ISO/IEC JTC 1/SC 22 N 0000

Date: 2015-06-23

ISO/IEC TR 24772-5

Edition 1

ISO/IEC JTC 1/SC 22/WG 23

Secretariat: ANSI

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

Document type: International standard

Document subtype: if applicable

Document stage: (10) development stage

Document language: E

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

Warning

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

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

**Copyright notice**

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

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

*ISO copyright office*

*Case postale 56, CH-1211 Geneva 20*

*Tel. + 41 22 749 01 11*

*Fax + 41 22 749 09 47*

*E-mail copyright@iso.org*

*Web www.iso.org*

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

Violators may be prosecuted.

Contents Page

# Foreword

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

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

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

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

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

ISO/IEC TR 24772-8, was prepared by Joint Technical Committee ISO/IEC JTC 1, *Information technology*, Subcommittee SC 22, *Programming languages, their environments and system software interfaces*.

# Introduction

This Technical Report provides guidance for the programming language Ada 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 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 technical can also be used in comparison with companion technical reports and with the language-independent report, TR 24772-1, to select a programming language that provides the appropriate level of confidence that anticipated problems can be avoided.

This technical report part is intended to be used with TR 24772-1, which discusses programming language vulnerabilities in a language independent fashion.

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

**Information Technology — Programming Languages — Guidance to avoiding vulnerabilities in programming languages through language selection and use** **– Vulnerability descriptions for the programming language Spark**

# 1. Scope

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

Vulnerabilities described in this technical report document the way that each vulnerability described in the language-independent writeup (in Tr 24772-1) is manifested in Spark.

# 2. Normative references

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

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

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

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

[SPARK Language Definition](http://www.altran-praxis.com/sparkTechnicalReferences.aspx): “SPARK95: The SPADE Ada Kernel (Including RavenSPARK)” Latest version always available from [www.altran-praxis.com](http://www.altran-praxis.com).

[ISO/IEC TR 15942:2000](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=29575), Guidance for the Use of Ada in High Integrity Systems.

[ISO/IEC TR 24718:2005](http://www.iso.org/iso/iso_catalogue/catalogue_tc/catalogue_detail.htm?csnumber=38828), Guide for the use of the Ada Ravenscar Profile in high integrity systems.

ISO IEC ???? 754-2008, Binary Floating Point Arithmetic, IEEE, 2008.

ISO IEC ???? 854-1987, Radix-Independent Floating-Point Arithmetic, IEEE, 1987

# 3. Terms and definitions, symbols and conventions

## 3.1 Terms and definitions

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

The precise statement of the following definitions can be found in the Fortran standard.

***argument association***: association between an effective argument and a dummy argument

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

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

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

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

***image***: one of a mutually cooperating set of instances of a Fortran pro­gram; each has its own execution state and set of data objects

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

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

***module***: a separate scope that contains definitions that can be accessed from other scopes

***obsolescent feature***: a feature that is not recommended because better methods exist in the current standard

***processor***: combination of computing system and mechanism by which programs are transformed for use on that computing system

***processor dependent***: not completely specified in the Fortran standard, having one of a set of methods and semantics determined by the processor

***pure procedure***: a procedure subject to constraints such that its execution has no side effects

***type***: named category of data characterized by a set of values, a syntax for denoting these values, and a set of operations that interpret and manipulate the values

## 4 Language concepts

The SPARK language is a contractualized 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 TR.

Many terms and concepts applicable to Ada also apply to SPARK. See *4 Language concepts*.

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

**Soundness**

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

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

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

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

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

**SPARK Processor**

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

In SPARK, certain forms of analysis are said to be **mandatory** – they are required to be implemented and programs must pass these checks to be valid SPARK. Examples of mandatory analyses are the enforcement of the SPARK language subset, static semantic analysis (e.g. enhanced type checking) and information flow analysis [IFA].

Some analyses are said to be **optional** – a user may choose to enable these additional analyses at their discretion. The most notable example of an optional analysis in SPARK is the generation of verification conditions and their proof using a theorem proving tool. Optional analyses may provide greater depth of analysis, protection from additional vulnerabilities, and so on, at the cost of greater analysis time and effort.

**Failure modes for static analysis**

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

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

## 5 General guidance for Spark

*[ See Template] [Thoughts welcomed as to what could be provided here. Possibly an opportunity for the language community to address issues that do not correlate to the guidance of section 6. For languages that provide non-mandatory tools, how those tools can be used to provide effective mitigation of vulnerabilities described in the following sections]*

**6 Specific Guidance for Spark**

## 6.1 General

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

## 6.2 Type System [IHN]

SPARK mitigates this vulnerability.

