ISO/IEC JTC 1/SC 22/WG23 ~~N1057~~N10806

Date: 2021-~~03-12~~04-30

ISO/IEC 24772–6

Commented [pb2R1]: Fixed: Double spaces removed

Style Definition: Index 1: Tab stops: 2.99", Right,Leader: ...

Commented [pb3]: Tucker Taft: General ISO rule: avoid using "e.g." and instead use "for example":

"e.g." appears in various other places in the document -- a search should find them all.

Commented [pb4R3]: Fixed: All "e.g."s changed to "for examples"

Style Definition: Index 2: Tab stops: 2.99", Right,Leader: ...

Commented [pb5]: Son Hoang: Consistency for references to ISO/IEC 24772-1. Currently, mixture of "ISO/IEC 24772-1" and "ISO/IEC 24772-1:2019" are used. This need to be consistent over the whole document. No preferences here, but If the year number is used, I assume that it will be updated to "ISO/IEC 24772-1:2021".

Commented [pb6R5]: For WG23 to decide

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 26 April 2021:
Stephen Michell – Convenor
Paul Butcher - AdaCore
Erhard Ploedereder
Tullio Vardanega - Italy

Regrets – Rod Chapman – AdaCore

Results of the meeting are captured in the tracked changes and commented regions in this document.
The major remaining work is

1. Create a “top 10” table for SPARK. The major guidances should be around running the SPARK analyser and handling issues arising from analysis; Dealing with non-SPARK code; and managing special cases such as using the analyser to justify suppression of checks.
2. Create a compliance statement, likely based on 24772-2 Ada’s.
3. Complete the Index.
4. Negotiate WG 9’s review of 24772-6.

Commented [pb7]: John Barnes

Baseline Edition ISO/IEC 24772–6

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

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

Warning

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

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

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

Contents

Commented [pb8]: Tucker Taft: ToC - Bold font usage is weird

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.

Commented [RCC9]: Is this bit really needed if this document becomes an IS?

Commented [SM10R9]: Unfortunately, yes. This is ISO boilerplate.

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 11 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;
  - [UJO] Modifying constants
- Guidance material for each vulnerability given in subclause 6.X.2 is reworded to be more explicit and directive.

Additional material has been ~~added~~ included for some vulnerabilities to reflect additional knowledge gained since the publication of ISO/IEC 24772-2

Commented [pb11]: Tucker Taft

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 document can also be used in comparison with companion standards and with the language-independent standard, ISO/IEC 24772–1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

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

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

1. Scope

This document specifies software programming language vulnerabilities to be avoided in the development of systems where assured behaviour is required for security, safety, mission-critical and business-critical software. In general, this guidance is applicable to the software developed, reviewed, or maintained for any application.
This document describes the way that the vulnerabilities presented in the language-independent ISO/IEC 24772–1 manifest in SPARK.

This document 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 document.

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 used 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

Commented [pb12]: Tucker Taft

3. Terms and definitions, symbols and conventions

Commented [pb13]: Tucker Taft: It is weird that clause headings and subclass headings are in different fonts (one sans-serif, the other with serif)

Commented [pb14R13]: For WG23 to decide

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 and the following apply. Other terms are defined where they appear in italic type.

See clause 2. Normative references~~2. Normative references~~, plus the bibliography.

In the body of this annex, the following document is referenced using the short abbreviation that introduces the 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. Using this document

ISO/IEC 24772-1:20xx clause 4.2 documents the process of creating software that is safe, secure and trusted within the context of the system in which it is ~~fielded~~used. The SPARK programming language was explicitly designed for safety, security and the early elimination of errors from SPARK programs. Nevertheless, as this document shows, vulnerabilities exist in the SPARK programming environment, and organizations are responsible for understanding and addressing the programming language issues that arise in the context of the real-world environment in which the program will be fielded.

Commented [pb15]: John Barnes: Opening paragraph uses the phrases "in which it is fielded" and "in which the program will be
Fielded" Is this a recognized use of fielded? I would have just said "in which it is used" and so
on.

Organizations following this document, meet the requirements of clause 4.2 of ISO/IEC 24772-1, repeated here for the convenience of the reader:

1. Identify and analyze weaknesses in the product or system, including systems, subsystems, modules, and individual components;
2. Identify and analyze sources of programming errors;
3. Determine acceptable programming paradigms and practices to avoid vulnerabilities using guidance drawn from clauses 5.3 and 6 in this document;
4. Determine avoidance and mitigation mechanisms using clause 6 of this document as well as other technical documentation;
5. Map the identified acceptable programming practices into coding standards;
6. Select and deploy tooling and processes to enforce coding rules or practices;

7. Implement controls (in keeping with the requirements of the safety, security and general requirements of the system) that enforce these practices and procedures to ensure that the vulnerabilities do not affect the safety and security of the system under development.

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

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

Commented [pb16]: Tucker Taft

Commented [pb17]: Tucker Taft

Commented [pb18]: John Barnes: The following two paragraphs are curious.

2
Tool vendors follow this document by providing tools that diagnose the vulnerabilities
described in this document. Tool vendors also document to their users those vulnerabilities
that cannot be diagnosed by the too.
Programmers and software designers follow to this document by following the architectural
and coding guidelines of their organization, and by choosing appropriate mitigation
techniques when a vulnerability is not avoidable.
These paragraphs need to be improved. Perhaps adhere would be better than follow. And
"too" at end of first paragraph should be "tools". So I suggest something like
"Tool vendors should adhere to this document by providing tools that diagnose the
vulnerabilities described in this document. Tool vendors should also document for their users
those vulnerabilities that cannot be diagnosed by the tools.
Programmers and software designers should adhere to this document by following the
architectural and coding guidelines of their organization, and by choosing appropriate
mitigation techniques when a vulnerability is not avoidable."
But it's still a bit strange.

Formatted: Tab stops: 4.74", Left

5. Language concepts, common guidance

5.1 Language concepts

5.1.1 SPARK language design

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. As a subset of Ada, SPARK shares the applicable vulnerabilities of Ada. However, beyond enforcing the Ada rules~~–~~ about soundness and the subset restrictions, SPARK programs are also subjected to mandatory static analyses, which prevent vulnerabilities present in Ada.

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.

5.1.2 Soundness

Soundness 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 behaviours in programming languages. Claims of soundness made by tool vendors should be carefully evaluated to verify that they are reasonable for a particular language, compiler~~s~~ and target machine~~s~~. Soundness claims are always underpinned by assumptions (for example, regarding the reliability of memory, or the correctness of compiled code) that should also be validated by users for appropriateness ~~to~~ in their situation.

Commented [pb19]: Tucker Taft

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.

Note: There is also the concept of~~–~~ unsound in practice owing to defects in the implementation of analysis tools. Users should seek evidence to support any soundness claim made by language designers and tool vendors.

The single overriding design goal of SPARK is the provision of a static analysis framework which is sound in theory.

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. We say that SPARK mitigates a particular vulnerability if, between the SPARK analyses and user action, the vulnerability can be identified and avoided.

5.1.3 SPARK Analyzer

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

Commented [pb20]: John Barnes

Commented [pb21]: •Tucker Taft: A *SPARK Analyzer* [to be]{is} a tool that implements the various forms of static analysis required by the SPARK language definition. Without having been analyzed by a SPARK Analyzer, a program cannot reasonably be claimed to be SPARK, much in the same way as a compiler checks the static semantic rules of a standard programming language.

