Technical report

ESVW-TN-FTC-001 - Fault Type Catalog

Catalog of Fault Types in Space Software from the ESVW investigation

Dr. Rainer Gerlich / BSSE
Dr. Ralf Gerlich / BSSE

Dr. Christian Prause / DLR (Ed.)

Version 1.0, June 2016

DLR

Bonn, June 2016

Foreword to the fault type catalog

Space software may contain faults. However, as faults onboard a spacecraft might develop into extremely costly software and system failures during a mission, every effort is made to remove faults before costly failures commence. One building block of prevention methods is to reduce the risk of occurrence of runtime errors through verification.
There are numerous tools on the market that promise to detect many of the various faults that can cause failures at runtime of the software. Yet it is unclear, what the different tools can achieve in practice, and what tools to best apply to real-world space software. Will they be able to detect all the faults they promise to detect, and will they differ in their fault detection capabilities? What efforts are typically associated with verification using such tools?
Therefore, the product assurance department of the DLR Space Administration initiated the ESVW project ("Evaluierung von Softwareverifikationsmethoden und –Werkzeugen") under contract number 50PS1502, executed by *BSSE Software and System Engineering* and *etamax space GmbH*. The aim of the ESVW project was to develop and evaluate a technique for investigating, comparing and characterizing different software verification methods and tools in the context of space flight software.
As part of the evaluation activities, software faults and the problem reports generated by the different tools had to be harmonized. In particular, the way how tools report findings differ quite significantly. We needed a mapping from reported presumable faults to actual faults and false reports, and a classification scheme for known faults.
This technical report presents the detailed work results regarding the fault ontology and mapping.

Dr. Christian Prause
(Project manager of ESVW)

Disclaimer

(Page intentionally left blank.)

Table of Contents

1. Introduction

This document is an outcome of the ESVW activity (Evaluation of Software Verification Methods and Tools) and lists a set of known fault types and as such may be the basis for the categorisation of faults.
The list is the result of previous experience, and may continuously be extended and/or modified.

1.1. Terms and Definitions

In the context of this report, the following definitions apply to the terms fault, error, failure, defect and bug (as used in ECSS-Q-HB-80-03A[2]):

- An error is a bad or undesired state in a software system.
- A fault is the cause of an error having its origin in the code, which may also be called a *mistake*.
- A *failure* is a non-compliance regarding external behavior being recognized between expected and observed properties of the software product as a consequence of an error.
- A *defect* commonly refers to troubles with a software product, with its external behavior or its internal features. This includes consideration of the risk of faults by potential changes of the context.
- A *bug* is a synonym for a fault (as used by testers).

Note that an error can be encountered either while abstractly reasoning about the software, e.g. in the context of the virtual semantics of a programming language, or during actual execution, e.g. on the intended target hardware or on some host system.
From these definitions a chain of causality results as shown in Fig. 1-1.

Term	**Scope**
Fault	Mistake in code
⇓	⇓
Error	Bad state of a system
⇓	⇓
Failure	Unexpected observed behaviour

Fig. 1-1 Causality Chain Fault, Error, Failure

The term *fault coverage* describes the degree to which faults present in the software are or were detected or detectable in the course of the defined verification process. It is usually represented by the ratio of the number of faults recognised and the number of faults present, while the exact value of the latter is typically unknown and can at best be estimated.
A *fault type* represents a set of faults which are grouped together due to common properties. For example, the fault type "buffer overflow" consists of all faults that result in write access beyond the defined bounds of a specific object in memory.

Note that the classification of faults according to fault types does not have to be unique: A fault may be a member of more than one fault type, and there may or may not be a hierarchy of fault types. For example, the mentioned "buffer overflow" fault type is a subset of the more generic "undefined result" fault type, which is defined as all faults that may lead to an undefined result as defined by the C-Standard, as well as of the "memory corruption" fault type.

1.2. Verification Issues

Today, standards such as ECSS, DO178 or EN 50128 require the definition of a verification process by which a software supplier shall demonstrate that the software has the required properties and in addition does not have undesired properties.
Desired and undesired properties are usually expressed in the form of requirements. Some of these requirements address specific parts of the system or the code, while others are generic in nature, possibly applicable to every line of code. Examples for these generic requirements are the RAMS requirements on Reliability, Availability, Maintainability and Safety.
Such generic requirements are typically expressed in the form of generic rules, such as semantic constraints expressed in language standards, coding guidelines or other guides and standards which are mostly application-independent, but may be specific to the application domain (e.g. space, aviation, medical applications, etc.)
One of the current approaches to demonstrate compliance between properties of a software product and requirements is to analyze the source code and to run tests, aiming to cover both, compliance and non-compliances.
Due to the increasing amount of software, its increasing complexity and its criticality, verification tools –represetning another, complimentary approach – shall also help to decrease the effort and to increase the reliability of this process.
It is next to impossible that a verification tool can identify all faults present, just based on the very generic definition of the term. Thus these tools are generally developed based on a set of generic rules, a violation of which usually indicates the presence of a fault.
As a consequence, it seems reasonable to characterise verification tools by the fault types they are able to detect.
In an ideal world, the capability to identify faults of a given type would imply completeness and correctness, meaning that a report about a fault is issued if and only if such a fault is present. Unfortunately, already theory suggests that at least one of genericity, correctness or completeness must be sacrificed. For example, it is impossible to devise an algorithm which can determine for any given program and any given input whether the program will terminate on that input, i.e. finish within finite time. This is the so-called Halting Problem. As a further consequence, it is impossible to derive an algorithm for a given non-trivial property that detects for any program provided as input whether that property holds for that program. A property is non-trivial as long as there is at least one program for which it holds and another program for which it does not hold. This result is known as Rice's Theorem.

Practice further constrains the best capabilities to be expected from verification tools. Firstly, resource limitations – time, memory – may limit the accuracy of the analysis. Secondly, verification tools themselves are software products and thus may also contain faults.
Thus, a verification tool may – for one or more reasons – fail to report a fault or may report a fault where no fault is present. There are 4 cases to be distinguished based on the actual presence of a fault and the presence of a report, as shown in Fig. 1-2, where green indicates the correct cases and red the wrong cases.