SPARK’s type system is a simplification of Ada’s type system. Both Explicit and Implicit conversions are permitted in SPARK, as is instantiation and use of Unchecked\_Conversion [SB 1.3].

A design goal of SPARK is the provision of *static type safety,* meaning that programs can be shown to be free from all run-time type failures using entirely static analysis. If this optional analysis is achieved, a SPARK program should never raise an exception at run-time.

## 6.3 Bit Representation [STR]

SPARK mitigates this vulnerability.

* SPARK is designed to offer a semantics which is independent of the underlying representation chosen by a compiler for a particular target machine. Representation clauses are permitted, but these do not affect the semantics as seen by a static analysis tool [SB 1.3].

## 6.4 Floating-point Arithmetic [PLF]

### 6.4.1 Applicability to language

Spark specifies adherence to the IEEE Floating Point Standards (IEEE-754-2008, IEEE-854-1987).

The vulnerability in Spark is as described in Section 6.4.2.

### 6.4.2 Guidance to language users

* Rather than using predefined types, such as Float and Long\_Float, whose precision may vary according to the target system, declare floating-point types that specify the required precision (for example, digits 10). Additionally, specifying ranges of a floating point type enables constraint checks which prevents the propagation of infinities and NaNs.
* Avoid comparing floating-point values for equality. Instead, use comparisons that account for the approximate results of computations. Consult a numeric analyst when appropriate.
* Make use of static arithmetic expressions and static constant declarations when possible, since static expressions in Ada are computed at compile time with exact precision.
* Use Ada's standardized numeric libraries (for example, Generic\_Elementary\_Functions) for common mathematical operations (trigonometric operations, logarithms, and others).
* Use an Ada implementation that supports Annex G (Numerics) of the Ada standard, and employ the "strict mode" of that Annex in cases where additional accuracy requirements must be met by floating-point arithmetic and the operations of predefined numerics packages, as defined and guaranteed by the Annex.
* Avoid direct manipulation of bit fields of floating-point values, since such operations are generally target-specific and error-prone. Instead, make use of Ada's predefined floating-point attributes (such as 'Exponent).

In cases where absolute precision is needed, consider replacement of floating-point types and operations with fixed-point types and operations.

## 6.5 Enumerator Issues [CCB]

### 6.5.1 Applicability to language

Enumeration representation specification may be used to specify non-default representations of an enumeration type, for example when interfacing with external systems. All of the values in the enumeration type must be defined in the enumeration representation specification. The numeric values of the representation must preserve the original order. For example:

**type** IO\_Types **is** (Null\_Op, Open, Close, Read, Write, Sync);

**for** IO\_Types **use** (Null\_Op => 0, Open => 1, Close => 2,

Read => 4, Write => 8, Sync => 16 );

An array may be indexed by such a type. Ada does not prescribe the implementation model for arrays indexed by an enumeration type with non-contiguous values. Two options exist: Either the array is represented “with holes” and indexed by the values of the enumeration type, or the array is represented contiguously and indexed by the position of the enumeration value rather than the value itself. In the former case, the vulnerability described in 6.6 exists only if unsafe programming is applied to access the array or its components outside the protection of the type system. Within the type system, the semantics are well defined and safe. The vulnerability of unexpected but well-defined program behaviour upon extending an enumeration type exist in Ada. In particular, subranges or **others** choices in aggregates and case statements are susceptible to unintentionally capturing newly added enumeration values.

### 6.5.2 Guidance to language users

* For **case** statements and aggregates, do not use the **others** choice.
* For **case** statements and aggregates, mistrust subranges as choices after enumeration literals have been added anywhere but the beginning or the end of the enumeration type definition.

## 6.6 Numeric Conversion Errors [FLC]

SPARK prevents this vulnerability.

SPARK is designed to be amenable to static verification of the absence of predefined exceptions, and in particular all cases covered by this vulnerability [SB 11]. All numeric conversions (both explicit and implicit) give rise to a verification condition that must be discharged, typically using an automated theorem-prover.

## 6.7 String Termination [CJM]

With the exception of unsafe programming (see *4 Language* concepts), this vulnerability is not applicable to Spark as strings in Spark are not delimited by a termination character. Spark programs that interface to languages that use null-terminated strings and manipulate such strings directly should apply the vulnerability mitigations recommended for that language.