[...] mean delete, {...} means insert.

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 most runtime errors that would raise a predefined exception in Ada, such as buffer overflow, division-by-zero, and arithmetic overflow.

In addition to the language analysis, SPARK supports the static analysis of user-written preconditions, postconditions, loop invariants, type invariants and assertions that allow verification beyond the scope of the mandatory analysis. The use of such user-written assertions is optional, as is the application of some analyses. The most notable example of an optional analysis in SPARK is the generation and proof of verification conditions for user-defined contracts. Optional analyses may provide greater depth of analysis, protection from additional vulnerabilities, and functional proofs of correctness.

5.1.4 Static type safety

ISO/IEC 24772-1, clause 6.2.3, defines:

“The type system of a language is the set of rules used by the language to structure and organize its collection of types. Any attempt to manipulate data objects with inappropriate operations is a type error. A program is said to be type safe (or type secure) if it can be demonstrated that it has no type errors.”

It also notes that most languages enforce their type system with a mix of both static (i.e. prior to program execution) and dynamic (i.e. during program execution) verification, but leaves it to the language-specific Parts to define the notions for each language.

The notion of “type safety” for a particular language therefore depends on the definition of “appropriate operations” for all types, when the rules are checked (statically or dynamically) and what happens when a dynamic check fails.

Ada (SPARK’s parent language) provides a hybrid model for type safety, in that:

- Some typing rules are required to be checked statically (by a compiler). Failure to meet these rules prevents compilation and deployment of a program.
- ~~Some typing rules are checked dynamically, such as checks on downcasts of an object of tagged type.~~ Some typing rules are checked dynamically, such as the checks associated with a type conversion from some tagged type to a descendant of that tagged type.
- Failure of such a runtime check in Ada is required to raise an exception and the programmer has the option of adding exception handlers to catch and respond to these.

Commented [pb22]: John Barnes: please clarify downcast in Some typing rules are checked dynamically, such as checks on downcasts of an object of
tagged type. Failure of such a runtime check in Ada is required to raise an exception and the
Clearly this is talking about Ada, but Ada does not use the term cast for type conversions.

Commented [pb23R22]: **** NEEDS RESOLVING ****

Commented [pb24R22]: Resolved

SPARK goes further. It strengthens Ada’s existing typing rules to verify by static analysis the absence of all runtime type errors. A SPARK program that has met this depth of verification and is free from unsafe programming techniques (see subclause 6.53) is said to be statically type safe, meaning that any execution of the verified program:

- Will not exhibit undefined behaviour; and
- Will not enter a state that would require a predefined exception to be raised.

5.1.5 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 ~~mandatory~~ static analysis tool to their code.
2. The user fails to review or mis-interprets the output of static analysis.

Commented [pb25]: John Barnes:
One might argue that if something is mandatory then the user cannot fail to apply it. So
maybe omit mandatory. And misinterprets is one word.

In the discussion of specific vulnerabilities in clause 6, this document assumes that all proof obligations have been successfully discharged via a SPARK Analyzer. It is also assumed that pragma Assume (an explicitly unsafe construct that can be used to "prove" things that are not true) is not used. It is also assumed that any non-SPARK code in the closure of the program does nothing to invalidate the guarantees that are ensured for "proven" 100% SPARK code.~~In the discussion of specific vulnerabilities in clause 6, this document assumes that a user has sufficient expertise to apply a SPARK Analyzer and interpret the results correctly.~~

Commented [pb26]: With that change, we don't need to talk about pragma Assume in 6.53.

In addition, Ada allows the exception Storage_Error to be raised during the execution of any construct if there is insufficient storage for that execution (Ada RM 11.1(6)). It is also assumed that this possibility has been ruled out using other static analysis techniques and tools. Such tools are available and are commonly used in practice.

Commented [pb27]: Is this acceptable? If so it allows us to remove all mention of Storage_Errors.

I'm not sure how I feel about bringing in “other static analysis techniques/tools of some sort”...

For WG23 to resolve.

Commented [pb28R27]: Resolved: removed “”of some sort” added “ Such tools are available and are commonly used in practice.”

Commented [pb29R27]:

5.1.6 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. See 6.53.
- 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.
- Interfacing a SPARK program with code written in other languages (except Ada); for associated vulnerabilities see 6.47.
- 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]~~6.52 Suppression of language-defined run-time checking [MXB]~~.

T~~We call t~~he use of these language features is called unsafe programming.

Commented [pb30]: John Barnes

Formatted: Normal, No bullets or numbering

5.1.7 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 ~~Owned~~Owner or an Observer of the designated memory, but not both. 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.

Commented [pb31]: Tucker Taft

An Observing access value has read-only permission on an object, and several such observers are allowed to exist.

Any one area of allocated memory has exactly one owner, or one or more observers, but not both, so there can be no aliasing effects by assignments.

~~Because~~ As a consequence of the above rules, SPARK avoids all aliasing effects in a program.

Commented [pb32]: John Barnes

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].

5.2 Top avoidance mechanisms

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	Use a SPARK Analyzer to perform mandatory static verification{XE "static verification"} of all SPARK language rules, including type safety.	All
2	Develop, document and deploy a process for managing false-positive results that arise from static verification.	All
3	Develop, document, and deploy an automated process that prevents building and deployment of an application if static verification goals are not met.	All
4	Do not use features explicitly identified as unsafe (including `Unchecked_Conversion`, mixed-language programming, and `pragma Assume`) unless absolutely necessary and then with extreme caution.	6.53 [SKL], 6.14 [XYK],6.37 [AMV],6.47 [DJS],6.52 [MXB]
5	Use the type system of SPARK and contracts (including preconditions, postconditions, assertions, subtype predicates and type invariants) to specify and enforce constraints on data and formal parameters.	6.2 [IHN], 6.32 [CSJ], 6.34 [OTR], 6.44 [BKK], 6.46 [TRJ]
6	Document all implementation-defined behaviour that an application depends on, and verify that the	6.57 [FAB]

	behaviour implemented by a compiler matches that expected or assumed by a SPARK Analyzer.	
7	Use pragma Restrictions to prevent the use of language features not required by an application (~~e.g.~~for example recursion, tasking or floating point types), to prevent unspecified behaviour, and to prevent the use of specific attributes and predefined packages.	6.35 [GDL], 6.37 [AMV], 6.53 [SKL], 6.55 [BQF]
8	Use the ‘Valid attribute to check the value returned from any call to Unchecked_Conversion or any value returned from non-SPARK code.	6.37 [AMV], 6.47 [DJS]
9	Whenever possible, use the attributes 'First, 'Last , and 'Range 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, case expressions, and aggregates, do not use the others choice.	6.5 [CCB], 6.27 [CLL]

Commented [pb33]: Tucker Taft

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.~~

Commented [RCC34]: Fix these items when the table above is complete

As stated above, every guidance provided in this clause, and in the corresponding Part 6 clause, is supported by material in clause 6 of this document. Clause 6 subclauses also contain 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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.2 is mitigated by SPARK, because SPARK is designed to offer strong, and wholly static type safety.

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]~~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]~~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 alarms.
- 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.
- A SPARK-Analyzer will not detect lacking or inappropriate uses of the type system, for example~~e.g.~~, modeling meters and feet as subtypes of Integer.

