*<<<<*

*This is my proposed set of changes to deal with the 2 remaining issues summarized by:*

1. *“recommending Pragma Restrictions(No\_Recursion) is a really bad idea.”*
2. *“Aliasing effects are possible after all in volatile functions with volatile actuals.”*

*In the process, I discovered that volatile actuals are in general another source of the vulnerability (of non-deterministic subprog outcome) and wrote it up accordingly.*

*It is quite possible that some SPARK rule in the context of volatiles prevents or alters the problem scenarios, e.g., by prescribing the parameter passing mode or restricting the types of volatiles, or … However, these rules have not been forthcoming in past exchanges. So, if they exist, they need to be cited in a changed write-up or discussed in a TelCon so we can phrase it online.*

*>>>>>*

### In 5.1.7

As a consequence of the above rules, SPARK avoids all aliasing effects on allocated objects in a program.

### 6.32 Passing parameters and return values [CSJ]

### 6.32.1 Applicability to language

The vulnerability as described in ISO/IEC 24772-1 subclause 6.32 is mitigated in SPARK, because, with the exception of some rare cases explained below the vulnerability is prevented by the following language concepts:

* 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 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 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 the semantics of a procedure call cannot include aliasing effects as described in ISO/IEC 24772-1 subclause 6.32.
* There are no checks required against aliasing among parameters and globals, since in (non-volatile) functions the assignments necessary to cause aliasing effects are disallowed in order to disable side-effects.
* SPARK requires static verification of information flow to verify that the value returned from a function call is never ignored.

The rare cases exposed to the vulnerability involve volatile parameters:

* In volatile functions, volatile actual parameters can be aliased to each other or to a global of the function. A concurrent external assignment to one of the volatile aliases causes an aliasing effect on any other aliased parameter if this parameter is passed by reference. because in this case, the parameter changes value, while for by-copy passing, it does not. As programmers are well aware that volative variables can have their value changed asynchronously, the code vulnerable to aliasing effects on volatile variables needs to be under close scrutiny already, which mitigates the vulnerability. It matters little whether the potential value change is by external assignment or by aliasing effect.
* For any volatile actual parameter (of any subprogram), an external assignment to the actual parameter concurrent to the execution of the subprogram body will have different effects on the later execution, depending on the parameter passing mechanism chosen by the compiler for the parameter.

### 6.32.2 Guidance to language users

* Avoid volatile actual parameters.
* If volatile actual parameters are present in a call on a volatile function, ensure that there is no aliasing within the set of actual parameters and globals of the function.
* If volatile actual parameters are present in a subprogram call, be cognizant of the parameter passing mechanism chosen by the compiler.

### 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 error status and unhandled exceptions [OYB].

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. The assertion aspect Subprogram\_Variant can be applied to recursive subprograms to specify a monotonously increasing or decreasing expression that assists in verifying the termination of the recursion.

The aspect Restrictions (No\_Recursion) does not enforce the absence of recursion; it merely renders the program erroneous if it executes any recursive call.

#### 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.
* Assist the termination proofs for recursive subprograms by means of the assertion aspect Subprogram\_Variant.
* Do not apply the restriction aspect Restrictions (No\_Recursion).