ISO/IEC JTC 1/SC 22/WG 23 N 0275

Draft language-specific annex for SPARK

Date	2 September 2010
Contributed by	SC 22/WG 9
Original file name	N502, SPARK_Annex Draft 1, 16 Aug 2010.doc
Notes	

1

Annex SPARK – Final Draft

SPARK.Specific information for vulnerabilities

Status and History
September 2009: First draft from SPARK team.
November 2009: Second draft following comments from HRG.
May 2010: Updates to be consistent with Ada Annex and new vulnerabilities in the parent TR.
June 2010: Updates following review comments from HRG.
July 2010: Submit to WG9.

SPARK.1 Identification of standards and associated documentation

See Ada.1, plus the references below. In the body of this annex, the following documents are referenced using the short abbreviation that introduces each document, optionally followed by a specific section number. For example "[SLRM 5.2]" refers to section 5.2 of the SPARK Language Definition.

[SLRM] SPARK Language Definition: "SPARK95: The SPADE Ada Kernel (Including RavenSPARK)" Latest version always available from www.altran-praxis.com.

[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.

SPARK.2 General terminology and 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.

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.

SPARK.3.BRS Obscure Language Features [BRS]

SPARK mitigates this vulnerability.

SPARK.3.BRS.1 Terminology and features

As in Ada.3.BRS.1.

SPARK.3.BRS.2 Description of vulnerability

As in Ada.3.BRS.2.

SPARK.3.BRS.3 Avoiding the vulnerability or mitigating its effects

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.

SPARK.3.BRS.4 Implications for standardization

None.

SPARK.3.BRS.5 Bibliography

None.

SPARK.3.BQF Unspecified Behaviour [BQF]

SPARK prevents this vulnerability.

SPARK.3.BQF.1 Terminology and features

As in Ada.3.BQF.1.

SPARK.3.BQF.2 Description of vulnerability

As in Ada.3.BQF.2.

SPARK.3.BQF.3 Avoiding the vulnerability or mitigating its effects

SPARK is designed to eliminate all unspecified language features and bounded errors, either by subsetting to make the offending language feature illegal in SPARK, or by ensuring that the language has neutral semantics with regard to an unspecified behaviour.

“Neutral semantics” means that the program has identical meaning regardless of the choice made by a compiler for a particular unspecified language feature.

For example:

- Unspecified behaviour as a result of parameter-passing mechanism is avoided through subsetting (no access types) and analysis to make sure that formal and global parameters do not overlap and create a potential for aliasing [SLRM 6.4].

- Dependence on evaluation order is prevented through analysis so that all expressions in SPARK are free of side-effects and potential run-time errors. Therefore, any evaluation order is allowed and the result of the evaluation is the same in all cases [SLRM 6.1].

- Bounded error as a result of uninitialized variables is prevented by application of static information flow analysis [IFA].

SPARK.3.BQF.4 Implications for standardization

None.

SPARK.3.BQF.5 Bibliography

None.

SPARK.3.EWF Undefined Behaviour [EWF]

SPARK prevents this vulnerability.

SPARK.3.EWF.1 Terminology and features

As in Ada.3.EWF.1.

SPARK.3.EWF.2 Description of vulnerability

As in Ada.3.EWF.2.

SPARK.3.EWF.3 Avoiding the vulnerability or mitigating its effects

SPARK prevents all erroneous behaviour, either through subsetting or static analysis [SB 1.3].

SPARK.3.EWF.4 Implications for standardization

None.

SPARK.3.EWF.5 Bibliography

None.

SPARK.3.FAB Implementation-Defined Behaviour [FAB]

SPARK mitigates this vulnerability.

SPARK.3.FAB.1 Terminology and features

As in Ada.3.FAB.1.

SPARK.3.FAB.2 Description of vulnerability

As in Ada.3.FAB.2.

SPARK.3.FAB.3 Avoiding the vulnerability or mitigating its effects

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.

SPARK.3.FAB.4 Implications for standardization

None.

SPARK.3.FAB.5 Bibliography

None.

SPARK.3.MEM Deprecated Language Features [MEM]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.MEM.

SPARK.3.NMP Pre-Processor Directives [NMP]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.NMP.