6.2.2 Guidance to language users

- Follow the guidance of ISO/IEC 24772-2 (Ada) 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: 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

In general, the type system of SPARK mitigates the vulnerabilities outlined in subclause 6.3 of ISO/IEC~~–~~ 24772-1:2019. The vulnerabilities caused by the inherent conceptual complexity of bit level programming are as described in subclause 6.3 of ISO/IEC~~–~~ 24772-1:2019.

For the traditional approach to bit level programming, SPARK provides modular types and literal representations in arbitrary bases from 2 to 16 to deal with numeric entities and correct handling of the sign bit.
Specifying a value of 1 for the Component_Size aspect of an array-of-Boolean type provides~~The use of **pragma** Pack on arrays of Booleans provides~~ a type-safe way of manipulating bit strings and eliminates the use of error-prone arithmetic operations.

Commented [PR35]: Component_Size is far better because the request must be honored, or the program rejected. In contrast, the pragma Pack is just a request for the compiler's best effort.

Commented [pb36R35]: **** NEEDS RESOLVING ****

Commented [pb37R35]: Resolved

6.3.2 Guidance to language users

Follow the guidance of ISO/IEC 24772-2 (Ada) clause 6.3.2

6.4 Floating-point arithmetic [PLF]

6.4.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.4 applies to SPARK~~,~~ in the same way that it applies to Ada. 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.
- 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.~~for example, rounding mode) made by that implementation.
- Verify and document that the SPARK Analyzer in use makes the same implementation-defined choices for verification as the target compiler and run-time 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 (Not a Number).

Commented [pb38]: John Barnes: Should there be a reference to NaN? Or clarify Not A Number.

Commented [pb39R38]: Fixed

6.5 Enumerator issues[CCB]

6.5.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.5 is mitigated by SPARK, because SPARK requires mandatory verification of type safety for enumeration~~ed~~ types, and through SPARK's restrictions on the use of `Unchecked_Conversion`.

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 to the use of non-standard representation clauses with enumeration~~ed~~ types do not apply to SPARK, since ~~SPARK's~~ the semantics of SPARK are independent of representation values.

Commented [pb40]: John Barnes: Insert "to" thus …Vulnerabilities relating to the use…
I would also prefer: "…since the semantics of SPARK are independent…"

Commented [pb41R40]: Fixed

Vulnerabilities relating to `Unchecked_Conversion` of enumeration~~ed~~ 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].

Commented [pb42]: John Barnes: Uses "enumerated types" several times as well as enumeration types. The Ada/Spark term is
enumeration types.

Commented [pb43R42]: Fixed

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 languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

6.8 Buffer boundary violation [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]~~6.9 Unchecked array indexing [XYZ]~~ and 6.10 Unchecked array copying [XYW]~~6.10 Unchecked array copying [XYW]~~).

6.9 Unchecked array indexing [XYZ]

6.9.1 Applicability to language

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

6.9.2 Guidance to language users

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, because 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]

6.13.1 Applicability to language

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.

6.13.2 Guidance to language users

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.
- Access values cannot be communicated between tasks owing to SPARK's ownership and volatility rules.

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.17 applies to SPARK. This vulnerability does not address overloading, which is covered in 6.20 Identifier name reuse [YOW]~~6.20 Identifier name reuse [YOW].~~

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].~~

Field Code Changed

ISO/IEC 24772-1 documents the risk of confusion by the use of similar names that can 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.
  - Underscores. 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 (that is punctuation stops). 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.

Commented [pb44]: John Barnes: Confirm that period is a punctuation stop and not a time interval. Perhaps by saying Periods (that is stops) in SPARK ...

Commented [pb45R44]: Fixed

- 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:

a. Declares such similar names to have different types in which case the type system will guarantee safe usage; or
b. 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 as having external effects through the use of volatile. 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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.19 is mitigated by SPARK analyzers which identify unreferenced variable declarations and ineffective formal parameters of subprograms.~~is mitigated by SPARK analyzers which identify:~~

~~Variables which are declared but not referenced.~~
~~Variables which are assigned to, but the resulting value is not used in any way that affects an output of the enclosing subprogram.~~

Commented [pb46]: Chris Martin: The second bullet point in here looks to me like the "dead store"
case (which is 6.18 and yes, SPARK identifies all ineffective assignments).
SPARK can also detect the case where an "in out" formal parameter of a
subprogram is neither used in a way that affects the output of the subprogram,
nor is it written to (i.e. SPARK doesn't just check "in" parameters for this
vulnerability.)

Commented [pb47R46]: **** NEEDS RESOLVING ****

Commented [pb48R46]: Resolved

~~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 and take appropriate action to remove or justify any discovered issues.

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 if the two objects have the same or compatible type. 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 potential hiding also takes the signatures of the entities into account. Entities with different signatures~~,~~ ~~therefore,~~ do not hide each other.

Commented [pb49]: John Barnes

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 24772-2 (Ada).

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~~ must disambiguate the name usage by using an expanded name that identifies the exporting package.

Commented [pb50]: Chris Martin: I believe that the programmer *must* disambiguate the name usage in SPARK.

Commented [pb51R50]: Fixed

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 which ensures the presence of initialization before use. Additionally, in SPARK a variable must be initialized with a value which is legal for its type and subtype (if any).

Note: Variables that are declared to be `external` are assumed to be initialized externally.

6.23 Operator precedence and associativity [JCW]

6.23.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.23 is mitigated by SPARK.

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.

In addition, bitwise operations can only be performed on variables of modular type. This mitigates the problem by making an entire class of failure mechanism only applicable to a rarely used type. Also mitigating the problem is that the result of binary comparison operations (<, <=, >, >=, =, /=) is type Boolean; no predefined binary operators can be used on expressions involving two different types. Hence even if variable 'x' is modular type, the Ada-equivalent of the expression in the example from ISO/IEC 24772-1 subclause 6.23 (for example, x and 1 = 0) will fail because the SPARK Examiner will detect that the higher precedence operation "1 = 0" produces a Boolean result, which is of a different type to x.

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.

Commented [pb52]: Chris Martin: It may be worth stating that additionally, in SPARK a variable must
be initialized with a value which is legal for its type and subtype (if any).

Commented [pb53R52]: Fixed

Commented [pb54]: John Barnes: This needs clarification perhaps.

Commented [pb55R54]: **** NEEDS RESOLVING ****

Commented [pb56]: Chris Martin: It also may be worth stating that bitwise operations can only be performed on variables of modular type (mitigating the problem by making an entire class of
failure mechanism only applicable to a rarely-used type.)
Also mitigating the problem is that the result of binary comparison operations
(<, <=, >, >=, =, /=) is type Boolean; no predefined binary operators can be
used on expressions involving two different types.
Hence even if variable 'x' is modular type, the Ada-equivalent of the expression
in the example from ISO/IEC 24772-1 subclause 6.23, i.e.
x and 1 = 0
will fail because the Examiner detects that the higher-precedence operation
"1 = 0" produces a Boolean result, which is of a different type to x.

Commented [pb57R56]: Fixed – but should be checked

Formatted: Comment Text, Space Before: 0 pt, After: 0 pt, No bullets or numbering

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 except volatile functions (and hence function calls) are free from side-effects.
    - o Note that functions which access volatile data are themselves volatile, and must be declared with the Volatile aspect. SPARK has rules which constrain the use of volatile data and volatile functions (see SPARK LRM section 7.1.3) which prevent volatile functions being used in an unsafe manner or ambiguity being introduced to an expression.
    - o Also note that Ada (and by inference SPARK) have no builtin "increment" or similar operators.
- Assignment is a statement, not an expression.
- In SPARK, expression evaluation order is unspecified, but the language design requires mandatory static verification that, for any possible evaluation order, all intermediate expressions are type safe, and the expression yields the same result, except for rounding errors of floating-point arithmetic.