## 6.8 Buffer Boundary Violation (Buffer Overflow) [HCB]

SPARK prevents this vulnerability.

SPARK is designed to permit static analysis for all boundary violations, through techniques such as theorem proving or abstract interpretation [SB 11].

SPARK programs that have been subject to this level 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.

## 6.9 Unchecked Array Indexing [XYZ]

This vulnerability is absent in Spark. See *Error! Reference source not found.*

## 6.10 Unchecked Array Copying [XYW]

This vulnerability is absent in Spark since array assignments in SPARK are only permitted between objects that have statically matching bounds. Hence all violations are detected at compile time.

## 6.11 Pointer Type Conversions [HFC]

This vulnerability is absent in Spark since Spark forbids the declaration of access (pointer) types. [SB 1.3, SLRM 3.10]..

## 6.12 Pointer Arithmetic [RVG]

This vulnerability is absent in Spark since Spark forbids the declaration of access (pointer) types. [SB 1.3, SLRM 3.10].

## 6.13 Null Pointer Dereference [XYH]

This vulnerability is absent in Spark since Spark forbids the declaration of access (pointer) types. [SB 1.3, SLRM 3.10].

## 6.14 Dangling Reference to Heap [XYK]

This vulnerability is absent in Spark since Spark forbids the declaration of access (pointer) types. [SB 1.3, SLRM 3.10].

## 6.15 Arithmetic Wrap-around Error [FIF]

See *TR 24772-2 6.15 Arithmetic Wrap-around Error [FIF]*. In addition, SPARK mitigates this vulnerability through static analysis to show that a signed integer expression can never overflow at run-time [SB 11].

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

SPARK is identical to Ada with respect to this vulnerability and is mitigation. See TR 24772-2 *6.16 Using Shift Operations for Multiplication and Division [PIK]*.

## 6.17 Choice of Clear Names [NAI]

* SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *TR 24772-2 6.17 Choice of Clear Names [NAI]*.

## 6.18 Dead store [WXQ]

This vulnerability is not applicable to SPARK since Spark provides through mandatory static information flow analysis [IFA], which detects dead stores. Additionally, SPARK requires variables that are used for output to the environment, or for communication between tasks to be specifically identified. [IFA] 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]

SPARK mitigates this vulnerability.

As in *6.19 Unused Variable [YZS]*. Also, SPARK is designed to permit sound static analysis of the following cases [IFA]:

* Variables which are declared but not used at all.
* Variables which are assigned to, but the resulting value is not used in any way that affects an output of the enclosing subprogram. This is called an “ineffective assignment” in SPARK.

## 6.20 Identifier Name Reuse [YOW]

SPARK prevents this vulnerability.

* This vulnerability is prevented through language rules enforced by static analysis. SPARK does not permit names in local scopes to redeclare and hide names that are already visible in outer scopes [SLRM 6.1].

## 6.21 Namespace Issues [BJL]

* SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *6.21 Namespace Issues [BJL]*.

## 6.22 Initialization of Variables [LAV]

## This vulnerability is not applicable Spark since Spark provides mandatory static information flow analysis [IFA].

## 6.23 Operator Precedence/Order of Evaluation [JCW]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *TR 24772-2 6.23 Operator Precedence/Order of Evaluation [JCW]*.

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

This vulnerability is not applicable Spark since SPARK does not permit functions to have side-effects, so all expressions are side-effect free. Static analysis of run-time errors also ensures that expressions evaluate without raising exceptions. Therefore, expressions are neutral to evaluation order and this vulnerability does not occur in SPARK [SLRM 6.1]

## 6.25 Likely Incorrect Expression [KOA]

SPARK is identical to Ada with respect to this vulnerability and its mitigation (see *6.25 Likely Incorrect Expression [KOA]*) although many cases of “likely incorrect” expressions in Ada are forbidden in SPARK.

## 6.26 Dead and Deactivated Code [XYQ]

SPARK mitigates this vulnerability.

In addition to the advice of *TR 24772-2 6.26 Dead and Deactivated Code [XYQ]*, SPARK is amenable to optional static analysis of dead paths. A dead path cannot be executed in that the combination of conditions for its execution are logically equivalent to *false.* Such cases can be statically detected by theorem proving in SPARK

## 6.27 Switch Statements and Static Analysis [CLL]

As in *TR 24772-2 6.27 Switch Statements and Static Analysis [CLL]*, this vulnerability is prevented by SPARK. The vulnerability relating to an uninitialized variable and the “when others” clause in a case statement is also prevented – see *Error! Reference source not found.*..