		Code	
		Defect present	*Defect NOT present*
Result	*Defect Reported*	true positive	false positive
	Defect NOT reported	false negative	true negative

Fig. 1-2 Classification of Tool Reports on Faults

The following abbreviations are used for the 4 cases:

TP	true positive
FP	false positive
TN	true negative
FN	false negative

Fault coverage can be defined based on this classification as the ratio of true positives and the sum of true positives and false negatives.
Sensitivity or *recall* – as used in information retrieval theory – is defined as the quotient TP/(TP+FN). It represents the portion of present defects that was identified by the analysis approach. A sensitivity of 1 thus indicates that all defects present were found.
Precision is defined as the quotient TP/(TP+FP) and represents the portion of reported defects that are actual defects. A precision of 1 thus indicates that all defects reported were true positives.
These relations are illustrated in Fig. 1-3.

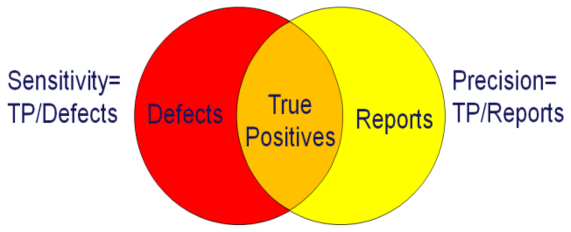

Fig. 1-3: Sensitivity and Precision

The actual number of unreported faults (FN) is unknown. In this activity, the number of FN has therefore been approximated by assuming that the overall set of actual defects (true positives, TP) reported by all tools and found during review of the reports represents the total set of defects present in the software, as illustrated in Fig. 1-4.
This assumption is optimistic to the advantage of the tools as even in combination they may not find all relevant defects in the software. This means that the sensitivity figures derived from the analysis represent an upper bound on the actual sensitivity of the tools.

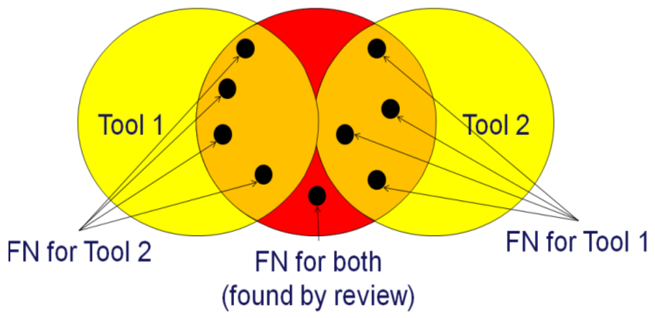

Fig. 1-4: Identifying False Negatives by a Set of Tools

Further, the number of false negatives (FN) can only be considered to address the defect types actually supported by the tools in combination. This means that – in the context of this evaluation – an unreported defect of any other type does not count towards the set of false negatives.

The most critical case regarding software quality from a principal point of view are false negatives, i.e. faults which are present, but not reported. These faults cannot be fixed due to lack of information about their existence.
However, from a usability point of view, false positives may be just as devastating for software quality: If a tool produces too many fault reports, time and budget constraints may require to limit analysis of the reports to a subset. As it cannot be known in advance whether a report represents a true or a false positive, the subset may very well exclude reports about actual, critical faults. Such reports would therefore effectively become false negatives, just because they go unnoticed.
Thus, while sensitivity may be high in theory, low precision may reduce the actual sensitivity encountered in practice, and this reduction and its amount may remain unknown.

.

2. Documents

[1] ISO/IEC 9899: Information technology - Programming languages – C, 2011.

[2] ECSS-Q-HB-80-03A

.

3. Examples

The following Tab. 3-1 provides – anonymized – examples of known faults and and Tab. 3-2 in some cases – correct – counter examples.
All these examples were found in software in the past.
Each of the fault types is assigned a main type and a subtype. For each fault type pseudo-code and an explanation is provided.

3.1 Fault Examples

Tab. 3-1 provides the observed fault types.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
1	Inconsistent Declarations and Uses of Functions	Mismatch of Integer Parameter Types	*`file1.c:`* `int function(int);` *`file2.c:`* `int function(unsigned int) {` `...` `}`	This may result in an inconsistent interpretation of signed values.
2		Mismatch of Parameter Count	`unsigned long inet_addr();` `const char* server_name = ...;` `unsigned long host_addr =` `inet_addr(server_name);`	**`inet_addr`** is defined as **`in_addr_t inet_addr(const char *`*`cp`*`);`** while it was declared with zero parameters and called with one parameter.
3		Mismatch of Enumeration Parameter Types	common.h `typedef enum { … } TyEnum1;` `typedef enum { … } TyEnum2;` prototypes.h: `#include “common.h”` `extern void func1(TyEnum1 para1);` `extern void func2(TyEnum2 para1);` bodies.c: `#include “common.h”` `#include “prototypes.h”` `void func1(TyEnum1 para1) { … }` `void func2(TyEnum1 para1) { … }`	The function `func2` is declared to have a single parameter of type `TyEnum2` in the header file, while the c-file contains a declaration giving `TyEnum1` as the type for the single parameter. This violates the language rules in that two declarations for the same name – in this case `func2` – exist, declaring the name to have non-compatible types (Clause 6.2.7, Paragraph 2 of the C-Standard[1]).
4		Mismatch of	`typedef enum {EMPTY,FULL} STATE;`	The parameters in the call to `create` are in the wrong order.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

		parameter order in presence of enumeration types	`int create(int flag, STATE state);`
			`int id=create(EMPTY,0xff00ff00);`
				However, the enumeration constants themselves are of type `int` (Clause 6.7.2.2, Paragraph 3 of the C-Standard[1]), and the enumeration type itself is compatible to `char` or a signed or unsigned integer type (Clause 6.7.2.2, Paragraph 4 of the C-Standard[1]).
				The parameter values are assigned to the parameters (Clause 6.5.2.2, Paragraph 4 of the C-Standard) and therefore implicitly converted to the declared parameter type.
				The mix-up of parameters therefore does not violate any semantic rule of the C-Language.

5	Unreachable Code	Code following unconditional jump-statement	`return;`
			`break;`
				The code after the jump-statement clearly is not reachable. However, it cannot be concluded whether the jump-statement itself or the code following it is wrong.

6		Retained test statement	`errCode=SUCCESS;    // TEST !!!!`
			`if (errCode==ERROR) {`
			`}`
				A statement inserted for test purposes was not removed. This is often used to force specific parts of the code to be executed that under development conditions are difficult to reach. Typically this addresses error handling code, which either is forced to be entered or forced to be skipped this way.

7		Branching based on function returning constant value	`if (myfunc(para)==0) ...`
			`else ...`
				If myfunc either always or never returns 0 according to its specification, the condition here is constant, and thus either the then- or the else-part is unreachable.

