), but the only way to set an access value back to null in SPARK is to call Unchecked\_Deallocation on it.

SPARK does not directly address the issue of heap fragmentation, so this vulnerability remains, especially for long-running systems.

### 6.39.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.39.5 of ISO/IEC TR 24772-1:2019.
* Declare access types in a nested scope where possible.
* Use moded formal parameters and composite types appropriately to avoid the use of access types altogether.
* 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]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.40 does not apply to SPARK, because:

* SPARK’s 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 and SPARK generates static checks for each instantiation of the generic.
* A compiler is able to check the generic body for programming errors, independently of actual instantiations. At each actual instantiation, the compiler will also check that the instantiated type meets all the requirements of the generic contract.
* SPARK 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

SPARK permits single inheritance of data members and operations from an ancestor class, where only that one ancestor (the parent) may implement operations. Other ancestors (known as “Interfaces”) can only specify the signature of additional operations, and whether each operation must be overridden, or can simply do nothing if never explicitly defined. Therefore, SPARK does not suffer from multiple-inheritance related vulnerabilities.

In SPARK, a user can specify if a redefined operation must override or must never override an inherited operation. When these specifications are given, they are verified statically, so their use prevents vulnerabilities relating to accidental overriding or failure to override.

SPARK has no preference rules to resolve ambiguities of calls on primitive operations of tagged types and thus reports the ambiguity for the programmer to disambiguate. Hence the related vulnerability documented in ISO/IEC 24772-1 subclause 6.41 does not apply.

SPARK also requires static verification to ensure that all data members of an object are correctly initialized before use, even when such initialization is achieved by delegation to the parent’s constructor operation or by a re-dispatching call to a constructor [SRM 6.1.7]. These rules also mitigate vulnerabilities caused by operations that must establish or maintain a type invariant.

### 6.41.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.41.5 of ISO/IEC 24772-1.
* Use the overriding indicators on potentially inherited subprograms to ensure that the intended set of operations are overridden, thus preventing the accidental redefinition or failure to redefine an operation of the parent.
* Specify the Global’Class and Depends’Class aspects for primitive operations to ensure that information-flow requirements as respected in derived classes [SRM 6.1.6].
* Specify Pre’Class and Post’Class aspects when a primitive operation is initially defined, to indicate the properties of inputs that any overridings must accept, and the properties of outputs that any overridings must produce.

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

### 6.42.1 Applicability to language

SPARK extends Ada’s capabilities in this area, allowing fully static verification of the LSP/Behavioural subtyping principle, assuming that a user has specified appropriate preconditions and postconditions on the primitive and overridden operations of tagged types.

SPARK also defines language rules [SRM 6.1.6] that allow the Global contract of an overriding subprogram to be modified from that inherited from its parent, but only in a way that does not violate LSP.

### 6.42.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.42.5 of ISO/IEC TR 24772-1:2019.
* Specify Pre’Class and Post’Class for all primitive operations of tagged types.
* Use a SPARK Analyzer to verify LSP for all descendent types.

## 6.43 Redispatching [PPH]

### 6.43.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1 subclause 6.43 is mitigated by SPARK . As in Ada, calls are non-dispatching by default in SPARK.

A redispatching call can only occur if an object of a specific type T is explicitly converted to the classwide type T’Class before being passed as the controlling parameter of a call. Such conversions are only allowed in SPARK if the enclosing subprogram has the Extensions\_Visible aspect applied to it. This aspect also modifies the required data initialization rules for that subprogram so that hidden components of the object cannot be left uninitialized [SRM 6.1.7].

### 6.43.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.43.5 of ISO/IEC TR 24772-1:2019.
* If redispatching is necessary, document the behaviour explicitly.
* Do not use the Extensions\_Visible aspect in order to forbid redispatching.

## 6.44 Polymorphic variables [BKK]

### 6.44.1 Applicability to language

There are three specific vulnerabilities to consider:

*Unsafe casts* are not permitted in SPARK.