## 6.28 Demarcation of Control Flow [EOJ]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *TR 24772-2 6.28 Demarcation of Control Flow [EOJ]*

## 6.29 Loop Control Variables [TEX]

SPARK prevents this vulnerability in the same way as Ada. See *6.29 Loop Control Variables [TEX]*

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

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *TR 24772-2 6.30 Off-by-one Error [XZH]*. Additionally, 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 detected by static analysis in SPARK [SB 11].

## 6.31 Structured Programming [EWD]

SPARK mitigates this vulnerability.

Several of the vulnerabilities in this category that affect Ada are entirely eliminated by SPARK. In particular: the use of the goto statement is prohibited in SPARK [SLRM 5.8], loop exit statements only apply to the most closely enclosing loop (so “multi-level loop exits” are not permitted) [SLRM 5.7], and all subprograms have a single entry and a single exit point [SLRM 6]. Finally, functions in SPARK must have exactly one return statement which must be the final statement in the function body [SLRM 6].

## 6.32 Passing Parameters and Return Values [CSJ]

SPARK mitigates this vulnerability by not providing pointers and by prohibiting aliasing.

SPARK goes further than Ada with regard to this vulnerability. Specifically;

* SPARK forbids all aliasing of parameters and name [SLRM 6]
* SPARK is designed to offer consistent semantics regardless of the parameter passing mechanism employed by a particular compiler. Thus this implementation-dependent behaviour of Ada is eliminated from SPARK.

Both of these properties can be checked by static analysis.

## 6.33 Dangling References to Stack Frames [DCM]

This vulnerability is not applicable Spark since SPARK forbids the use of the ‘Address attribute to read the address of an object [SLRM 4.1] and sincethe ‘Access attribute and all access types are forbidden..

## 6.34 Subprogram Signature Mismatch [OTR]

### 6.34.1 Applicability to language

SPARK mitigates this vulnerability.

Default values for subprogram are not permitted in SPARK [SLRM 6], so this case cannot occur. SPARK does permit calling modules written in other languages so, as in *6.34 Subprogram Signature Mismatch [OTR]*, additional steps are required to verify the number and type-correctness of such parameters.

SPARK also allows a subprogram body to be written in full-blown Ada (not SPARK). In this case, the subprogram body is said to be “hidden”, and no static analysis is performed by a SPARK Processor. For such hidden bodies, some alternative means of verification must be employed, and the advice of C.36 should be applied.

## 6.35 Recursion [GDL]

### 6.35.1 Applicability to language

This vulnerability is not applicable since SPARK does not permit recursion. [SLRM clause 6]

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

SPARK mitigates this vulnerability.

In SPARK, the normal approach is to use static analysis to prove that predefined exceptions cannot be raised. User-defined exceptions are not permitted.

As recommended in *6.36.2 Guidance to language users*, it may be appropriate to retain a single “top-level” exception handler for each task as an additional defense.

The vulnerability relating to an ignored error status is prevented by SPARK through static information flow analysis [IFA].

## 6.37 Fault Tolerance and Failure Strategies [REW]

### 6.37.1 Applicability to language

SPARK mitigates this vulnerability.

SPARK permits a limited subset of Ada’s tasking facilities known as the “Ravenscar Profile” [SLRM 9]. There is no nesting of tasks in SPARK, and all tasks are required to have a top-level loop which has no exit statements, so this vulnerability does not apply in SPARK.

SPARK is also amenable to static analysis for the absence of predefined exceptions [SB 11], thus mitigating the case where a task terminates prematurely (and silently) owing to an unhandled predefined exception.

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

### 6.38.1 Applicability to language

SPARK mitigates this vulnerability.

SPARK permits the instantiation and use of Unchecked\_Conversion as in Ada. The result of a call to Unchecked\_Conversion is not assumed to be valid, so static verification tools can then insist on re-validation of the result before further analysis can succeed [SB 11].

At the time of writing, SPARK does not permit discriminated records, so vulnerabilities relating to discriminated records and unchecked unions are prevented.

## 6.39 Memory Leak [XYL]

This vulnerability is absent in Spark since SPARK does not permit the use of access types, storage pools, or allocators [SLRM 3]. In SPARK, all objects have a fixed size in memory, so the language is also amenable to static analysis of worst-case memory usage.

## 6.40 Templates and Generics [SYM]

At the time of writing, SPARK does not permit the use of generics units, so this vulnerability is currently prevented. In future, the SPARK language may be extended to permit generic units, in which case section C.42 [SYM] applies.