8		Tautological condition	`void func(int a, int b, int c) {`
			`  if (a*b>=c || c>a*b) {`
				The else-part of the if-statement is never reached, as the condition represents A || !A and therefore is always true.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`    ...`
			`  } else {`
			`    ...`
			`  }`
			`}`	

9	Uninitialised Data	Uninitialised Index	`typedef struct {`
			`  int id[10];`
			`} TyDBEntry;`
			`void func(TyDBEntry *db) {`
			`  int i,j;`
			`  printf(“Id:%d\n”,db[i].id[j]);`
			`}`
				The variables `i` and `j` are used before being initialised. As they are declared in block scope, their value will be undefined, leading to a possible invalid memory access.

10		Conditional initialisation	`void flash(int id) {`
			` char str[100],msg[1000];`
			` switch (id) {`
			`  case XXX: strcpy(str,“defined”);`
			`  default: // nothing`
			` }`
			` sprintf(msg,”msg=%s”,str);`
			`}`
				If the value of `id` is not one of the handled cases, the contents of `str` remain undefined. As a result, the call to `sprintf` may lead to accessing invalid indices of of `str` and writing beyond the end of `msg`.

11		Uninitialised Pointer	`char *fn;`
			`void myfunc() {`
			` if ((fd = fopen(fn,”a”)) == NULL)`
			` {`
			`    ...`
			` }`
				Here, the global variable `fn` is not explicitly initialised, but passed to `fopen`.
				
				Flagging this as a defect may lead to a false positive on subsystem- or system-level if there is an initialisation function that initialises `fn` properly before `myfunc` can be called.

Id	Main Fault Type	Fault Subtype	Example	Explanation
			`}`	

12	Unfulfilled or unchecked contract	Wrong return value	**`File ctrlInit.c`**
			`int ctrlArr[4]={1,1,1,0};`
			
			`File ctrlVal.c`
			`extern int ctrlArr[];`
			`int funcCreatePtr`
			`(char **ptr, int ind) {`
			`  if (ctrlArr [ind])`
			`  {`
			`    *ptr=getStrPtr();`
			`    return SUCCESS;`
			`  } else`
			`    return SUCCESS;`
			`}`
			
			`File ctrlExec.c`
			`int funcCreatePtr(char** ptr, int ind);`
			`char *ptr=NULL;`
			`if (ERR==funcCreatePtr(&ptr,3))`
			`  exit(99);`
			`else`
			`  strcpy(ptr,”xxx”);`
				The (implicit) contract of `funcCreatePtr` allows the function to return `SUCCESS` only if `*ptr` was properly initialised. However, `funcCreatePtr` always returns `SUCCESS`, so that the code in `ctrlExec.c` will never enter the then-part of its if-statement and may exhibit a write access to a `NULL`-pointer via the call to `strcpy`.
				
				The passing of a wrong return value in general is a functional defect, but its consequences may lead to runtime errors. Therefore there may be faults in this type which are easily detected by various generic methods, while others may only be detectable using application-specific review and test.

13		Unchecked Index	`void resetTime(int sub,`
			`               int status,`
			`               int activity) {`
				The (implicit) contract specifies that `activity` may not be outside the bounds of the array `execTime`.
				
				The fulfillment of the contract can be easily checked by

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  switch (sub) {`
			`  case XXX:`
			`    execTime[activity].tv_sec = 0;`
			`    break;`
			`  }`
			`}`
				`resetTime` itself.

14		Unchecked length	`typedef struct TyMsg {`
			`  unsigned char msgLen;`
			`  char msgData[128];`
			`} TyMsg;`
			`char buf[128];`
			`void handleMsg(TyMsg* msg) {`
			`  memcpy(buf,`
			`         &msg->msgData[0],`
			`         msg->msgLen);`
			`}`
				The value of `msgLen` is expected to be no more than 128, but this is not checked, leading to two possible out-of-bounds conditions, one by reading beyond the end of `msgData`, the other by writing beyond the end of `buf`.
				This could be avoided by checking `msgLen` inside `handleMsg`.

15		Implicit correlation between pointers	`void func(char* start, char* stop) {`
			`  char* ptr;`
			`  for (ptr=start;ptr<stop;ptr++) {`
			`    ...`
			`  }`
			`}`
				The parameters `start` and `stop` are implicitly correlated in that they should refer to the same array object and that `stop` should refer to an element of that array not before that referred to by `start`.
				The first condition is impossible to be checked by standard C constructs. The second implies `stop>=start`, which can be checked, but is not a sufficient condition for fulfilment of the contract.

16		Assumption	`#define MAX_MSG_LENGTH 1200`
			`uint8_t* inputBuffer;`
				In the function `processByte` it is assumed that `inputBuffer`

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

		about length of buffer	
			```
			size_t inputIndex;
			uint8_t byteFromLastCall;
			int foundBOM;

			int receiveMessage(void* msg,
			                   int32_t* bytesRecvd)
			{
			  inputBuffer = (uint8_t*)inMsg;
			  while(isDataReady()) {
			    uint8_t c = getByteFromStream();
			    if(processByte(c)) {
			      *bytesRecvd = inputIndex;
			      return 1;
			    }
			  }
			  return 0;
			}

			int processByte(uint8_t byte) {
			  uint8_t prevByte = byteFromLastCall;
			  byteFromLastCall = byte;

			  if(byte==MARK) { return 0; }

			  if (prevByte==MARK) {
			    switch (byte) {
			    case EOM:
			      foundBOM = 0;
			      return 1;
			    case BOM:
			      foundBOM = 1;
			      inputIndex = 0;
			      return 0;
			    case ESCAPE:
			      if(foundBOM) {
			        inputBuffer[inputIndex] = MARK;
			        if (inputIndex<MAX_MSG_LENGTH-
			1)
			```
				points to a buffer of at least `MAX_MSG_LENGTH` bytes. Whether this is the case can only be determined from the contexts in which `processByte` and `receiveMessage` are called.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
			```
			                inputIndex++;
			      }
			      return 0;
			    }
			  } else {
			    inputBuffer[inputIndex] = byte;
			    if(inputIndex < MAX_MSG_LENGTH - 1)
			      inputIndex++;

			    return 0;
			  }
			}
			```

17		Implicit correlation between data and code	
			```
			typedef struct TyCmdDescr {
			      unsigned int cmd;
			      unsigned short      minLen;
			      unsigned short      maxLen;
			} TyCmdDescr;
			TyCmdDescr cmdDescr[ ]={
			  {cmd1,3,10},
			  …
			};
			char buffer[65532];

			void recvCmd(char        *cmdData,
			             unsigned int offset,
			             unsigned int entry) {
			      unsigned short len;
			      memcpy(&len,
			           cmdData+offset,
			           sizeof(len));
			```
				In this example, the length passed to `memcpy` in `execCmd` may become much greater than 65532, if `len` is less than 3. The result of `len-3` may become negative, and by way of integer conversion may be interpreted as a very (large) unsigned integer.
				