Commented [pb58]: Chris Martin: The statement "In SPARK, all functions (and hence function calls)
are free from side-effects" has one exception: volatile functions. (Functions
which access volatile data are themselves volatile, and must be declared with
the Volatile aspect.) SPARK has rules which constrain the use of volatile data
and volatile functions (see SPARK LRM section 7.1.3) which prevent volatile
functions being used in an unsafe manner or ambiguity being introduced to an
expression.
Also note that Ada (and by inference SPARK) have no builtin "increment" or
similar operators.

Commented [pb59R58]: Fixed – but needs checking

Formatted

6.25 Likely incorrect expression [KOA]

6.25.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.25 is mitigated by SPARK.

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~~ the successful evaluation of that expression 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.

6.26 Dead and deactivated code [XYQ]

6.26.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.26 is mitigated by SPARK.

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 24772-2 (Ada).

6.27 Switch statements and static analysis [CLL]

Commented [pb60]: Chris Martin: The wording of the last sentence in 6.27.1 paragraph 2 is not quite correct.
I recommend simply striking out "but not at the first or last of that type" from "Likewise, the inclusion (say, during maintenance) 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), can hide..."
Similarly, section 6.27.2 the second bullet-point should remove the "anywhere but the beginning or the end of the enumeration type definition" clause.
DETAIL:
It *is possible* to add a value to a defined/internal range by adding a value to an enumerated type as the first or last value of the type. This is because it is possible to define a sub-range using the 'First and 'Last attributes of the enumeration type.
It's easiest to show this by example:

```
type Colours_T is (Red, Orange, Yellow, Green, Blue, Violet);
subtype Low_Wavelength is Colours_T range Colours_T'First .. Yellow;
My_Colour : Colours_T;
...
case My_Colour is
  when Low_Wavelength => Do_Something(...);
```

If during maintenance, we add "Black" as a colour e.g.

```
type Colours_T is (Black, Red, Orange, Yellow, Green, Blue, Violet);
```

"Black" is processed as being in the "Low_Wavelength" case.

Commented [pb61R60]: Fixed – but needs checking

6.27.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.27 is mitigated by SPARK, 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~~Likewise, the inclusion (say, during maintenance) 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~~), can hide the new~~and some~~ case statement~~s~~ choices ~~hide the addition~~ in the~~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.
- Ada requires that a case statement provides exactly one alternative for each value of the expression's subtype. There are times where it can be statically proven that some values of the the expression's subtype cannot be achieved; in other words, Ada forces the programmer to insert case handlers for provably unachievable cases and thereby insert "dead" code. In these cases, a useful language construct is to insert pragma Assert (False); into the cases which cannot be reached.

Commented [pb62]: Chris Martin: SPARK flow analysis requires that a case statement provides exactly one alternative for each value of the expression's subtype. There are times where it can be statically proven that some values of the expression's subtype cannot be achieved; in other words, the SPARK flow analyzer forces the programmer to insert case handlers for provably unachievable cases and thereby insert "dead" code. In these cases, a useful language construct is to insert pragma Assert (False); into the cases which cannot be reached.

Commented [pb63R62]: Fixed – but needs checking

Commented [pb64R62]: Resolved: changed to reflect that this is a requirement of Ada not SPARK

Commented [pb65R62]:

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.

Commented [pb66]: John Barnes: I don't understand 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. what is a loop variant contract? Please clarify. Maybe this is a recent addition to SPARK.

Commented [pb67R66]: **** NEEDS RESOLVING ****

Commented [pb68R66]: Resolved – replaced "loop variant contract" with "loop_variant pragma"

For more general loops, SPARK also supports the specification and verification of a loop_variant pragma~~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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.30 is mitigated by SPARK, because even though SPARK permits the use of cardinal numbers for indexing arrays and loops,~~.~~ SPARK ~~does, however,~~ provides alternative syntax which, ~~if~~when used~~,~~ dramatically reduces the occurrence of such errors.

Mitigating the ~~C~~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.

Commented [pb69]: Chris Martin

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.

~~C~~Mitigating the 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~~SPARK allows the use of explicit values and loop termination tests. Off-by-one errors can result in these circumstances. Care should be taken when using the `'Length` attribute in the loop termination expression. The expression should generally be relative to the `'First` value. The mitigation is provided by ~~Again~~by the SPARK analyzer which prevents~~,~~ 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 example ~~-~~**`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.

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt, Not Bold

Formatted: Font: (Default) Courier New, 10 pt

Formatted: Font: (Default) Courier New, 10 pt, Not Bold

Formatted: Font: (Default) Courier New, 10 pt

6.31 Unstructured programming [EWD]

6.31.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.31 is mitigated by SPARK, because SPARK forbids some control-flow statements, such as `goto` and exception handlers, and does not provide non-local jumps or subprograms with multiple entries.

Commented [pb70]: Chris Martin: I'm just double-checking, but if this document is specific to
the 2020 release of SPARK then this is fine. Since the 2020 release, "goto" and
some exception handling have become allowed in SPARK (under strictly-controlled
conditions!)

Commented [pb71R70]: **** NEEDS RESOLVING ****

Commented [pb72R70]: Resolved – no change required

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.

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.
- SPARK flow analysis detects and reports when a subprogram does not actually "change data visible to the calling program" (quoted text from ISO/IEC 24772-1 subclause 6.32), or attempts illegal changes to formal/global parameters declared as mode "in".
- 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, the~~n~~ Global Contract is verified by static verification, otherwise it is derived by an analysis of the unit body and all called units.
- 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

Formatted: Font: Bold, Italic

Formatted: Font: Bold, Italic

Formatted: Font: Bold, Italic

Commented [pb73]: Chris Martin: would be clearer if the keywords "in", "out" and "in out"
were written in bold or a different font.
Also note that SPARK flow analysis detects and reports when a subprogram does
not actually "change data visible to the calling program" (quoted text from
ISO/IEC 24772-1 subclause 6.32), or attempts illegal changes to formal/global
parameters declared as mode "in".

Commented [pb74R73]: Fixed

Commented [pb75]: Chris Martin

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.34 does not apply to SPARK except for the case of calls to/from subprograms where the other side is a foreign language. This case is addressed in subclause 6.47 Inter-language calling [DJS]~~6.47 Inter-language calling [DJS]~~

In all other cases, the parameter association is checked at compilation time 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 6.34.2 of ISO/IEC 24772-2.

6.35 Recursion [GDL]

6.35.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.35 is mitigated by SPARK.

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 to handle the exception. For vulnerabilities relating to unhandled exceptions, see subclause 6.36 Ignored

Commented [pb76]: Chris Martin: SPARK now allows 'Access and 'Address, however the constraints on
the use of 'Access (see the SPARK 2020 RM section 3.10, or Claire Dross's blog
post https://blog.adacore.com/using-pointers-in-spark) and to 'Address
(only allowed in representation clauses, see section 15.2 of the SPARK RM)
prevent dangling references to stack frames.

Commented [pb77R76]: **** NEEDS RESOLVING ****

Commented [pb78R76]: I think the "Community 2020" release of SPARK did not support the use of the 'Access or 'Unchecked_Access attributes. Perhaps 'Address was supported, but you can't do anything hazardous with a System.Address value in SPARK (as opposed to in Ada).