A *downcast* in SPARK requires mandatory static verification that the *tag* of the object matches that of the target type or one its descendants.

An *upcast* to a specific tagged type is permitted in SPARK, and can never give rise to a runtime error. By specifying a Type\_Invariant aspect on a private extension, the programmer can ensure that the semantic requirements of the private extension, as captured by the type invariant, are preserved across such conversions to an ancestor specific type, in that they are re-checked after the construct manipulating the upward conversion is complete. If it is specified, then SPARK requires static verification that a Type\_Invariant is always preserved.

As noted in subclause 6.43, an *upcast* to a classwide type is not permitted in SPARK, unless the enclosing subprogram has the Extensions\_Visible aspect applied it.

### 6.44.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.44.5 of ISO/IEC TR 24772-1:2019.
* Use the Type\_Invariant contract to specify and verify the semantic consistency of derived types.
* Do not use the Extensions\_Visible aspect in order to forbid redispatching.

## 6.45 Extra Intrinsics [LRM]

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

## 6.46 Argument Passing to Library Functions [TRJ]

### Applicability to language

The general vulnerability that parameters might have values that are precluded by preconditions of the called routine applies to SPARK.

To the extent that the preclusion of values can be expressed as part of the type system of SPARK, the preconditions are verified by a SPARK Analyzer statically or can be checked by dynamic checks and thus are no longer vulnerabilities. For example, any range constraint on values of a parameter can be expressed in SPARK by means of type or subtype declarations. Type violations are detected at compile time; subtype violations cause run-time exceptions. For that situation, preconditions, postconditions, type invariants, and subtype predicates can be specified explicitly to express more complex restrictions to be observed by callers.

### 6.46.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.46.5 of ISO/IEC 24772-1.
* Exploit the type and subtype system of SPARK to express restrictions on the values of parameters and results.
* Specify explicit preconditions and postconditions for subprograms wherever practical.
* Specify subtype predicates and type invariants for subtypes and private types when appropriate.

## 6.47 Inter-language Calling [DJS]

### 6.47.1 Applicability to language

SPARK 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 mitigated.

Additionally, some parts of a SPARK program may be written in Ada by specifying the aspect “SPARK\_Mode => Off” for those units.

### 6.47.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.47.5 of ISO/IEC 24772-1.
* For units written in Ada (and therefore not subject to mandatory static verification with a SPARK Analyzer), follow the mitigations in ISO/IEC 24772-2. Consider adding a top-level exception handler in each Ada unit to catch and prevent an unhandled exception from propagating into SPARK code.
* Use the inter-language methods and syntax specified by SPARK and ISO/IEC 8652 [2] when the routines to be called are written in languages that ISO/IEC 8652 [2] 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 a SPARK application.

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.48 does not apply to SPARK, because SPARK supports neither dynamic linking nor self-modifying code.

## 6.49 Library Signature [NSQ]

### 6.49.1 Applicability to language

SPARK provides mechanisms to explicitly interface to modules written in other languages. The aspects Import, Export and Convention permit the name of the external unit and the interfacing convention to be specified.

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

### 6.49.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.49.5 of ISO/IEC 24772-1.
* Refer to ISO/IEC 8652 Annex B (“Interfacing to Other Languages”) to understand how each language-specific convention applies to different types and parameter modes.
* Verify that a particular compiler follows the implementation advice given in ISO/IEC 8652 Annex B.

## 6.50 Unanticipated Exceptions from Library Routines [HJW]

### 6.50.1 Applicability to language

SPARK permits the declaration and raising of exceptions, but does not support exception handlers, so any exception raised will cause either the task that was subject to the exception to silently terminate, or the main program to terminate.

Since SPARK is a subset of Ada, it is possible to hide the main body of a task or the main subprogram from SPARK and place an exception handler there to perform appropriate notifications or last wishes.