				The (implicit) contract seems to contain a requirement towards the caller of `execCmd` to ensure that the `len` is not less than 3.
				
				The caller – in this case `recvCmd` – ensures the fulfillment of this requirement by checking the length against a length derived from a table.
				
				Changes to `cmdDescr` may introduce violations of the contract.
				
				However, from a design point of view, this requirement does not have to be part of the contract or at least could be easily checked by `execCmd`.
				
				Further, the implicit contract contains a requirement that the buffer pointed to by `cmdData` must be large enough to fit the data of the length given by `len-3` (assuming `len>=3`).

Id	Main Fault Type	Fault Subtype	Example	Explanation

			```
				if (len >=
			cmdDescr[entry].minLen)
					execCmd(cmdData,offset);
			}
			void execCmd(char	*cmdData,
					unsigned int offset){
				unsigned short len;
				memcpy(&len,
					cmdData+offset,
					sizeof(len));
				memcpy(buffer,
					cmdData+offset+sizeof(len),
					len-3);
			}
			```
				This cannot be checked or enforced by standard language means. Passing the size of the buffer pointed to by `cmdData` would help verification at the lower levels, but the correspondence between given and actual size would have to be enforced elsewhere.

18	Access out of array range	Terminating 0 of string not considered	
			```
			const char cstr[]=”12345678”;
			typedef enum {false,true}
			 Boolean;
			void func() {
			  char vstr[8];
			  Boolean bool;
			  strcpy(vstr,cstr);
			}
			```
				The variable `vstr` is declared as an array of 8 `char` elements, while the constant string in `cstr` consists of 9 `char` elements, including the terminating 0. The call to `strcpy` therefore results in a buffer overflow.
				
				The result is highly architecture-dependent. The placement and order of the relevant bytes of `bool` in memory depends on the alignment requirements and the byte order of the platform.
				
				The compiler is free to chose the integer base type used to represent the enumeration type `Boolean` ([1], Clause 6.7.2.2).
				
				In the first two cases, the compiler may even decide not to store `bool` in memory at all, if register usage allows and no address of

19			
			```
			const char cstr[]=”12345678”;
			typedef enum {false,true}
			 Boolean;
			void cpyfunc(char vstr[8]) {
			```

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  strcpy(vstr,cstr);`
			`}`
			`void func() {`
			`  char vstrloc[8];`
			`  Boolean bool;`
			`  cpyfunc(vstrloc,true);`
			`}`
				`bool` is taken anywhere in the function `func`.
				
				Different variants with varying degrees of locality of the information relevant for detecting this defect are given.

20			`const char cstr[]=”12345678”;`
			`typedef enum {false,true}`
			` Boolean;`
			`char vstr[8];`
			`Boolean bool;`
			`void func() {`
			`  strcpy(vstr,cstr);`
			`}`	

21		Invalid array length	`#define SIZE 500`
			`int i,j,arr[SIZE],result;`
			`for (i=0;i<SIZE;i++)`
			`  for (j=0;j<SIZE;j++)`
			`    result=arr[i+j];`
				In this case it is obvious at multiple points during the execution of the loops, the index will be out of range, most specifically during the last iteration of the outer and inner loop (i+j==2*SIZE-2). According to the C-Standard, the value of result after execution of the loops is therefore undefined.
				
				The example is iterated in the source code for matters of tool evaluation.

22		Invalid Index	`#define QUEUE_SIZE  10`
			`#define BUFFER_SIZE 1024`
			`typedef struct TyQueue {`
				In this part of an implementation of a ring-buffer, an out-of-range-condition may occur in the expressions `queue[head-1]`, namely after a wrap-around of the ring-buffer, with the buffer not

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
			(code below)	being full. This can happen if elements from the start of the ring-buffer are being processed and removed, while the buffer is constantly filled and nevery becomes completely empty in the process.

```
  unsigned int start;
  unsigned int len;
} TyQueue;
unsigned int head=0;
unsigned int tail=0;
unsigned int next=0;
bool    full=FALSE;
TyQueue queue[QUEUE];
char    *buffer[BUFFER_SIZE];

void storeQueue(char *data,
                unsigned int sz){
  if (head!=tail || full==TRUE) {
       next=queue[head-1].start +
            queue[head-1].len;
  } else {
      next = 0;
  }
  if ((next+sz) < BUFFER_SIZE){
    queue[head].start=next;
    queue[head].len  =sz;
    memcpy(buffer+next,data,sz);
    head++;
    if (head >= QUEUE_SIZE)
      head=0;
    if (head == tail)
      full=TRUE;
```

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  }`
			`}`	

23		Invalid size passed to memcpy due to signed/unsigned mixup	`#define MAXLEN 128`
			`typedef struct TyMsg {`
			`  char msgLen;`
			`  char msgData[MAXLEN];`
			`} TyMsg;`
			`TyMsg theMsg;`
			`char buf[MAXLEN];`
			`void getMsg(TyMsg* msg);`
			
			`getMsg(&theMsg);`
			`memcpy(buf,`
			`       &theMsg.msgData[0],`
			`       theMsg.msgLen);`
				The signedness of `char` is implementation-defined (Clause 6.2.5, Paragraph 15 of the C-Standard[1]).
				In an implementation where `char` is signed, a data element of 128 bytes – the maximum value allowed here – would be represented by a value of 128 in `msgLen`, which would be interpreted as -128.
				Passing this as the third parameter of `memcpy` would imply an implicit cast to `size_t`, an unsigned integer type. Therefore `memcpy` would interpret the length given as the maximum value of `size_t` minus 127 (Clause 6.3.1.3, Paragraph 2 of the C-Standard[1]), which would usually represent at least the size of the address space minus 128.
				The example is iterated in the source code for matters of tool evaluation.

24		Buffer Overflow in Structure	`typedef struct {`
			`  char buf[20];`
			`  char marker;`
			`} TyRecord;`
			`void func(TyRecord* record) {`
			`  record->buf[20]='A';`
			`}`
				The array access writes beyond the space allocated to buf. However, the memory written to is accessible as well, and the access will thus not raise an exception, which makes detection difficult.

25		One-off size calculations	`#define LEN 10`
			`char myChArr[LEN];`
				The 0 is written into invalid memory (out-of-range).