Therefore we now think the proposed wording is correct, but this needs to be confirmed.
As with goto statements (and perhaps exception handlers), the situation may
be different for later versions of SPARK.

Commented [pb79]: John Barnes: In this section is subprobprogram meant to be subprogram?

Commented [pb80R79]: Fixed

Commented [pb81]: Son Hoang: + Comment: Subclause 6.34.2, Remove "Follow the mitigation mechanisms of subclause 6.34.2 of ISO/IEC 24772-2.". Add "For the case of calls to/from subprograms where the other side is a foreign language, follow the mitigation mechanisms of subclause 6.47.2"
+ Justification: The vulnerability does not apply to SPARK, except for the other case which is handled in subclause 6.47.

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

Recursion can be forbidden using `pragma Restrictions (No_Recursion)`if it is not required by an application.

SPARK is designed to be amenable to static analysis of worst-case stack usage. In the presence of recursion, a programmer may have to supply additional information to the analysis tool to bound the depth of recursion, and therefore memory consumption.

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.36 is mitigated by SPARK.

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~~ termination or restart.

Commented [pb82]: John Barnes: I do not recognise the term "closeout". does it mean termination?

Commented [pb83R82]: Fixed

The `'Valid` attribute can be used to check the result of `Unchecked_Conversion` or a value read from an external device, and to handle resulting error conditions by explicit code such as an if statement.

SPARK checks that assignments to formal `in out` parameters and `out` parameters are not ineffective assignments, and that function returns are subsequently read (See 6.19 Unused variable [YZS]~~6.19 Unused variable [YZS]~~). Therefore, it is guaranteed that error codes returned via a formal parameter or as a result are inspected.

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 exceptions raised by predefined checks.
- 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 recovery or notification of failure can be performed.

6.37 Type-breaking reinterpretation of data [AMV]

6.37.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.37 does not apply to SPARK, except in the case of easily identifiable unsafe programming. Even in those circumstances, SPARK mitigates the resulting vulnerabilities as follows:

SPARK permits the instantiation and use of Unchecked_Conversion as in Ada, but limits instantiation for a source type S and a target type T as follows:

- Neither S nor T or any component thereof is of a limited type, a tagged type, an access type, or subject to a predicate or type invariant.
- The number of valid values for S must be equal to 2**(S'Object_Size), and
- The number of valid values for T must be equal to 2**(T'Object_Size) ,and
- S'Object_Size is equal to T'Object_Size, so (by implication from the above), the number of valid values for S and T is the same.

Commented [pb84]: John Barnes

Note that these rules exclude all floating-point types, since NaN is not considered a valid value. Array and record types can be used in an instantiation of Unchecked_Conversion if they meet the requirements above, with the number of valid values determined from the types of the fields and component types.

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 'Object_Size attribute is 16 in both cases, and both have exactly 2^{16} valid values. Conversely, an instantiation of Unchecked_Conversion from Interfaces.Unsigned_8 to Boolean is not permitted, since the former has 256 valid values, while the latter only has 2.

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.

For the case of calling on external subprograms written in other languages, see subclause 6.19 Unused variable [YZS]~~6.19 Unused variable [YZS]~~].

Language rules prevent ~~the~~ changinges ~~of~~to a discriminant of a variable unless the whole object is written, so reinterpreting an object's 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` and values from foreign languages or libraries and handle both outcomes.

6.38 Deep vs. shallow copying [YAN]

6.38.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.38 applies to SPARK.

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.

Commented [SM85]: Stephen, Erhard, consider taking this guidance to 24772-2 Ada.

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.39 is mitigated by SPARK.

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.
- Consider a completely static model where all storage is preallocated 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

The vulnerability documented in ISO/IEC 24772-1 subclause 6.41 is mitigated by SPARK.

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

SPARK has no preference rules to resolve ambiguities of calls on primitive operations of tagged types. Hence the related vulnerability documented in ISO/IEC TR 24772-1:2019 subclause 6.41 does not apply to SPARK.

In SPARK, a user can specify if a redefined operation must be `overriding` or must be `not overriding`. When these specifications are given, they are verified statically, so their use prevents vulnerabilities relating to accidental overriding or failure to override.

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. See subclauses 6.43 Redispatching [PPH], and 6.44 Polymorphic variables [BKK].

Commented [pb86]: John Barnes

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

The vulnerability documented in ISO/IEC 24772-1 subclause 6.42 is mitigated by SPARK.

SPARK extends Ada's capabilities in this area, allowing fully static verification of the Liskov Substitution Principle (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.
- Avoid the use of the `Extensions_Visible` aspect if redispatching is to be forbidden.
- If redispatching is necessary, document the behaviour explicitly.

6.44 Polymorphic variables [BKK]

6.44.1 Applicability to language

The vulnerability documented in ISO/IEC 24772-1 subclause 6.44 is mitigated by SPARK.

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~~t~~ to a specific tagged type is permitted in SPARK and can never give rise to a runtime error. By specifying the aspect `Type_Invariant` 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

Commented [pb87]: John Barnes: A typical SPARK user might not understand the terms upcast and downcast since they are not
used by the Ada community. Some clarification should be added. The definition of downcast in my Oxford dictionary is not relevant!

Commented [pb88R87]: **** NEEDS RESOLVING ****

Commented [pb89R87]: those terms are explicitly defined in section 6.44.1 of the base document. Therefore, the objection is not relevant.

No change required.

conversion is complete. If a type invariant is specified, then SPARK requires static verification that it 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 aspect `Type_Invariant` ~~contract~~ to specify and verify the semantic consistency of derived types.

Commented [pb90]: John Barnes: Type_Invariant is an aspect not a contract.

Commented [pb91R90]: Fixed

6.45 Extra intrinsics [LRM]

The vulnerability as described in ISO/IEC 24772-1 subclause 6.45 does not apply to SPARK, because, as in Ada, 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]

6.46.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.46 is mitigated by SPARK.

There are three cases to consider, depending on the language used to implement a particular library being called from SPARK:

Commented [RCC92]: Re-wrote this bit for body in SPARK, Ada, or "other"

- If the library is itself written in SPARK, and is subject to mandatory verification of type safety, then no vulnerability exists.
- If the library is written in Ada (but not meeting the rules of SPARK), then appropriate contracts (~~e.g.~~ for example, preconditions and parameter subtypes) and runtime checks can be used to mitigate this vulnerability.
- If the library is written in a foreign language other than SPARK or Ada, then subclause 6.47 Interlanguage calling [DJS] applies.

6.46.2 Guidance to language users

- 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.
- When a library body is written in Ada, follow the mitigation mechanisms of subclause 6.46.5 of ISO/IEC 24772-2.

Commented [RCC93]: Moved this down to here, and only apply to Ada as per Steve's suggestion.

6.47 Inter-language calling [DJS]

6.47.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.47 applies to SPARK.

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. Other languages can also be called: this is normally achieved using the C calling convention.

Commented [SM94]: Add a note that other languages are usually interfaced through the C language calls.

Commented [RCC95R94]: Done

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. In addition, 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 support 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.

Commented [pb96]: John Barnes: It is generally considered bad to finish a sentence with a preposition such as "with". Better to
phrase it thus
... are written in languages with which ISO/IEC 8652 [2] specifies an interface.
But maybe I am being pedantic.

Commented [pb97R96]: Fixed

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.49 applies to SPARK.