## 6.41 Inheritance [RIP]

SPARK mitigates this vulnerability.

SPARK permits only a subset of Ada’s inheritance facilities to be used. Multiple inheritance, class-wide operations and dynamic dispatching are not permitted, so all vulnerabilities relating to these language features do not apply to SPARK [SLRM 3.8].

SPARK is also designed to be amenable to static verification of the Liskov Substitution Principle [LSP].

## 6.42 Extra Intrinsics [LRM]

SPARK prevents this vulnerability in the same way as Ada. See *6.42 Extra Intrinsics [LRM]*.

## 6.43 Argument Passing to Library Functions [TRJ]

SPARK mitigates this vulnerability by providing for preconditions to be checked statically by a theorem-prover.

## 6.44 Inter-language Calling [DJS]

SPARK mitigates this vulnerability by providing for preconditions to be checked statically by a theorem-prover.

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

SPARK prevents this vulnerability in the same way as Ada. See *6.45 Dynamically-linked Code and Self-modifying Code [NYY]*.

## 6.46 Library Signature [NSQ]

SPARK prevents this vulnerability in the same way as Ada. See *6.46 Library Signature [NSQ]*.

## 6.48 Unanticipated Exceptions from Library Routines [HJW]

SPARK prevents this vulnerability in the same way as Ada. See *6.48 Unanticipated Exceptions from Library Routines [HJW]*. SPARK does permit the use of exception handlers, so these may be used to catch unexpected exceptions from library routines.

## 6.48 Pre-Processor Directives [NMP]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *6.48 Pre-Processor Directives [NMP]*.

## 6.49 Suppression of Language-defined Run-time Checking [MXB]

The Fortran standard has many requirements that cannot be statically checked. While many processors provide options for run-time checking, the standard does not require that any such checks be provided.

## 6.50 Provision of Inherently Unsafe Operations [SKL]

As in Ada, SPARK allows the use of Unchecked\_Conversion, so the advice of *6.50 Provision of Inherently Unsafe Operations [SKL]* applies here.

SPARK provides a provision for “hidden” bodies – units not written in SPARK at all that are ignored by a SPARK Processor. These units are assumed to be written in Ada, so for these units, the advice of the entire Ada Annex should be applied.

## 6.51 Obscure Language Features [BRS]

SPARK mitigates this vulnerability.

The design of the SPARK subset avoids many language features that might be said to be “obscure” or “hard to understand”, such as controlled types, unrestricted tasking, anonymous access types and so on.

SPARK goes further, though, in aiming for a completely *unambiguous* semantics, removing all erroneous and implementation-dependent features from the language. This means that a SPARK program should have a single meaning to programmers, reviewers, maintainers and all compilers.

SPARK also bans the aliasing, overloading, and redeclaration of names, so that one entity only ever has one name and one name can denote at most one entity, further reducing the risk of mis-understanding or mis-interpretation of a program by a person, compiler or other tools.

## 6.52 Unspecified Behaviour [BQF]

This vulnerability is not applicable to Spark since there are no unspecified behaviours in Spark.

## 6.53 Undefined Behaviour [EWF]

SPARK prevents this vulnerability through subsetting and static analysis. The language is designed to exhibit no undefined behaviours.

## 6.54 Implementation-Defined Behaviour [FAB]

SPARK mitigates this vulnerability.

SPARK allows a number of implementation-defined features as in Ada. These include:

* The range of predefined integer types.
* The range and precision of predefined floating-point types.
* The range of System.Any\_Priority and its subtypes.
* The value of constants such as System.Max\_Int, System.Min\_Int and so on.
* The selection of T’Base for a user-defined integer or floating-point type T.
* The rounding mode of floating-point types.

In the first four cases, static analysis tools can be configured to “know” the appropriate values [SB 9.6]. Care must be taken to ensure that these values are correct for the intended implementation. In the fifth case, SPARK defines a contract to indicate the choice of base-type, which can be checked by a pragma Assert. In the final case, additional static analysis of numerical precision must be performed by the user to ensure the correctness of floating-point algorithms.