Id	Main Fault Type	Fault Subtype	Example	Explanation
			`myChArr[LEN]=0;`	

26		One-off size calculations	`typedef struct{`
			`   size_t source;`
			`   size_t length;`
			`} TyQueueEntry;`
			
			`TyQueueEntry queue[MAX_QUEUE_SIZE];`
			`char buffer[MAX_BUF_SZ];`
			`unsigned int head, tail;`
			`unsigned int full;`
			
			`TyResult write_data(void* data,`
			`                    size_t data_len)`
			`{`
			`  unsigned int start;`
			`  unsigned int space_avl;`
			`  TyResult     result;`
			
			`  if((head != tail) || full!=0)`
			`     start =`
			`       queue[head - 1u].source +`
			`       queue[head - 1u].length);`
			`  else`
			`     start = 0u;`
			
			`  if((MAX_BUF_SZ-(data_len-1u))< start)`
				The call to memcpy can lead to a one-byte-overflow in buffer. The reason for this is the `-1u` in `(MAX_BUF_SZ - (data_len - 1u)) < start`. If `start` is `MAX_BUF_SZ +1-data_len`, this condition will be false and the data will be written into `buffer` starting at `start`, with the last byte being written at index `MAX_BUF_SZ`.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`   start = 0u;`
			
			`if(head==tail && full==0)`
			`   space_avl = MAX_BUF_SZ;`
			`else`
			`   space_avl =`
			`     (queue[tail].source – start) %`
			`     MAX_BUF_SZ;`
			
			`if(space_avl >= data_len)`
			`{`
			`  memcpy(&(buffer[start]),`
			`         data,data_len);`
			`  if(full==0)`
			`  {`
			`    queue[head].source = start;`
			`    queue[head].length = data_len;`
			`    head++;`
			`    if (head==MAX_QUEUE_SIZE)`
			`      head = 0;`
			`    if(head==tail)`
			`      full = 1;`
			`    result = ok_data_queued;`
			`  }`
			`  else`
			`    result = nok_queue_full;`
			`}`	

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  else`
			`    result = nok_buffer_full;`
			`  return result;`
			`}`	

27		Wrong index value	`#define NUM_ELEMS 4`
			`typedef enum {FALSE ,TRUE   } TyBool;`
			`typedef enum {NOSUCC,SUCCESS} TyStatus;`
			`TyStatus OSfunc(int threadId);`
			`TyBool elemList[NUM_ELEMS]=`
			`    {FALSE, FALSE, FALSE, FALSE};`
			`void myFunc(){`
			`  int      elemId=0, freeElem = -1;`
			`  TyBool   free_elem_found = FALSE;`
			`  TyStatus status;`
			`  while ((free_elem_found == FALSE) &&`
			`         (elemId < NUM_ELEMS)){`
			`    if (FALSE == elemList[elemId]){`
			`      freeElem         = elemId;`
			`      elemList[elemId] = TRUE;`
			`      free_elem_found  = TRUE;`
			`    }`
			`    elemId++;`
			`  }`
			`  if (TRUE == free_elem_found){`
			`    status = OSfunc(elemId);`
			`    if (status!=SUCCESS){`
				The function given here shall find an unused entry in `elemList`, marked by the value of the entry being `FALSE`. The entries in `elemList` represent threads available to process jobs.
				
				Once such an entry is found, it is marked as used and an operating system function `OSfunc` is called to wake up the relevant thread. This function may return an error. If so, the original entry is intended to be freed again.
				
				However, the function contains a functional error in that not original entry is set to `FALSE`, but rather the entry following it.
				
				This is due to the fact that instead of `freeElem`, which is assigned the index of the element to be used, `elemId` is used as index when resetting the entry. Before exiting the loop, the statement incrementing `elemId` is being traversed once again, so that after that `elemId` points to the element after `freeElem`.
				
				This functional defect only violates language rules when only the last entry of `elemList` is `FALSE`. In that case, the reset would access the element behind the end of the array, but only if the operating system function returns an array.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`      /* error handling */`
			`      elemList[elemId] = FALSE;`
			`    }`
			`  }`
			`  return;`
			`}`	

28		Wrong size	`void copy(int dest[20], int src[20]) {`
			`  memcpy(dest,src,sizeof(src));`
			`}`
				This will not copy 20 integers, as possibly expected, but rather only as much bytes as are required to store a pointer to integer.
				
				Parameters of array type are demoted to parameters of equivalent pointer type. Thus the type of `src` is not “array of 20 integers”, but instead “pointer to integer”.

29		Missing table termination	`typedef struct {`
			`  unsigned char id1;`
			`  unsigned char id2;`
			`  ...`
			`} TyEntry;`
			`#define INVALID_ID 0xFF`
			`TyEntry pkttab[TABLEN]={`
			`  {0,0,...},`
			`  {0,0,...},`
			`  {0,1,...},`
			`  {1,0,...},`
			`  {1,2,...},`
			`  {1,2,...}`
			`};`
				Here, a table is iterated in three levels. The table contains entries grouped by two identifiers, `id1` and `id2`. The middle loop is intended to iterate over consecutive entries with common `id1`, and the inner loop shall iterate over consecutive entries with common `id1` and `id2`.
				
				For this, three index variables, `idx1`, `idx2` and `idx3` are used. The inner loop increments `idx3`, which is then used to forward `idx2`, which in turn is used to forward `idx1`. Only the outer loop contains a check for validity of `idx1`.
				
				Thus at some point the state will be such that the inner loop is entered with `id1`=1, `id2`=2, `idx1`=4, `idx2`=4 and `idx3`=5. The loop condition holds, so `idx3` is incremented.
				
				The loop condition is checked again, by accessing the element

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`#define TABLEN 6`
			`unsigned int idx1 = 0;`
			`unsigned int idx2 = 0;`
			`unsigned int idx3 = 0;`
			`unsigned char id1, id2;`
			`while(idx1<TABLEN) {`
			`  id1 = pkttab[idx1].id1;`
			`  id2 = INVALID_ID;`
			`  while((pkttab[idx2].id1== id1) &&`
			`        (id2 != pkttab[idx2].id2)) {`
			`    id2 = pkttab[idx2].id2;`
			
			`    while((pkttab[idx3].id1 == id1) &&`
			`          (id2 == pkttab[idx3].id2)) {`
			`      ...`
			`      idx3++;`
			`    }`
			`    idx2 = idx3;`
			`  }`
			`  idx1 = idx2;`
			`}`
				with index 6 - which is beyond the end of the loop.
				