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

The vulnerability as described in ISO/IEC 24772-1 subclause 6.50 applies to SPARK.

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 subprogram to terminate. For the vulnerability of unhandled exceptions, see subclause 6.36 Ignored error status and unhandled exceptions [OYB]~~6.36 Ignored error status and unhandled exceptions [OYB]~~.

Commented [pb98]: John Barnes: The main program is known as the main subprogram in Ada and SPARK.

Commented [pb99R98]: fixed

Commented [p100]: Add: for the vulnerability of unhandled exceptions, see subclause 6.36.

Commented [RCC101R100]: Done

Commented [pb102]: Stephen Baird

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.

If the failure does not fit into the above categories, see ISO/IEC 24772-1 clause 7.31.

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.
- For calling libraries that can raise exceptions, consider “wrapping” these calls in an Ada subprogram that calls the desired subprogram, but catches and handles any exceptions locally before returning a suitable error code to the SPARK caller.
- When calling a function in a foreign language that can raise an exception, handle that exception in the foreign language unit, rather than allowing an exception to propagate from one language to another.
- Consider failure strategies(see ISO/IEC 24772-1 clause 7.xx Failure strategies…) and consider adding Ada code with Ada exception handlers at the top level of all tasks and the main subprogram.

  ~~Note: Since exception declarations are external to SPARK, consider 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, consider 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 as described in ISO/IEC 24772-1 subclause 6.52 is mitigated by SPARK.

~~The vulnerability exists in SPARK since p~~P**ragma** `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]).

Commented [p103]: What about wrapping such calls in Ada code with an exception handler and error status returns, and call the Ada wrapper?

Commented [RCC104R103]: Added bullet 3 here

Commented [SM105]: We need advice about handling local exceptions, such as file-not-found.
Need advice to handle any exception raised in a different language system should be handled in that language system (i.e. write a wrapper)

Commented [RCC106R105]: Added bullet 4 recommending no cross-language exception propagation.

Commented [SM107]: This is false. Exception declarations are legal in SPARK, but cannot be handled inside the language, hence the exception and its top-level handler should both be in a non-SPARK unit.

Commented [SM108]: Needs clarification or move to description. Issues about Ravenscar tasks and top-level wrapper functions that may include exception handlers.

Commented [RCC109R108]: Actually, this is OK. A top-level task body CAN be written in Ada (so can have a top-level handler) calling a “main loop” procedure that is in SPARK. I can supply example code if necessary.

Commented [pb110]: Stephen Baird: When we talk about whether a vulnerability applies to SPARK, are we talking about 100% SPARK with all proof obligations discharged? If so, then why are we saying here that this vulnerability applies to SPARK?

I think the second paragraph is trying to say that if we are talking about 100% SPARK with all proof obligations discharged, then this vulnerability does not apply. No vulnerability is introduced if a user suppresses a check that is statically known to always succeed. The first paragraph says that the vulnerability does apply in other cases.

And that is the order in which they should be presented.

In the Ada annex, the notion of unsafe programming was introduced; this includes things like Unchecked_Conversion. It is explicitly stated "up front" that properties such as strong typing can be circumvented through such techniques. This approach eliminates the need to then mention unsafe programming in the context of each specific vulnerability, although ... [1]

Commented [pb111R110]: **** NEEDS RESOLVING ****

Commented [pb112R110]: If we are talking about 100% SPARK with all proof obligations satisfied and no use of pragma Assume or any other such cheats, then the vulnerability does not exist in SPARK. As noted in the 3rd paragraph, it is ok to suppress ... [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 to checks that verification has proven to be redundant.

6.52.2 Guidance to language users

- Verify type safety using a SPARK Analyzer.
- Only apply pragma `Suppress` for code fully verified by the SPARK analyzer without reliance on the `Assume` pragma (6.53 Provision of inherently unsafe operations [SKL]~~6.53 Provision of inherently unsafe operations [SKL]~~).
- Follow the mitigation mechanisms of ISO/IEC 24772-1 subclause 6.52.5 when SPARK type safety cannot be guaranteed.

6.53 Provision of inherently unsafe operations [SKL]

6.53.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.53 is mitigated by SPARK.

~~Other than the use of `pragma Assume`, t~~The classes of vulnerability identified in ISO/IEC 24772-1 subclause 6.53 and techniques defined as Unsafe programming in clause 5.1.6 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]~~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]~~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]~~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]~~6.52 Suppression of language-defined run-time checking [MXB]~~.

~~The use of `pragma Assume` can introduce unsoundness into verification because the property being introduced is not required to be verified by a SPARK Analyzer. For example:~~

~~**pragma** Assume (**if** World_Is_Flat **then** Pi = 42);~~

~~would be accepted and used by a SPARK Analyzer, although it may be inconsistent, contradictory or just nonsense. As such, the use of pragma Assume is considered to be an instance of unsafe programming.~~

Commented [p113]: Indeed, strengthen, Probably to "not apply", since the checking is mandatory, right?
Rephrase Suppress as the consequential means in SPARK to tell the Ada compiler not to bother to create checks.

Commented [RCC114R113]: Not sure… what if type safety verification yields false alarms?

Commented [pb115]: Stephen Baird: This section seems like an odd place to talk about pragma Assume.
Formally, a pragma Assume is similar to an Assert pragma that generates an undischarged proof obligation. So why are we talking
about one of these cases here and not the other?

As mentioned earlier (in 6.52), it seems like we want to define somewhere what we mean by unsafe programming and deal with that
once and for all outside of the discussion of any one particular vulnerability.

Commented [pb116R115]: **** NEEDS RESOLVING ****

Commented [pb117]: This can be removed due to the statement in 5.1.5

Commented [RCC118]: Added material on pragma Assume and how it can be abused.

Commented [RCC119]: Not too controversial I hope!

Commented [pb120]: John Barnes: I enjoyed
pragma Assume (if World_Is_Flat then Pi = 42);
but those of a pedantic mind might object in that Pi in Ada.Numerics is a real constant so
perhaps it should be 42.0.
I assume that 42 was chosen since it was a favourite number of Lewis Carroll.

Commented [pb121R120]: Fixed

Commented [pb122]: This can be removed due to the statement in 5.1.5

6.53.2 Guidance to language users

- Use a SPARK Analyzer to identify inherently unsafe operations.
- Avoid the use of unsafe programming practices, unless they are functionally essential.
- 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`.
- Require ~~a~~ manual review ~~process must be used~~ to verify the consistency and truthfulness of any ~~such properties~~property introduced by `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. (see clause 6.47 Interlanguage calling)

Commented [RCC123]: Added this bit.

6.54 Obscure language features [BRS]

In SPARK, assertions can be used as a superior alternative to comments to improve readability. The correctness of an assertion, as opposed to that of a comment, is checked by the SPARK tools.

6.54.1 Applicability to~~f~~ language

Commented [pb124]: Stephen Baird: Is it worth noting that even if code is chock-full of obscure
hard-to-understand code, an AoRTE (absence of runtime errors) proof helps a lot in dealing with the situation.

In particular, the various assertion-related constructs (subprogram
contracts, subtype predicates, Assert pragmas, etc.)
can be used in SPARK as statically-enforced-and-therefore-trustworthy
comments.

Perhaps it would be too hard to state this without sounding like advertising.