SPARK.3.NAI Choice of Clear Names [NAI]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.NAI.

SPARK.3.AJN Choice of Filenames and other External Identifiers [AJN]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.AJN.

SPARK.3.XYR Unused Variable [XYR]

SPARK mitigates this vulnerability.

SPARK.3.XYR.1 Terminology and features

As in Ada.3.XYR.1.

SPARK.3.XYR.2 Description of vulnerability

As in Ada.3.XYR.2.

SPARK.3.XYR.3 Avoiding the vulnerability or mitigating its effects

As in Ada.3.XYR.3. 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.

SPARK.3.XYR.4 Implications for standardization

None.

SPARK.3.XYR.5 Bibliography

None.

SPARK.3.YOW Identifier Name Reuse [YOW]

SPARK prevents this vulnerability.

SPARK.3.YOW.1 Terminology and features

As in Ada.3.YOW.1.

SPARK.3.YOW.2 Description of vulnerability

As in Ada.3.YOW.2.

SPARK.3.YOW.3 Avoiding the vulnerability or mitigating its effects

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].

SPARK.3.YOW.4 Implications for standardization

None.

SPARK.3.YOW.5 Bibliography

None.

SPARK.3.BKL Namespace Issues [BJL]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.BJL.

SPARK.3.IHN Type System [IHN]

SPARK mitigates this vulnerability.

SPARK.3.IHN.1 Terminology and features

SPARK’s type system is a simplification of that of Ada. 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.

SPARK.3.IHN.2 Description of vulnerability

As in Ada.3.IHN.2 for Unchecked_Conversion.

SPARK.3.IHN.3 Avoiding the vulnerability or mitigating its effects

Vulnerabilities relating to value conversions, exceptions, and assignments are mitigated by static analysis. Vulnerabilities relating to the use of Unchecked_Conversion are as in Ada.

SPARK.3.IHN.4 Implications for standardization

None.

SPARK.3.IHN.5 Bibliography

None.

SPARK.3.STR Bit Representation [STR]

SPARK mitigates this vulnerability.

SPARK.3.STR.1 Terminology and features

As in Ada.3.STR.1.

SPARK.3.STR.2 Description of 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].

SPARK.3.STR.3 Avoiding the vulnerability or mitigating its effects

As in Ada.3.STR.4.

SPARK.3.STR.4 Implications for standardization

None.

SPARK.3.STR.5 Bibliography

None.

SPARK.3.PLF Floating-point Arithmetic [PLF]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.PLF.

SPARK.3.CCB Enumerator Issues [CCB]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.CCB.

SPARK.3.FLC Numeric Conversion Errors [FLC]

SPARK prevents this vulnerability.

SPARK.3.FLC.1 Terminology and features

As in Ada.3.FLC.1.

SPARK.3.FLC.2 Description of vulnerability

As in Ada.3.FLC.2.

SPARK.3.FLC.3 Avoiding the vulnerability or mitigating its effects

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.

SPARK.3.FLC.4 Implications for standardization

None.

SPARK.3.FLC.5 Bibliography

None.

SPARK.3.CJM String Termination [CJM]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.CJM.

SPARK.3.XYX Boundary Beginning Violation [XYX]

SPARK prevents this vulnerability.

SPARK.3.XYX.1 Terminology and features

As in Ada.3.XYX.1.

SPARK.3.XYX.2 Description of vulnerability

As in Ada.3.XYX.2.

SPARK.3.XYX.3 Avoiding the vulnerability or mitigating its effects

SPARK is designed to permit static analysis for all such 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.

SPARK.3.XYX.4 Implications for standardization

None.

SPARK.3.XYX.5 Bibliography

None.

SPARK.3.XYZ Unchecked Array Indexing [XYZ]

SPARK prevents this vulnerability.

SPARK.3.XYZ.1 Terminology and features

As in Ada.3.XYZ.1.

SPARK.3.XYZ.2 Description of vulnerability

As in Ada.3.XYZ.2.

SPARK.3.XYZ.3 Avoiding the vulnerability or mitigating its effects

As per SPARK.3.XYX.3 – this vulnerability is eliminated in SPARK by static analysis using the same techniques.