				This can be avoided by adding a terminating entry to the loop, with `id2==INVALID_ID`, or by checking for the end of the table (`idx2`,`idx3<TABLEN`) in every loop.

30	Malformed logical expressions	Assignment instead of comparison	`void func(int *a,`
			`          int b,`
			`          int *c,`
			`          int d) {`
			`  if (*a=b && *c=d) {`
				The condition contains assignment operators instead of equality comparison operators.
				
				As the logical-and operator has higher precedence than the assignment operator, the expression is equivalent to `*a=(b && (*c=d))`.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`    ...`
			`  }`
			`}`
				If `b` is zero, the value of `*c` remains unchanged, `*a` is assigned 0 and the body of the if-statement is not entered.
				If `b` is zero, `d` is assigned to `*c`. The body of the if-statement is entered if and only if `d` is non-zero.

31		Assignment instead of comparison	`int x;`
			`x=...;`
			`if (x=1 || x=2) ...`
			`else ...`
				The variable `x` is always assigned 1 leading to the else-branch never being executed. The second part of the disjunction is never evaluated.

32		Conditional Initialisation due to short-circuit code	`int func1();`
			`char* ptr;`
			`if (func1()==0 ||`
			`    (ptr=malloc(40))==NULL) {`
			`  ...`
			`}`
			`strcpy(ptr,”something”);`
				The call to `malloc` shall serve to initialize `ptr`, but if `func1` returns 0, `malloc` will not be called.

33		Condition depending on pointer values is constant	`#define LOG_FILE_PATH "ramd:/logdir/"`
			`if (LOG_FILE_PATH == NULL) {`
			`  ...`
			`} else {`
			`  ...`
			`}`
				As two constant string pointers are compared to each other, the result of the comparison is constant as well. This means that either the then- or the else-part is dead code.
				Most likely, the then-part will never be reached, as the string is not expected to be stored at address 0.

34		Condition depending on function pointer is constant	`void myfunc(int);`
			`if (myfunc==0) ...`
			`else ...`
				The then-part of the if-statement is never executed, as `myfunc` is the address of a function and therefore constant and not `NULL`.
				Possibly, the condition should have read `myfunc()==0`.

Id	Main Fault Type	Fault Subtype	Example	Explanation

35		Comparison of floating point values for equality	`int func(float p, float q, float x) {`
			`  if (x*x+p*x+q==0.0) {`
			`    return 1;`
			`  } else {`
			`    return 0;`
			`  }`
			`}`
				Floating point arithmetic is – independently of the precision of the floating point type – prone to numerical inaccuracies. Therefore, direct comparison for equality seldomly is appropriate.

36	Duplicate Declarations	Data declaration without intialisation	*`File myFile1.c`*
			`int multiDecl;`
			`File myFile2.c`
			`int multiDecl;`
				The C-Standard allows declarations with the same name to exist in different compilation units (Clause 6.2.2 of the C-Standard[1]).
				
				The concept of linkage exists to decide whether these refer to the same object. Names declared with *external linkage* refer to the same object also across compilation unit boundaries. Names declared with *internal linkage* refer to the same object only within a compilation unit.
				
				A name can be declared at global to have internal linkage by adding the storage-class specifier `static`. Sometimes, this addition is forgotten, leading to the name referring to the same object, although it was originally intended to refer to different objects in different compilation units.
				
				During the linking process usually this can only be detected if at least two of the declarations also contain an initialisation of the object. In that case the linker will have to report an error. In all other cases, no language rule is violated, although the defect may have consequences for the semantics of the code.

37		Data declaration with initialisation	*`File myFile1.c`*
			`int multiDecl=0;`
			`File myFile2.c`
			`int multiDecl;`	

38		Incompatible	*`File file1.c:`*	The variable declarations for `enumVar` refer to the same object,

Id	Main Fault Type	Fault Subtype	Example	Explanation

		Declarations	`typedef enum {`
			`  value1 = 0,`
			`  value2 = 1`
			`} TyEnum;`
			`TyEnum enumVar;`
			`File file2.c:`
			`typedef enum {`
			`  value1 = 1,`
			`  value2 = 2`
			`} TyEnum;`
			`TyEnum enumVar;`
				as they have external linkage. The C-Standard[1] requires the types to be compatible, which in case of enumeration types means that constants of the same name also must have the same value(Clause 6.2.7, Paragraphs 1f). However, as the compiler will not see `file1.c` and `file2.c` at the same time, and usually no information is output to the linker about the enumeration-specifiers, this inconsistency cannot be detected.

39	Format String Issues	Using „%s“ for an integer parameter	`int a;`
			`printf(“%s\n”,a);`
				Not only is the behaviour undefined, but may also lead to read accesses from invalid adresses if the value of `a` does not contain a zero byte.

40		Too many arguments	`int a, b;`
			`printf(“%d\n”,a,b);`
				The output might be intended to list the value of `b` as well. Otherwise, this is non-critical.

41		Too few arguments	`int a;`
			`printf(“%d:%s\n”,a);`
				The output is invalid, and in case a string is expected, this may lead to invalid memory access.

42		Signedness Mixup	`char c;`
			`printf(“%u\n”,c);`
				The output is intended to be unsigned, with `c` representing a byte value (0-255). However, the type `char` may be signed (implementation dependent), in which case it is cast to `int` before passing it as argument to the variable argument list, resulting in a value in the range `UINT_MAX-127` to `UINT_MAX` or 0 to 127 being output.

Id	Main Fault Type	Fault Subtype	Example	Explanation
43	Macro Issues	Multiple statements in one macro	#define XY stmt1; stmt2 if (COND) XY;	The programmer expects both statements to be conditional, while only the first actually is.
44		Side-effects in macro parameters	`#define MAX(a,b) ((a)>(b)?(a):(b))` `char func(char* ptr, int x, int y) {` `int max = MAX(x++,y);` `return ptr[max];` `}`	The expression `x++` will be evaluated twice if the original value of `x` is larger than that of `y`. Otherwise, `x++` will be evaluated only once.
45		Redefiniton of the language	`#define return ;`	Here the semantics of the keyword `return` are changed drastically.
46	Dereferencing invalid pointers	NULL-pointer access (conditional initialisation, unconditional use)	`int Day(time_t myTime)` `{ struct tm *myLocalTime;` `myLocalTime = localtime(&myTime);` `if(myLocalTime ==NULL)` `printf(„no conversion\n");` `return myLocalTime->tm_mday;` `}`	Here, the return value of `localtime` may be `NULL`, which is checked for in the if-statement. However, the return statement can be executed even `myLocalTime` is `NULL`, leading to an invalid memory access.
47		NULL-pointer access (unconditional initialisation, unconditional use)	`char *str;` `str=malloc(100);` `str=0;//should be *str=0;` `...` `strcat(str,”xyz”);`	The pointer is initialised instead of the area it is pointing to. Apart from that, another fault is present as the return value of `malloc()` is not checked.
48		NULL-Pointer	`int flashRead(int fd,`	In this example, `fileSizePtr` is initialized to `NULL` in the