## 6.55 Deprecated Language Features [MEM]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See *6.55 Deprecated Language Features [MEM]*

## 6.56 Concurrency – Activation [CGA]

TBD

## 6.56.1 Applicability to language

### 6.56.2 Guidance to language users

## 6.57 Concurrency – Directed termination [CGT]

## TBD

## 6.57.1 Applicability to language

### 6.57.2 Guidance to language users

## 6.58 Concurrent Data Access [CGX]

TBD

## 6.58.1 Applicability to language

### 6.58.2 Guidance to language users

## 6.59 Concurrency – Premature Termination [CGS]

## TBD

## 6.59.1 Applicability to language

### 6.59.2 Guidance to language users

## 6.60 Protocol Lock Errors [CGM]

TBD

## 6.60.1 Applicability to language

### 6.60.2 Guidance to language users

## 6.61 Uncontrolled Format String [SHL]

TBD

## 7 Language specific vulnerabilities for Fortran

### 8 Implications for standardization

Same as Ada.

# Bibliography

[SB] “High Integrity Software: The SPARK Approach to Safety and Security.” John Barnes. Addison-Wesley, 2003. ISBN 0-321-13616-0.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

1. Lecture Notes on Computer Science 5020, “Ada 2012 Rationale: The Language, the Standard Libraries,” John Barnes, Springer, 2012. ???????

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[46] Lundqvist, K and Asplund, L., “*A Formal Model of a Run-Time Kernel for Ravenscar*”, The 6th International Conference on Real-Time Computing Systems and Applications – RTCSA 1999