### 6.50.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.50.5 of ISO/IEC 24772-1.
* Ensure that the interfaces with libraries written in other languages are compatible in the naming and generation of exceptions.
* Consider failure strategies and consider placing exception handlers at the top level of all tasks and the main subprogram.

Note: Since exception declarations are external to SPARK, wrapping the main subprogram with another subprogram that exclusively calls the main SPARK subprogram and handles and exception minimizes the amount of non-SPARK code. Similarly, for tasks, placing the task code in a subprogram that never exits and making the task body contain only the call to that subprogram and the exception handlers minimizes the amount of non-SPARK code.

* Document any exceptions that may be raised by any Ada units being used as library routines.

## 6.51 Pre-processor Directives [NMP]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.51 does not apply to SPARK, because SPARK 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 SPARK 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 subclause 11.5 of ISO/IEC 8652 [2]).

SPARK requires mandatory static verification of type safety, which means that a run-time check will never fail, so this depth of verification provides assurance that pragma Suppress can be applied.

### 6.52.2 Guidance to Language Users

* Follow the mitigation mechanisms of ISO/IEC 24772-1 subclause 6.52.5.
* Verify type safety using a SPARK Analyzer.

## 

## 6.53 Provision of Inherently Unsafe Operations [SKL]

### 6.53.1 Applicability to language

The classes of vulnerability identified in ISO/IEC 24772-1 subclause 6.53 and techniques defined as “Unsafe programming” in clause 3 are covered by other subclauses of this document. Specifically:

* Vulnerabilities related to unchecked type conversion are covered in subclause 6.37 Type-breaking Reinterpretation of Data [AMV].
* Vulnerabilities related to deallocation of dynamically allocated memory are covered in subclause 6.14 Dangling Reference to Heap [XYK].
* Vulnerabilities related to mixed-language programming and the use of full Ada within a SPARK program are covered in subclause 6.47 Inter-language Calling [DJS].
* Vulnerabilities related to the suppression of run-time checking are covered in subclause 6.52 Suppression of Language-defined Run-time Checking [MXB].

### 6.53.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.53.5 of ISO/IEC 24772-1.
* Use a SPARK Analyzer to identify inherently unsafe operations.
* Avoid the use of unsafe programming practices.
* Use the pragma Restrictions to prevent the inadvertent use of unsafe language constructs. In particular, use pragma Restrictions (No\_Use\_Of\_Pragma => Assume) to prevent the use of pragma Assume.
* Carefully scrutinize any code that refers to a program unit explicitly designated to provide unchecked operations.
* Use non-SPARK units sparingly and ensure that a thorough analysis is performed on the code since a SPARK Analyzer will not be used.

## 6.54 Obscure Language Features [BRS]

### 6.54.1 Applicability of language

SPARK is designed to offer a completely unambiguous semantics, where a SPARK program that is verified with a SPARK Analyzer exhibits no undefined behaviour and no dependence on unspecified behaviour.

Nonetheless, SPARK provides facilities for a wide range of application areas. Because some areas are specialized, it is likely that a programmer not versed in a special area might misuse features for that area. For example, the use of tasking features for concurrent programming requires knowledge of this domain.

### 6.54.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.54.5 of ISO/IEC 24772-1.
* Use the pragma Restrictions to prevent the use of obscure features of the language. For example, a project might decide to completely forbid floating point types, access types, or tasking.
* Use the language-defined pragme Restriction (No\_Dependence => …) to prevent the use of specified predefined or user-defined libraries.

## 6.55 Unspecified Behaviour [BQF]

### 6.55.1 Applicability of language

SPARK is designed to either prevent or remove dependence on unspecified behaviour. For example, expression evaluation order is unspecified, but the rules of SPARK and static verification ensure that any evaluation order always yields the same result.

Bounded errors are entirely prevented by mandatory static verification.