Id	Main Fault Type	Fault Subtype	Example	Explanation

		Access (unconditional initialisation, conditional use)	`              size_t offset,`
			`              void* dest,`
			`              size_t size);`
			
			`unsigned long fileSize;`
			`if (COPY_FROM_FLASH == copy_mode)`
			`{ unsigned long *fileSizePtr = NULL;`
			`  if (flashRead(flashIO,`
			`                offset+totalBytes,`
			`                (void*)fileSizePtr,`
			`                sizeof(unsigned long))`
			`      == ERROR)`
			`  { //error reading file size}`
			`  else`
			`  { //success reading file size`
			`    fileSize = *fileSizePtr;`
			`  }`
			`}`
				beginning. The call to `flashRead()` cannot change that. Therefore `fileSizePtr` is still `NULL` when entering the else-part of the if-statement, leading to an invalid memory access.

49		NULL-Pointer Access (conditional initialisation, unconditional use)	`void fullName(const char *path,`
			`              const char *fn,`
			`              char *ffn){`
			` if (path == NULL)`
			`    strcpy(ffn, fn);`
			` else`
			`    sprintf(ffn,”%s/%s”,path,fn);`
			`}`
				The case `fn==NULL` is not considered.

Id	Main Fault Type	Fault Subtype	Example	Explanation

50		`realloc` without pointer update	`void func1(int* ptr, size_t sz) {`
			`  ptr=realloc(ptr,(sz+1)*sizeof(int));`
			`  ptr[sz] = 1;`
			`}`
			`void func2(size_t sz) {`
			`  int* ptr = malloc(sz*sizeof(int));`
			`  ...`
			`  func1(ptr, sz);`
			`  free(ptr);`
			`}`
				If `realloc` cannot grow the original memory block to the desired size, a new memory block is allocated, the data is copied to the new block and the old block is freed.
				As `func1` does not give the new address of the block back to `func2`, `func2` may use an outdated pointer to access that memory block and to free it. The memory block may already have been reused in an intermediate call to `malloc`.

51		Access via invalid pointer (`!=NULL`)	`int* ptr = ...; /* not NULL */`
			`int i = 10;`
			`while (i>0) {`
			`  *ptr = i;`
			`  ptr++;`
			`  i--;`
			`}`
				The pointer may point to an invalid address space or to the wrong object.

52	Name Overloading	Type and variable names	`struct xy {`
			`...`
			`} xy;`
				In the C-Language, the tags of struct, union and enumeration specifiers reside in a namespace separate from that in which other names – such as those of variables – are listed. The struct in this case can always be accessed unambiguously by struct xy, while the variable is referred to by xy.
				However, this impacts readability and thus also verifiability.

53	Pointer Arithmetic	Arithmetic on pointer to void	`void* myBase=...;`
			`void* ptr=myBase+4;`
				The C-Standard[1] only allows pointers to complete object types as operands to additions or subtractions for pointer arithmetic

Id	Main Fault Type	Fault Subtype	Example	Explanation
		or function type		(Clause 6.5.6, Paragraph 2). Function types are not object types in general, and `void` is not a complete object type. Compilers might still generate code for such an operation, with undefined result, as the size of `void` or function types is not defined.

54		Unchecked conversion from int to pointer	`int func(int a) {`
			`  return *(int*)a;`
			`}`
				The information, that the argument actually represents an address in space, is hidden.
				Also, it is not guaranteed that `int` is wide enough to represent a pointer. Instead, the C-Standard provides `intptr_t` for that (Clause 7.20.1.4 of the C-Standard[1]).

55	Missing return for non-void function	Conditional return	`int func(int a) {`
			`  if (a!=0) {`
			`    return 1;`
			`  }`
			`}`
				If `a` is zero, the value of the function call expression is undefined (C-Standard[1], Clause 6.9.1, Paragraph 13).

56	Non-Termination	Endless loop	`void func(unsigned int step,`
			`          unsigned int limit)`
			`{`
			`  unsigned int i;`
			`  for (i=0;i<limit;i+=step)`
			`    ...`
			`}`
				The loop will never terminate if `para` is zero.

57		Endless Recursion	**`File fault.h`**
			`void verifyFault(int ind);`
				The function `handleFault` may be conditionally called recursively, without termination, if `ind` is not handled by any of

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`void handleFault(int ind);`
			
			`File verify.c`
			`#include <fault.h>`
			`void verifyFault(int ind) {`
			`  switch (ind) {`
			`    case 0: ...;`
			`    case 1: ...;`
			`    default: handleFault(ind);`
			`  }`
			`}`
			
			`File verify.c`
			`#include <fault.h>`
			`void handleFault(int ind) {`
			`  switch (ind) {`
			`    case 0: ...;`
			`    case 1: ...;`
			`    default: handleFault(ind);`
			`  }`
			`}`
				the cases in the switch-statements.

58		Unintented Number of Loop Iterations	`unsigned int  itemUsed[..],cc,i;`
			`int           fc;`
			`int setItem(unsigned int);`
			`fc = setItem(cc);`
			`itemUsed[cc] = fc;`
			`for(i=1;i<itemUsed[cc];i++) {`
				Critical cast between signed and unsigned if `setItem` returns -1, if e.g. the value of `cc` is invalid. Then the upper limit of the loop is `UINT_MAX` instead of -1 which generates a quasi-endless loop.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  ...`
			` }`	

59		Unintented Number of Loop Iterations	`void myFunc(int lower, int upper)`
			`{`
			`   int i;`
			`   if ((upper-lower)>100)`
			`       return;`
			`   for (i=lower;i<upper;i++)`
			`       ...;`
			`}`
				As per the C-Standard, the result of the subtraction in the if-condition may be undefined, as it may not fit into an `int`. However, compilers do generate code that usually uses modulo-arithmetics.
				
				Using the minimum `int` value for `lower` and the maximum `int` value for `upper` will lead to `(upper-lower)==-1`. Although this value is less than 100, the loop will iterate a large number of times, e.g. 2^{32}-1 times in case of an 32-bit integer type.