SPARK.3.XYZ.4 Implications for standardization

None.

SPARK.3.XYZ.5 Bibliography

None.

SPARK.3.XYW Unchecked Array Copying [XYW]

SPARK prevents this vulnerability.

SPARK.3.XYW.1 Terminology and features

As in Ada.3.XYW.1.

SPARK.3.XYW.2 Description of vulnerability

As in Ada.3.XYW.2.

SPARK.3.XYW.3 Avoiding the vulnerability or mitigating its effects

Array assignments in SPARK are only permitted between objects that have statically matching bounds, so there is no possibility of an exception being raised [SB 5.5, SLRM 4.1.2]. Ada's “slicing” and “sliding” of arrays is not permitted in SPARK, so this vulnerability cannot occur.

SPARK.3.XYW.4 Implications for standardization

None.

SPARK.3.XYW.5 Bibliography

None.

SPARK.3.XZB Buffer Overflow [XZB]

SPARK prevents this vulnerability.

SPARK.3.XZB.1 Terminology and features

As in Ada.3.HCF.1.

SPARK.3.XZB.2 Description of vulnerability

As in Ada.3.XZB.2.

SPARK.3.XZB.3 Avoiding the vulnerability or mitigating its effects

As per SPARK.3.XYX.3 – this vulnerability is eliminated in SPARK by static analysis using the same techniques.

SPARK.3.XZB.4 Implications for standardization

None.

SPARK.3.XZB.5 Bibliography

None.

SPARK.3.HCF Pointer Casting and Pointer Type Changes [HCF]

SPARK prevents this vulnerability.

SPARK.3.HCF.1 Terminology and features

As in Ada.3.HCF.1.

SPARK.3.HCF.2 Description of vulnerability

As in Ada.3.HCF.2.

SPARK.3.HCF.3 Avoiding the vulnerability or mitigating its effects

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access (pointer) types [SB 1.3, SLRM 3.10].

SPARK.3.HCF.4 Implications for standardization

None.

SPARK.3.HCF.5 Bibliography

None.

SPARK.3.RVG Pointer Arithmetic [RVG]

SPARK prevents this vulnerability.

SPARK.3.RVG.1 Terminology and features

As in Ada.3.RVG.1.

SPARK.3.RVG.2 Description of vulnerability

As in Ada.3.RVG.2.

SPARK.3.RVG.3 Avoiding the vulnerability or mitigating its effects

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access (pointer) types [SLRM 3.10].

SPARK.3.RVG.4 Implications for standardization

None.

SPARK.3.RVG.5 Bibliography

None.

SPARK.3.XYH Null Pointer Dereference [XYH]

SPARK prevents this vulnerability.

SPARK.3.XYH.1 Terminology and features

As in Ada.3.XYH.1.

SPARK.3.XYH.2 Description of vulnerability

As in Ada.3.XYH.2.

SPARK.3.XYH.3 Avoiding the vulnerability or mitigating its effects

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access (pointer) types [SLRM 3.10].

SPARK.3.XYH.4 Implications for standardization

None.

SPARK.3.XYH.5 Bibliography

None.

SPARK.3.XYK Dangling Reference to Heap [XYK]

SPARK prevents this vulnerability.

SPARK.3.XYK.1 Terminology and features

As in Ada.3.XYK.1.

SPARK.3.XYK.2 Description of vulnerability

As in Ada.3.XYK.2.

SPARK.3.XYK.3 Avoiding the vulnerability or mitigating its effects

This vulnerability cannot occur in SPARK, since the SPARK subset forbids the declaration or use of access (pointer) types [SLRM 3.10].

SPARK.3.XYK.4 Implications for standardization

None.

SPARK.3.XYK.5 Bibliography

None.

SPARK.3.SYM 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 Ada.3.SYM applies.

SPARK.3.RIP Inheritance [RIP]

SPARK mitigates this vulnerability.

SPARK.3.RIP.1 Terminology and features

As in Ada.3.RIP.1.

SPARK.3.RIP.2 Description of vulnerability

As in Ada.3.RIP.1.

SPARK.3.RIP.3 Avoiding the vulnerability or mitigating its effects

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].

SPARK.3.RIP.4 Implications for standardization

None.