Commented [pb125R124]: **** NEEDS RESOLVING ****

Commented [pb126R124]: Resolved with additional statement

Formatted: Normal

Commented [pb127]: John Barnes: These headings have been “Applicability to language” but this has “of” instead of “to”. Is that
an error? It also applies to 6.55.1

Commented [pb128R127]: Fixed

The vulnerability as described in ISO/IEC 24772-1 subclause 6.54 is mitigated by SPARK.

SPARK is designed to offer 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 such a specific 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 `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 `pragma Restrictions (No_Dependence => …)` to prevent the use of specified predefined or user-defined libraries.

6.55 Unspecified behaviour [BQF]

6.55.1 Applicability ~~of~~ to language

Commented [pb129]: Stephen Baird: We might be claiming too much here. SPARK does not ensure deterministic behavior in all cases; SPARK does ensure AoRTE for any possible execution order.

Consider two tasks that are each trying to acquire a lock, generate some output, and release the lock. SPARK does not say anything about which task will acquire the lock first, and therefore about the order of the generated output.

Commented [pb130R129]: **** NEEDS RESOLVING ****

Commented [pb131R129]: Resolved – no change required

Commented [pb132]: John Barnes

The vulnerability as described in ISO/IEC 24772-1 subclause 6.55 is mitigated by SPARK.

The design intent of SPARK is 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.

Three cases remain:

Commented [RCC133]: Revised this bit and below following complete check of all unspecified behaviours

- The result of `S'Machine_Rounding (X)` is unspecified if X lies exactly halfway between two integers.
- Results of certain operations within ~~a~~ language-defined generic packages are unspecified if the actual subprogram associated with a particular formal subprogram does not meet stated expectations (such as "=" providing a true equality relationship)
- Functions declared in the predefined units `Ada.Numerics.Generic_Complex_Types` and `Ada.Numerics.Generic_Complex_Elementary_Functions` exhibit unspecified behaviour relating to overflow (and thus raising of exceptions) for certain arguments.

Commented [pb134]: John Barnes: There is a number clash. Either delete "a" or replace packages with "package".
There is also a double space between program and associated on the next line.

Commented [pb135R134]: Fixed

6.55.2 Guidance to language users

- Follow the mitigation mechanisms of subclause 6.55.5 of ISO/IEC 24772-1.
- Verify and document the behaviour of S'Machine_Rounding for both the compiler and SPARK Analyzer. Alternatively, forbid the use of this attribute using the `No_Use_Of_Attribute` restriction identifier.
- For situations involving generic formal subprograms, ensure that the actual subprogram satisfies all of the stated expectations.
- Document the behaviour of a particular implementation with respect to the `Ada.Numerics.Generic_Complex_Elementary_Functions` and `Ada.Numerics.Generic_Complex_Types` packages, and add user-defined Assertions in calling units to verify the absence of unspecified behaviour and exceptions from any such calls. Alternatively, forbid the use of these units using the pragma Restrictions(`No_Dependence =>` …) restriction identifier.

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, as described in section 5.1.5 of this

Commented [pb136]: Stephen Baird: As discussed in 6.52 above, do we want to drop the "other than specific cases of unsafe programming techniques" qualification here?

Commented [pb137R136]: **** NEEDS RESOLVING ****

Commented [pb138R136]: Resolved

document. Note that ISO/IEC 8652 and SPARK use the term “erroneous behaviour” with the same meaning as “undefined behaviour” used in ISO/IEC 24772-1.

6.57 Implementation–defined behaviour [FAB]

6.57.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.57 applies to SPARK.

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 error or warning, 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 ranges of the predefined Integer types, since these ranges impact the verification of the absence of arithmetic overflow in expressions. Similarly, bounds of some user-defined types (for example, "type T is range 1 .. 10;") are specified by the user, but their basetype bounds are implementation-defined; one case where this can make a difference is in overflow checking.

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. These constants can be referenced in the program and in proofs of correctness to determine statically that the implementation-specified characteristics result in correct programs.

Commented [SM139]: Include the Ada/SPARK term “erroneous behaviour”

Commented [RCC140R139]: Added below.

Commented [pb141]: Stephen Baird: Do we want to mention the portability issue with basetype selection
in a case like
 type T is range 1 .. 10;
? We need consistency between program analysis and program execution with respect to the bounds of T'Base.
If we don't mention this hazard and we offer as mitigation advice "declare your own numeric types", we may be leading users astray.

Commented [pb142R141]: **** NEEDS RESOLVING ****

Commented [pb143R141]: Resolved with the addition of:

“Similarly, bounds of some user-defined types (for example, "type T is range 1 .. 10;") are specified by the user, but their basetype bounds are implementation-dependent; one case where
this can make a difference is in overflow checking.”

Commented [pb144]: John Barnes: For consistency hyphenate implementation-defined.
But maybe this is deliberate when the adjectival phrase is not being used as an adjective in
front of a noun. Mmmm. Hyphens can drive one mad..

Commented [pb145R144]: Fixed

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.
- 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.
- 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.

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. SPARK, like Ada, provides a `pragma Restrictions (No_Obsolescent_Features)` that prevents the use of any obsolescent features within a program.

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]

6.59.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.59 is mitigated by 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.

Commented [pb146]: Stephen Baird: Instead of mentioning Ravenscar, we should be mentioning Ravenscar or Jorvik.

Commented [pb147R146]: **** NEEDS RESOLVING **** What reference would we use?

Commented [pb148R146]: Resolved – no change required

Assuming that mandatory static verification has been performed on all task bodies, a single failure mode remains: unexpected termination of a library level task owing to an exception being raised during its activation. In that case, the behaviour is implementation-defined. Possible behaviours include:

- Termination of the whole program, or
- A user-defined action, such as reset or restart of the target computer, or
- The program keeps running, but missing one or more tasks.

6.59.2 Guidance to language users

- ~~Avoid allocators in the declarative parts of library level tasks, since these might raise `Storage_Error`, and thus result in failure of task activation.~~
- Perform static analysis of worst-case stack usage for all tasks to ensure that memory space allocated to all tasks' stacks is sufficient.
- In the case of unexpected task termination during activation, verify and document the implementation-defined behavior of the implementation.

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 ensures that no tasks terminate.

6.61 Concurrent data access [CGX]

6.61.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.61 is mitigated by SPARK.

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, and therefore prevent data destruction because of a data race.

More specifically, Ada's Ravenscar Tasking Profile[4] does not prevent unsafe concurrent access to an unsynchronized global variable. However, SPARK can identify unsafe concurrent access to an unsynchronized global variable.

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, and X is shared:

```
   X := X + 100;
```

Commented [pb149]: Stephen Baird: Where did the "Possible behaviours" (with non-U.S. spelling for behaviour) list in this section come from? I don't know which Ada-or-SPARK RM rules this is derived from. As far as I know, program behavior is well-defined in the case of an exception that is raised during a task's activation. In addition to mentioning Ravenscar, can't we just say that proof of AoRTE ensures that this situation cannot arise?

Commented [pb150R149]: **** NEEDS RESOLVING ****

Commented [pb151R149]: Resolved – no change required

Commented [pb152R149]:

Commented [pb153]: John Barnes

Commented [pb154]: Stephen Baird: This is not the place to talk about Storage_Error, but doesn't it need to be mentioned somewhere? Perhaps this is more of a question about the language-independent document than the SPARK-specific annex.

Commented [pb155R154]: **** NEEDS RESOLVING ****

Commented [pb156R154]: Resolved: bullet deleted due to addition of para in section 5.1.5

Commented [pb157R154]:

Commented [pb158]: Stephen Baird: Should we mention that SPARK also disallows ATC (see Ada RM 9.7.4)? ATC introduces many of the same hazards as abort statements.