# Index

Ada, 13, 59, 63, 73, 76

AMV – Type-breaking Reinterpretation of Data, 72

*API*

Application Programming Interface, 16

APL, 48

Apple

OS X, 120

*application vulnerabilities*, 9

Application Vulnerabilities

Adherence to Least Privilege [XYN], 113

Authentication Logic Error [XZO], 135

Cross-site Scripting [XYT], 125

Discrepancy Information Leak [XZL], 129

Distinguished Values in Data Types [KLK], 112

Download of Code Without Integrity Check [DLB], 137

Executing or Loading Untrusted Code [XYS], 116

Hard-coded Password [XYP], 136

Improper Restriction of Excessive Authentication Attempts [WPL], 140

Improperly Verified Signature [XZR], 128

Inclusion of Functionality from Untrusted Control Sphere [DHU], 139

Incorrect Authorization [BJE], 138

Injection [RST], 122

Insufficiently Protected Credentials [XYM], 133

Memory Locking [XZX], 117

Missing or Inconsistent Access Control [XZN], 134

Missing Required Cryptographic Step [XZS], 133

Path Traversal [EWR], 130

Privilege Sandbox Issues [XYO], 114

Resource Exhaustion [XZP], 118

Resource Names [HTS], 120

Sensitive Information Uncleared Before Use [XZK], 130

Unquoted Search Path or Element [XZQ], 127

Unrestricted File Upload [CBF], 119

Unspecified Functionality [BVQ], 111

URL Redirection to Untrusted Site ('Open Redirect') [PYQ], 140

Use of a One-Way Hash without a Salt [MVX], 141

applicationvulnerability, 5

Ariane 5, 21

bitwise operators, 48

BJE – Incorrect Authorization, 138

BJL – Namespace Issues, 43

*black-list*, 120, 124

BQF – Unspecified Behaviour, 92, 94, 95

break, 60

BRS – Obscure Language Features, 91

buffer boundary violation, 23

buffer overflow, 23, 26

buffer underwrite, 23

BVQ – Unspecified Functionality, 111

C, 22, 48, 50, 51, 58, 60, 63, 73

C++, 48, 51, 58, 63, 73, 76, 86

C11, 192

*call by copy*, 61

*call by name*, 61

*call by reference*, 61

*call by result*, 61

*call by value*, 61

*call by value-result*, 61

CBF – Unrestricted File Upload, 119

CCB – Enumerator Issues, 18

CGA – Concurrency – Activation, 98

CGM – Protocol Lock Errors, 105

CGS – Concurrency – Premature Termination, 103

CGT - Concurrency – Directed termination, 100

CGX – Concurrent Data Access, 101

CGY – Inadequately Secure Communication of Shared Resources, 107

CJM – String Termination, 22

CLL – Switch Statements and Static Analysis, 54

concurrency, 2

continue, 60

cryptologic, 71, 128

CSJ – Passing Parameters and Return Values, 61, 82

dangling reference, 31

DCM – Dangling References to Stack Frames, 63

Deactivated code, 53

Dead code, 53

*deadlock*, 106

DHU – Inclusion of Functionality from Untrusted Control Sphere, 139

Diffie-Hellman-style, 136

digital signature, 84

DJS – Inter-language Calling, 81

DLB – Download of Code Without Integrity Check, 137

*DoS*

Denial of Service, 118

dynamically linked, 83

EFS – Use of unchecked data from an uncontrolled or tainted source, 109

encryption, 128, 133

endian

big, 15

little, 15

endianness, 14

Enumerations, 18

EOJ – Demarcation of Control Flow, 56

EWD – Structured Programming, 60

*EWF – Undefined Behaviour*, 92, 94, 95

*EWR – Path Traversal*, 124, 130

exception handler, 86

*FAB – Implementation-defined Behaviour*, 92, 94, 95

FIF – Arithmetic Wrap-around Error, 34, 35

FLC – Numeric Conversion Errors, 20

Fortran, 73

GDL – Recursion, 67

generics, 76

GIF, 120

goto, 60

HCB – Buffer Boundary Violation (Buffer Overflow), 23, 82

HFC – Pointer Casting and Pointer Type Changes, 28

HJW – Unanticipated Exceptions from Library Routines, 86

*HTML*

Hyper Text Markup Language, 124

HTS – Resource Names, 120

*HTTP*

Hypertext Transfer Protocol, 127

IEC 60559, 16

IEEE 754, 16

IHN –Type System, 12

inheritance, 78

IP address, 119

Java, 18, 50, 52, 76

JavaScript, 125, 126, 127

JCW – Operator Precedence/Order of Evaluation, 47

KLK – Distinguished Values in Data Types, 112

KOA – Likely Incorrect Expression, 50

*language vulnerabilities*, 9

*Language Vulnerabilities*

Argument Passing to Library Functions [TRJ], 80

Arithmetic Wrap-around Error [FIF], 34

Bit Representations [STR], 14

Buffer Boundary Violation (Buffer Overflow) [HCB], 23

Choice of Clear Names [NAI], 37

Concurrency – Activation [CGA], 98

Concurrency – Directed termination [CGT], 100

Concurrency – Premature Termination [CGS], 103

Concurrent Data Access [CGX], 101

Dangling Reference to Heap [XYK], 31

Dangling References to Stack Frames [DCM], 63

Dead and Deactivated Code [XYQ], 52

Dead Store [WXQ], 39

Demarcation of Control Flow [EOJ], 56

Deprecated Language Features [MEM], 97

Dynamically-linked Code and Self-modifying Code [NYY], 83

Enumerator Issues [CCB], 18

Extra Intrinsics [LRM], 79

*Floating-point Arithmetic [PLF]*, xvii, 16

Identifier Name Reuse [YOW], 41

Ignored Error Status and Unhandled Exceptions [OYB], 68

Implementation-defined Behaviour [FAB], 95

Inadequately Secure Communication of Shared Resources [CGY], 107

Inheritance [RIP], 78

Initialization of Variables [LAV], 45