One case remains: Results of certain operations within a language-defined generic package are unspecified if the actual associated with a particular formal subprogram does not meet stated expectations (such as “=” providing a true equality relationship)

### 6.55.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.55.5 of ISO/IEC 24772-1.
* For situations involving generic formal subprograms, ensure that the actual subprogram satisfies all of the stated expectations.

## 6.56 Undefined Behaviour [EWF]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.56 does not apply to SPARK, because (other than specific cases of unsafe programming techniques) undefined behaviour is prevented by mandatory static verification.

## 6.57 Implementation–defined Behaviour [FAB]

### 6.57.1 Applicability to language

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

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

Programmers must verify that the implementation-defined choices made by a compiler exactly match those made by a SPARK Analyzer. The most notable example is the range of the predefined Integer types, since these impact the verification of the absence of arithmetic overflow in expressions.

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, and System.Max\_Mantissa. Other implementation-defined limits are implicit in normal ‘First and ‘Last attributes of language-defined (sub) types, such as System.Priority'First and System.Priority'Last. Furthermore, the implementation-defined representation aspects of types and subtypes can be queried by language-defined attributes. Thus, code can be parameterized to adjust to implementation-defined properties without modifying the code.

### 6.57.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.57.5 of ISO/IEC 24772-1.
* Be aware of the contents of Annex M of ISO/IEC 8652 [2] and avoid implementation-defined behaviour whenever possible.
* Verify that the values of implementation-defined constants used by a SPARK Analyzer exactly match those used by the compiler.
* 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.

## 6.58 Deprecated Language Features [MEM]

### 6.58.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.58 is mitigated by SPARK.

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

### 6.58.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.58.5 of ISO/IEC 24772-1.
* Use pragma Restrictions (No\_Obsolescent\_Features) to prevent the use of any obsolescent features.
* Refer to Annex J of the ISO/IEC 8652 to determine whether a feature is obsolescent.

## 6.59 Concurrency – Activation [CGA]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.59 does not apply to SPARK, because SPARK’s concurrency is restricted to Ada’s Ravenscar Tasking Profile[4]. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. Therefore, all resources required for task activation are allocated before the main program begins, and failure in activation will result in exceptions in the main program.

## 6.60 Concurrency – Directed termination [CGT]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.60 does not apply to SPARK, because SPARK concurrency is restricted to Ada’s Ravenscar Tasking Profile[4]. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. In addition, the Ravenscar Tasking Profile prohibits the “abort” statement, and Ravenscar tasks never terminate before the main program ends, hence directed termination is not possible, the resources are not freed, and there is no risk of claiming a terminated task’s resources. Tasks may be effectively removed from consideration by reducing their priority to below that of the idle task, thereby preventing execution.

## 6.61 Concurrent Data Access [CGX]

### 6.61.1 Applicability to language

SPARK’s concurrency is restricted to Ada’s Ravenscar Tasking Profile[4]. Under this profile and SPARK, tasks communicate exclusively using atomic objects, suspension objects, or a limited form of protected objects. A SPARK analyzer is required to enforce these restrictions. Therefore, *data races* are eliminated.

Nevertheless, it is still possible for a program to exhibit a *race condition* with Atomic objects. Consider code that increments an Atomic Integer variable X:

X := X + 1;

This operations involves reading, incrementing, and writing the object. While the read and write operation are individually Atomic, this sequence of actions can still suffer interference from another task.

Such operations must be programmed using a protected object, which guarantee mutually exclusive access to the protected data for an entire sequence of statements.

### 6.61.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.61.5 of ISO/IEC 24772-1.
* Use a SPARK Analyzer to statically determine that no unprotected data is used directly by more than one task.
* Use the aspects Atomic andAtomic\_Components to ensure that all updates to objects and components happen atomically.
* Use the aspects Volatile and Volatile\_Components to ensure that all tasks see updates to the associated objects or array components in the same order.