Commented [pb159R158]: **** NEEDS RESOLVING ****

Commented [pb160R158]: Resolved – no change required

Commented [pb161]: Stephen Baird: In Ada, Ravenscar/Jorvik does not prevent unsafe concurrent access to an unsynchronized global variable. In contrast, SPARK does. This significant advantage of SPARK should be mentioned.

Commented [pb162R161]: Fixed.

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

Such interference can be avoided by placing the statement inside a protected subprogram or entry, which guarantee mutually exclusive access to all 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 protected objects in preference to other forms of synchronization such as atomic variables.
- Use a SPARK Analyzer to statically ensure that no unprotected data is used without synchronization by more than one task.
- Use protected objects where atomic access to a simple object is not sufficient or not supported.
- Use the aspects Atomic and Atomic_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 ensures that tasks do not terminate. The mechanisms that might lead to task termination in some other languages (e.g., task abortion, reaching the end of a task body, failure of a run-time check) are prevented statically in SPARK.~~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. Failing run-time checks are prevented in SPARK code by a SPARK Analyzer. The only remaining cause for premature termination is an exception raised by non-Spark code, (see subclause 6.47 Inter-language calling [DJS]).~~

Commented [pb163]: John Barnes: I don't like the use of the apostrophe for the genitive of inanimate objects. My favourite example is the case of the lamb and the leg. The phrase the lamb's leg leads me to expect that
the lamb is alive and perhaps has hurt its leg. However, the phrase the leg of lamb makes me
think of a nice roasted joint of lamb with mint sauce!
In the case here I would prefer to see this reworded as because concurrency in SPARK is restricted to the Ada Ravenscar Profile [4}
Note moreover that ref 4 is entitled ...... of the Ada Ravenscar Profile...
I suggest making this same change in 6.63.1

Commented [pb164R163]: Initial part fixed (removed SPARK's).

Second part not understood ("... of the Ada...")

Commented [pb165]: Stephen Baird: As discussed in 6.59, I hope we are talking about Storage_Error somewhere
since we are (correctly, IMO) not talking about it in this section. But
even if we are not going to explicitly mention Storage_Error, claiming
"The only remaining cause for premature termination is ..." seems dubious.

Commented [pb166R165]: **** NEEDS RESOLVING ****

Commented [pb167R165]: Resolved

Commented [pb168]: Stephen Baird: As discussed in 6.52 above, why are we talking about exceptions raised
by non-SPARK code in this section?

Commented [pb169R168]: **** NEEDS RESOLVING ****
Question for WG23?

Commented [pb170R168]: resolved

6.63 Lock protocol errors [CGM]

6.63.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.63 is mitigated by SPARK.

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 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 guaranteed to be free from circular waits for resources.
- 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 well documented design patterns for creating groups of tasks executing known protocols using Ravenscar [5].

6.64 Uncontrolled format string [SHL] Error! Bookmark not defined.Error! Bookmark not defined.Error! Bookmark not defined.

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. In particular, the Ada vulnerability of modifying constants via access discriminants on limited types does not exist in SPARK because access discriminants are not permitted.

Commented [pb171]: Stephen Baird: Ravenscar => Ravenscar/Jorvik

Commented [pb172R171]: **** NEEDS RESOLVING ****

No reference for Jorvik

Commented [pb173R171]: Resolved – no change

Commented [pb174]: Stephen Baird: I think the claim about SPARK preventing circular waits for
resources is only true on a monoprocessor (it is true in that case because of the Task_Dispatching_Policy and Locking_Policy implied by
Ravenscar/Jorvik). The current wording claims too much, right?

Commented [pb175R174]: **** NEEDS RESOLVING ****

Commented [pb176R174]: **** STILL NEEDS RESOLVING ****

Additional information:

Regarding circular dependencies, I would suggest either adding a qualifier to the end of the sentence, as in

"... guaranteed to be free from circular waits for resources, at least on a monoprocessor"

or talk to somebody who knows more than I do (perhaps Pat or Yannick) about whether the claim of freedom from deadlock is also true if there are multiple processors involved.

Steve Michell's comments seem to indicate that there could even be a problem in a monoprocessor environment. I was thinking that
the Ceiling Priority Protocol stuff prevented that, but perhaps he is right. I'll defer to others.

Commented [pb177R174]: Resolved (no change): deadlock in SPARK is prevented by checking respect of the
priority ceiling protocol, so for a PO subprogram to call another PO
procedure/function (but not entry), it needs to go to a higher value of
priority, so it's one-way only.

Commented [pb178R174]:

Commented [SM179]: I disagree with this statement. It is certainly true within a single protected object, but Ravenscar PO's are trivial, and most task interaction protocols will involve more than one PO and programmed sequences of calls to protected subprograms or protected entries in more than one PO. Hence, improperly constructed call sequences could result in deadlock. In general one would need a model checker to determine the correctness of the algorithm.

Commented [RCC180]: Added this ref to bibliography

Commented [pb181]: Stephen Baird: Is it worth mentioning here that another form of constant modification
that SPARK prevents is passing a variable by reference as an in-mode parameter and then, one way or another, modifying the variable?

Commented [pb182R181]: Fixed

In addition, SPARK will not permit the modification of a variable that has been passed into a subprogram by reference as “in-mode” parameter.

Bibliography

[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.

[5] Concurrent and Real-Time Programming In Ada. Alan Burns and Andy Wellings. Cambridge University Press, 2007. ISBN 978-0521866972.

Commented [RCC183]: Added Burns/Wellings book and removed all old entries

Index

Commented [RCC184]: Index entries need work

Page 44: [1] Commented [pb110] paul butcher 09-Sep-21 5:08:00 AM

Stephen Baird: When we talk about whether a vulnerability applies to SPARK, are we
talking about 100% SPARK with all proof obligations discharged? If
so, then why are we saying here that this vulnerability applies to SPARK?

I think the second paragraph is trying to say that if we are talking about 100% SPARK with all proof obligations discharged, then this vulnerability does not apply. No vulnerability is introduced if a user suppresses a check
that is statically known to always succeed. The first paragraph says that the vulnerability does apply in other cases.

And that is the order in which they should be presented.

In the Ada annex, the notion of unsafe programming was introduced; this includes things like Unchecked_Conversion. It is explicitly stated "up front" that properties such as strong typing can be circumvented through such techniques. This approach eliminates the need to then mention unsafe programming in the context of each specific vulnerability, although it is fine to still talk about unsafe programming in cases where that seems useful.

And when unsafe programming is discussed, it is as a secondary case. "Property Foo holds, except in some cases involving
unsafe programming", as opposed to "Property Foo does not hold, but violations can only occur as a result of unsafe programming".

That is how this vulnerability should be described for SPARK, if we want to mention the unsafe case at all.

Better still is the approach taken in 6.55, which states
"Bounded errors are entirely prevented by mandatory static verification."
and does not mention at all what happens in the case of undischarged proof obligations or other forms of unsafe programming.

Page 44: [2] Commented [pb112R110] paul butcher 23-Oct-21 1:25:00 AM

If we are talking about 100% SPARK with all proof obligations satisfied and no use of pragma Assume or any other such cheats, then the vulnerability does not exist in SPARK. As noted in the 3rd paragraph, it is ok to suppress
a check if you have shown statically that the check will never fail.

Resolved with the removal of “The vulnerability exists in SPARK since”