Inter-language Calling [DJS], 81

Library Signature [NSQ], 84

Likely Incorrect Expression [KOA], 50

Loop Control Variables [TEX], 57

Memory Leak [XYL], 74

Namespace Issues [BJL], 43

Null Pointer Dereference [XYH], 30

Numeric Conversion Errors [FLC], 20

Obscure Language Features [BRS], 91

Off-by-one Error [XZH], 58

Operator Precedence/Order of Evaluation [JCW], 47

Passing Parameters and Return Values [CSJ], 61, 82

Pointer Arithmetic [RVG], 29

Pointer Casting and Pointer Type Changes [HFC], 28

Pre-processor Directives [NMP], 87

Protocol Lock Errors [CGM], 105

Provision of Inherently Unsafe Operations [SKL], 90

Recursion [GDL], 67

Side-effects and Order of Evaluation [SAM], 49

Sign Extension Error [XZI], 36

String Termination [CJM], 22

Structured Programming [EWD], 60

Subprogram Signature Mismatch [OTR], 65

Suppression of Language-defined Run-time Checking [MXB], 89

Switch Statements and Static Analysis [CLL], 54

Templates and Generics [SYM], 76

Termination Strategy [REU], 70

Type System [IHN], 12

Type-breaking Reinterpretation of Data [AMV], 72

Unanticipated Exceptions from Library Routines [HJW], 86

Unchecked Array Copying [XYW], 27

Unchecked Array Indexing [XYZ], 25

Uncontrolled Fromat String [SHL], 110

Undefined Behaviour [EWF], 94

Unspecified Behaviour [BFQ], 92

Unused Variable [YZS], 40

Use of unchecked data from an uncontrolled or tainted source [EFS], 109

Using Shift Operations for Multiplication and Division [PIK], 35

language vulnerability, 5

LAV – Initialization of Variables, 45

LHS (left-hand side), 241

Linux, 120

*livelock*, 106

longjmp, 60

LRM – Extra Intrinsics, 79

MAC address, 119

macof, 118

MEM – Deprecated Language Features, 97

memory disclosure, 130

Microsoft

Win16, 121

Windows, 117

Windows XP, 120

*MIME*

Multipurpose Internet Mail Extensions, 124

MISRA C, 29

MISRA C++, 87

mlock(), 117

MVX – Use of a One-Way Hash without a Salt, 141

MXB – Suppression of Language-defined Run-time Checking, 89

NAI – Choice of Clear Names, 37

*name type equivalence*, 12

NMP – Pre-Processor Directives, 87

NSQ – Library Signature, 84

*NTFS*

New Technology File System, 120

NULL, 31, 58

NULL pointer, 31

null-pointer, 30

NYY – Dynamically-linked Code and Self-modifying Code, 83

OTR – Subprogram Signature Mismatch, 65, 82

OYB – Ignored Error Status and Unhandled Exceptions, 68, 163

Pascal, 82

PHP, 124

*PIK – Using Shift Operations for Multiplication and Division*, 34, 35, 197

*PLF – Floating-point Arithmetic*, xvii, 16

POSIX, 99

pragmas, 75, 96

predictableexecution, 4, 8

PYQ – URL Redirection to Untrusted Site ('Open Redirect'), 140

real numbers, 16

Real-Time Java, 105

resource exhaustion, 118

REU – Termination Strategy, 70

*RIP – Inheritance*, xvii, 78

rsize\_t, 22

RST – Injection, 109, 122

*runtime-constraint handler*, 191

RVG – Pointer Arithmetic, 29

safetyhazard, 4

safety-critical software, 5

SAM – Side-effects and Order of Evaluation, 49

securityvulnerability, 5

SeImpersonatePrivilege, 115

setjmp, 60

SHL – Uncontrolled Format String, 110

size\_t, 22

SKL – Provision of Inherently Unsafe Operations, 90

software quality, 4

*software vulnerabilities*, 9

*SQL*

Structured Query Language, 112

STR – Bit Representations, 14

strcpy, 23

strncpy, 23

*structure type equivalence*, 12

switch, 54

SYM – Templates and Generics, 76

symlink, 131

*tail-recursion*, 68

templates, 76, 77

TEX – Loop Control Variables, 57

**thread**, 2

TRJ – Argument Passing to Library Functions, 80

*type casts*, 20

*type coercion*, 20

*type safe*, 12

*type secure*, 12

*type system*, 12

UNC

Uniform Naming Convention, 131

Universal Naming Convention, 131

Unchecked\_Conversion, 73

UNIX, 83, 114, 120, 131

unspecified functionality, 111

*Unspecified functionality*, 111

*URI*

Uniform Resource Identifier, 127

URL

Uniform Resource Locator, 127

VirtualLock(), 117

*white-list*, 120, 124, 127

Windows, 99

WPL – Improper Restriction of Excessive Authentication Attempts, 140

WXQ – Dead Store, 39, 40, 41

XSS

Cross-site scripting, 125

XYH – Null Pointer Deference, 30

XYK – Dangling Reference to Heap, 31

XYL – Memory Leak, 74

*XYM – Insufficiently Protected Credentials*, 9, 133

XYN –Adherence to Least Privilege, 113

XYO – Privilege Sandbox Issues, 114

XYP – Hard-coded Password, 136

XYQ – Dead and Deactivated Code, 52

XYS – Executing or Loading Untrusted Code, 116

XYT – Cross-site Scripting, 125

XYW – Unchecked Array Copying, 27

XYZ – Unchecked Array Indexing, 25, 28

XZH – Off-by-one Error, 58

XZI – Sign Extension Error, 36

XZK – Senitive Information Uncleared Before Use, 130

XZL – Discrepancy Information Leak, 129

XZN – Missing or Inconsistent Access Control, 134

XZO – Authentication Logic Error, 135

XZP – Resource Exhaustion, 118

XZQ – Unquoted Search Path or Element, 127

XZR – Improperly Verified Signature, 128

XZS – Missing Required Cryptographic Step, 133

XZX – Memory Locking, 117

YOW – Identifier Name Reuse, 41, 44

*YZS – Unused Variable*, 39, 40