## 6.62 Concurrency – Premature Termination [CGS]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.62 does not apply to SPARK, because SPARK’s concurrency is restricted to Ada’s Ravenscar Tasking Profile[4]. Under this profile, all tasks are declared in library-level packages and are elaborated before the main program begins. In addition, the Ravenscar Tasking Profile prohibits the “abort” statement, and Ravenscar tasks never terminate, hence premature termination is not possible, the resources are not freed and there is no risk of claiming a terminated task’s resources. Mandatory static verification of type safety in SPARK prevents a task from terminating owing to an unhandled exception. Tasks may be effectively removed from consideration by reducing their priority to below that of the idle task, thereby preventing execution.

## 6.63 Lock Protocol Errors [CGM]

### 6.63.1 Applicability to language

SPARK is open to the errors identified in this vulnerability but supports a number of features that aid mitigation.

* Concurrent programming in SPARK is limited to Ada’s Ravenscar Profile[4].
* SPARK tasks are created before the main subprogram begins execution, never terminate and cannot be aborted.
* SPARK provides protected objects that provide single-threaded access to shared data contained in those objects as well as providing scheduling mechanism for tasks to be suspended upon a ‘protected entry’
* The protocol for controlling access to protected objects is implemented by the run-time library and/or the underlying operating system, and is not visible to the programmer.
* SPARK and the Ravenscar Profile employ a regime for task scheduling and priority assignment that is free from Deadlock.
* SPARK programs using the Ravenscar Profile are amenable to static verification of worst-case execution time, response time, and schedulability.

### 6.63.2 Guidance to language users

* Follow the mitigation mechanisms of subclause 6.63.5 of ISO/IEC 24772-1.
* Make use of loosely coupled communication using protected objects.
* Stay within the constraints defined by the Ravenscar Tasking profile [2].
* Use a SPARK Analyzer to verify type safety for all code, including protected operation bodies.

## 6.64 Uncontrolled Format String [SHL]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.64 does not apply to SPARK, because neither SPARK nor any of its predefined libraries use format strings.

## 6.65 Modifying Constants [UJO]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.65 does not apply to SPARK, because SPARK does not permit constant objects to be modified after they have been initialized.

# 7. Language specific vulnerabilities for SPARK

This clause is intentionally left blank.

# 8. Implications for standardization

In the future, SPARK will continue to evolve in line with ISO/IEC 8652, and will take advantage of appropriate developments in Ada. As such, clause 8 of ISO/IEC 24772-2 applies.

Mandatory requirements for static verification in SPARK may be extended as verification tools and algorithms improve.

# Bibliography

**New Bibliography created by RCC 15th Jan 2020. Only contains documents actually cited in the text.**

[1] SPARK *2014 Reference Manual Release 2020*. AdaCore and Altran UK, April 2020. Available from <https://www.adacore.com/papers/spark-2014-reference-manual-release-2020>

[2] ISO/IEC 8652:2012, *Information technology — Programming languages —* Ada. Available from <http://www.ada-auth.org/standards/ada12_w_tc1.html>

[3] ISO/IEC 60559:2020, *Information Technology — Microprocessor Systems — Floating-point arithmetic.*

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

**Old bibliography below. Retained here for now so entries can be cut and pasted as required. Will be deleted eventually.**

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

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

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

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

[8] ISO/IEC 1539-1:2010, *Information technology — Programming languages —* Fortran — Part 1: Base language

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

[10] ISO/IEC 14882:2011, *Information technology* — *Programming languages —* C++

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

[SB] Chapin, MacCormick J., Building High Integrity Applications with SPARK"

[SRM] SPARK 2014 Reference Manual, Release 2020. AdaCore and Altran UK, June 2020. <https://www.adacore.com/papers/spark-2014-reference-manual-release-2020>

[IFA] “Information-Flow and Data-Flow Analysis of while-Programs.” Bernard Carré and Jean-Francois Bergeretti, ACM Transactions on Programming Languages and Systems (TOPLAS) Vol. 7 No. 1, January 1985. pp 37-61.