SPARK.3.RIP.5 Bibliography

None.

SPARK.3.LAV Initialization of Variables [LAV]

SPARK prevents this vulnerability.

SPARK.3.LAV.1 Terminology and features

As in Ada.3.LAV.1.

SPARK.3.LAV.2 Description of vulnerability

Ada in Ada.3.LAV.2.

SPARK.3.LAV.3 Avoiding the vulnerability or mitigating its effects

This vulnerability is entirely prevented by use of static information flow analysis [IFA].

SPARK.3.LAV.4 Implications for standardization

None.

SPARK.3.LAV.5 Bibliography

None.

SPARK.3.XYY Wrap-around Error [XYY]

See Ada.3.XYY. In addition, SPARK mitigates this vulnerability through static analysis to show that a signed integer expression can never overflow at run-time [SB 11].

SPARK.3.XZI Sign Extension Error [XZI]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.XZI.

SPARK.3.JCW Operator Precedence/Order of Evaluation [JCW]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.JCW.

SPARK.3.SAM Side-effect and Order of Evaluation [SAM]

SPARK prevents this vulnerability.

SPARK.3.SAM.1 Terminology and features

As in Ada.3.SAM.1.

SPARK.3.SAM.2 Description of vulnerability

As in Ada.3.SAM.2.

SPARK.3.SAM.3 Avoiding the vulnerability or mitigating its effects

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].

SPARK.3.SAM.4 Implications for standardization

None.

SPARK.3.SAM.5 Bibliography

None.

SPARK.3.KOA Likely Incorrect Expression [KOA]

SPARK is identical to Ada with respect to this vulnerability and its mitigation (see Ada.3.KOA) although many cases of “likely incorrect” expressions in Ada are forbidden in SPARK.

SPARK.3.XYQ Dead and Deactivated Code [XYQ]

SPARK mitigates this vulnerability.

SPARK.3.XYQ.1 Terminology and features

As in Ada.3.XYQ.1.

SPARK.3.XYQ.2 Description of vulnerability

As in Ada.3.XYQ.2.

SPARK.3.XYQ.3 Avoiding the vulnerability or mitigating its effects

In addition to the advice of Ada.3.XYQ.3, 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.

SPARK.3.XYQ.4 Implications for standardization

None.

SPARK.3.XYQ.5 Bibliography

None.

SPARK.3.CLL Switch Statements and Static Analysis [CLL]

As in Ada.3.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 SPARK.3.LAV.

SPARK.3.EOJ Demarcation of Control Flow [EOJ]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.EOJ.

SPARK.3.TEX Loop Control Variables [TEX]

SPARK prevents this vulnerability in the same way as Ada. See Ada.3.TEX.

SPARK.3.XZH Off-by-one Error [XZH]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.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].

SPARK.3.EWD Structured Programming [EWD]

SPARK mitigates this vulnerability.

SPARK.3.EWD.1 Terminology and features

As in Ada.3.EWD.1

SPARK.3.EWD.2 Description of vulnerability

As in Ada.3.EWD.2

SPARK.3.EWD.3 Avoiding the vulnerability or mitigating its effects

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 the final statement in the function body [SLRM 6].

SPARK.3.EWD.4 Implications for standardization

None.

SPARK.3.EWD.5 Bibliography

None.

SPARK.3.CSJ Passing Parameters and Return Values [CSJ]

SPARK mitigates this vulnerability.

SPARK.3.CSJ.1 Terminology and features

As in Ada.CSJ.1.

SPARK.3.CSJ.2 Description of vulnerability

As in Ada.CSJ.3.

SPARK.3.CSJ.3 Avoiding the vulnerability or mitigating its effects

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

- SPARK forbids all aliasing of parameters and names [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.

SPARK.3.CSJ.4 Implications for standardization

None.

SPARK.3.CSJ.5 Bibliography

None.

SPARK.3.DCM Dangling References to Stack Frames [DCM]

SPARK prevents this vulnerability.

SPARK.3.DCM.1 Terminology and features

As in Ada.3.DCM.1.

SPARK.3.DCM.2 Description of vulnerability

As in Ada.3.DCM.2.