[LSP] “A behavioral notion of subtyping.” Barbara Liskov and Jeannette Wing. ACM Transactions on Programming Languages and Systems (TOPLAS), Volume 16, Issue 6 (November 1994), pp. 1811 - 1841.

[12] Motor Industry Software Reliability Association. *Guidelines for the Use of the C Language in Vehicle Based Software*, 2012 (third edition)16F[[1]](#footnote-1).

[13] ISO/IEC TR24731–1, *Information technology — Programming languages, their environments and system software interfaces — Extensions to the C library — Part 1: Bounds-checking interfaces*

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

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

[16] Motor Industry Software Reliability Association. *Guidelines for the Use of the C++ Language in critical systems*, June 2008

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

[18] L. Hatton, Safer C: developing software for high-integrity and safety-critical systems. McGraw-Hill 1995

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

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

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

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

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

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

[26] *ARIANE 5: Flight 501 Failure*, Report by the Inquiry Board, July 19, 1996 <http://esamultimedia.esa.int/docs/esa-x-1819eng.pdf>

[27] Hogaboom, Richard, *A Generic API Bit Manipulation in C*, Embedded Systems Programming, Vol 12, No 7, July 1999 <http://www.embedded.com/1999/9907/9907feat2.htm>

[28] Carlo Ghezzi and Mehdi Jazayeri, Programming Language Concepts, 3rd edition, ISBN-0-471-10426-4, John Wiley & Sons, 1998

[29] Lions, J. L. [ARIANE 5 Flight 501 Failure Report](http://en.wikisource.org/wiki/Ariane_501_Inquiry_Board_report). Paris, France: European Space Agency (ESA) & National Center for Space Study (CNES) Inquiry Board, July 1996.

[30] Seacord, R. *Secure Coding in C and C++*. Boston, MA: Addison-Wesley, 2005. See <http://www.cert.org/books/secure-coding> for news and errata.

[31] John David N. Dionisio. Type Checking. <http://myweb.lmu.edu/dondi/share/pl/type-checking-v02.pdf>

[32] MISRA Limited. "[MISRA C](http://www.misra.org.uk/): 2012 Guidelines for the Use of the C Language in Critical Systems." Warwickshire, UK: MIRA Limited, March 2013 (ISBN 978-1-906400-10-1 and 978-1-906400-11-8).

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

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

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

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

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

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

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

[40] CERT. *CERT C++ Secure Coding Standard*.  [https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637](https://www.securecoding.cert.org/confluence/pages/viewpage.action?pageId=637%20) (2009).

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

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

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

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

[48] GNU Project. GCC Bugs “Non-bugs” http://gcc.gnu.org/bugs.html#nonbugs\_c (2009).

# Index

Aspects

Atomic, 45

Atomic\_Components, 45

Convention, 39

Export, 39

Import, 39

Volatile, 45

Volatile\_Components, 45

Atomic, 45

Attribute

'First, 29, 43

'Image, 27

'Last, 29, 43

'Length, 29

'Range, 29

'Valid, 38

Case statement, 20, 27

CGM – Lock Protocol Errors, 46

CGS – Concurrency – Premature Termination, 45

Exception, 37, 40, 43

Constraint\_Error, 43

Storage\_Error, 31

Identifier length, 23

International character sets, 23

Language Vulnerabilities

Concurrency – Premature Termination [CGS], 45

Lock Protocol Errors [CGM], 46

Modifying Constants [UJO], 47

Uncontrolled Format String [SHL], 46

Mixed casing, 22

Postconditions, 37, 38

Pragma, 40

pragma Restrictions, 41, 42, 44

pragma Suppress, 40

Preconditions, 37, 38

SHL – Uncontrolled Format String, 46

Singular/plural forms, 22

Type invariants, 37, 38

UJO – Modifying Constants, 47

Underscores and periods, 22

Volatile, 45

1. The first edition should not be used or quoted in this work. [↑](#footnote-ref-1)