SPARK.3.DCM.3 Avoiding the vulnerability or mitigating its effects

SPARK forbids the use of the 'Address attribute to read the address of an object [SLRM 4.1]. The 'Access attribute and all access types are also forbidden, so this vulnerability cannot occur.

SPARK.3.DCM.4 Implications for standardization

None.

SPARK.3.DCM.5 Bibliography

None.

SPARK.3.OTR Subprogram Signature Mismatch [OTR]

SPARK mitigates this vulnerability.

SPARK.3.OTR.1 Terminology and features

See Ada.3.OTR.1.

SPARK.3.OTR.2 Description of vulnerability

See Ada.3.OTR.2.

SPARK.3.OTR.3 Avoiding the vulnerability or mitigating its effects

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 Ada.3.OTR.3, 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 Annex Ada should be applied.

SPARK.3.OTR.4 Implications for standardization

None.

SPARK.3.OTR.5 Bibliography

None.

SPARK.3.GDL Recursion [GDL]

SPARK does not permit recursion, so this vulnerability is prevented [SLRM 6].

SPARK.3.NZN Returning Error Status [NZN]

SPARK is identical to Ada with respect to this vulnerability and its mitigation. See Ada.3.NZN.

SPARK.3.REU Termination Strategy [REU]

SPARK mitigates this vulnerability.

SPARK.3.REU.1 Terminology and features

As in Ada.3.REU.1.

SPARK.3.REU.2 Description of vulnerability

As in Ada.3.REU.2.

SPARK.3.REU.3 Avoiding the vulnerability or mitigating its effects

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.

SPARK.3.REU.4 Implications for standardization

None.

SPARK.3.REU.5 Bibliography

None.

SPARK.3.LRM Extra Intrinsics [LRM]

SPARK prevents this vulnerability in the same way as Ada. See Ada.3.LRM.

SPARK.3.AMV Type-breaking Reinterpretation of Data [AMV]

SPARK mitigates this vulnerability.

SPARK.3.AMV.1 Terminology and features

As in Ada.3.AMV.1.

SPARK.3.AMV.2 Description of vulnerability

As in Ada.3.AMV.2.

SPARK.3.AMV.3 Avoiding the vulnerability or mitigating its effects

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.

SPARK.3.AMV.4 Implications for standardization

None.

SPARK.3.AMV.5 Bibliography

None.

SPARK.3.XYL Memory Leak [XYL]

SPARK prevents this vulnerability.

SPARK.3.XYL.1 Terminology and features

As in Ada.3.XYL.1.

SPARK.3.XYL.2 Description of vulnerability

As in Ada.3.XYL.2.

SPARK.3.XYL.3 Avoiding the vulnerability or mitigating its effects

SPARK does not permit the use of access types, storage pools, or allocators, so this vulnerability cannot occur [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.

SPARK.3.XYL.4 Implications for standardization

None.

SPARK.3.XYL.5 Bibliography

None.

SPARK.3.TRJ Argument Passing to Library Functions [TRJ]

SPARK mitigates this vulnerability.

SPARK.3.TRJ.1 Terminology and features

See Ada.3.TRJ.1.

SPARK.3.TRJ.2 Description of vulnerability

See Ada.3.TRJ.2.

SPARK.3.TRJ.3 Avoiding the vulnerability or mitigating its effects

SPARK includes all of the mitigations of Ada with respect to this vulnerability, but goes further, allowing preconditions to be checked statically by a theorem-prover. The language in which such preconditions are expressed is also substantially more expressive than Ada's type system.

SPARK.3.TRJ.4 Implications for standardization

None.

SPARK.3.TRJ.5 Bibliography

None.

SPARK.3.NYY Dynamically-linked Code and Self-modifying Code [NYY]

SPARK prevents this vulnerability in the same way as Ada. See Ada.3.NYY.

SPARK.3.NSQ Library Signature [NSQ]

SPARK prevents this vulnerability in the same way as Ada. See Ada.3.NSQ.

SPARK.3.HJW Unanticipated Exceptions from Library Routines [HJW]

SPARK prevents this vulnerability in the same way as Ada. See Ada.3.HJW. SPARK does permit the use of exception handlers, so these may be used to catch unexpected exceptions from library routines.