60		Iteration statement not in loop body	`void repeat(char* s, int n, char c) {`
			`  int i = 0;`
			`  while (i<n)`
			`    s[i]=c;`
			`    i++;`
			`  s[n]=0;`
			`}`
				The programmer obviously forgot the curly braces to delimit the body of the `while`-loop. As a result, the increment expression is not part of the loop, and if `0<n` holds in the beginning, the loop will be entered and never terminate.

61	Arithmetic Errors	Overflow allowed by C-Standard	`void func(short a, short b) {`
			`  unsigned short c;`
			`  c=a * b;`
			`  if (c>10) {`
			`  } else {`
			`  }`
			`}`
				The operations in this example are valid and well-defined according to the C-Standard[1]. However, the multiplication may lead to an overflow, which in turn may lead to unintended results.
				
				Both operands to the multiplication are first implicitly promoted to `int`, as all values of type `short` fit into `int` (Clause 6.3.1.1, Paragraph 2 of the C-Standard[1]). The result of the multiplication therefore also is of type `int`.
				
				As the values of the two operands are constrainted to the range

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
				of `short`, the result of the multiplication will fit into the range of `int`, so no exceptional condition in the sense of Clause 6.5, Paragraph 5 of the C-Standard[1] occurs during the multiplication itself. During simple assignment, the value of the right operand to the assignment is converted to the type of the left operand (Clause 6.5.16.1). In this case, `int` is converted to `unsigned short`. In case the result does not fit into the range of `unsigned short`, it is converted by repeatedly adding or subtracting the maximum value of `unsigned short` plus one to or from the value (modulo arithmetic, Clause 6.3.1.3, Paragraph 2 of the C-Standard[1]).

62		Overflow during operation	
			```
			void func(unsigned short a,
			          unsigned short b) {
			  unsigned short c;
			  c=a * b;
			  if (c>10) {
			  } else {
			  }
			}
			```
				While only slightly different from the previous example in that the input parameters are of type `unsigned short` instead of `short`, in this example, the result is undefined, as the result of a multiplication of values in the range of unsigned short do not fit into the range of `int` (C-Standard[1], Clause 6.5, Paragraph 5).
				
				Typically, compilers do generate code for this case, with the result being determined in a similar way as in Example 61.

63		Overflow during assignment	
			```
			void func(short a, short b) {
			  short c;
			  c=a * b;
			  if (c>10) {
			  } else {
			```
				This example differs the previous examples in that the result is stored in a variable of type `short` instead of `unsigned short`.
				
				The result is undefined because the result of the multiplication of the two variables may exceed the range of `short` (Clause 6.3.1.3, Paragraph 3 of the C-Standard[1]). This is different from conversion

Id	Main Fault Type	Fault Subtype	Example	Explanation
			`  }` `}`	to `unsigned short`.
64		Division by zero	`int func(int a, int b) {` `  return a/b;` `}`	The value of `b` may be zero.
65		Square root of negative number	`float solve(float a, float b, float c)` `{` `  return (-b + sqrt(b*b-4*a*c))/(2*a);` `}`	The value passed to `sqrt` may be negative.
66	Type Inconsistencies	`const` qualifier in pointer declarations	`#ifndef UNITTEST` `const` `#endif` `unsigned int *dest=<init value>;` `memcpy(dest,src,len);`	Here, a warning should be issued related to discarding the `const` qualifier in passing `dest` as first argument to `memcpy`, as that argument is declared to be of type `void*` (without `const`). This example – as most of the others – has been found in actual code. The switch seemed to be intended to make certain objects writable during unit test. However, to make dest itself constant instead of the objects it points to, the declaration would have to read: `unsigned int * const dest=...;`
67		Inconsistency between array type and array initializer	`int actl[4] = {0,1,0,0,1,1};`	The array is declared to contain only 4 elements, but is initialised with 6. This seems to violate the rule that no initialiser shall provide a value for an object not contained in the entity being initialised (Clause 6.7.9, Paragraph 2 of the C-Standard[1]).
68	Functional defects	Wrong operator	`int func(short i, short j) {` `  return i*j; //should be +`	In this function a functional defect is present in that the function was thought to calculate the sum of its two parameters,

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
			`}`	but instead calculates the product. Clearly, this code does not violate any rules that the correct implementation would not also violate (most specifically the rule that the result of an arithmetic operation must fit in the result type, here `int`). Therefore this defect can only be detected either by unit tests where the output of the function is verified, or due to fault propagation, e.g. when the result is used as an index in a calling function.

69		Comparison instead of assignment	`void func(int x) {`
			`  int y;`
			`  y==x;`
			`}`
				The expression statement contains comparison operators instead of assignment operators, and thus has no effect.

70	Implicit Conversions		`#define BUFLEN 256`
			`void func(unsigned short s) {`
			`  unsigned char buf[BUFLEN];`
			`  size_t buffer_index = 0;`
			`  buf[buffer_index] = s;`
			`  buffer_index += sizeof(s);`
			`}`
				Judging from the index increment, the intent is to write the value of `s` into consecutive bytes in `buf`.
				
				Due to implicit conversions taking place during assignment, only the least significant byte of the value of `s` is actually stored. The value of the following byte is undefined.

71	Undefined Result	Shift operation	`Unsigned short funcShift16(int width,int pos,int val){`
			`  int bitPos;`
			`  int shifts;`
			`  int mask;`
				The intention of `funcShift16` is to place the value val at a certain bit-position in a window of `width` bits (1..16) within a 16-bit word. In consequence, the range of `shifts` is -15..15. The result of the shift-operation is undefined if shifts <0.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  unsigned short word;`
			`  unsigned short mask;`
			`  mask  =((1 << width) -1) << pos;`
			`  bitPos=pos % 16;`
			`  shifts=16-(bitPos+width);`
			`  word &=~mask;`
			`  val  =val << shifts;`
			`  word|=val & mask;`
			`  return word;`
			`}`	

72	Missing return for non-void function	No return at all	`int func(int id) {`
			`  int ret=0;`
			`  switch (id)`
			`  {`
			`    case 0:  ret=0; break;`
			`    case 1:  ret=1; break;`
			`    default: ret=4;`
			`  }`
			`}`
				The function should return a value.

73	Malformed logical expression		`int myfunc(int arr[],int arrSize,int id)`
			`{`
			`  int ii=0;`
			`  while (ii<arrSize & arr[ii]!=id)`
			`    ii++;`
			`  if (arr[ii]>0)`
				The & should be &&. However, the while-loop is correctly executed, yielding a value of `arrSize` for `ii` when leaving the while-loop when `id` is not found in `arr`. In this case an out-of-range condition occurs for the following `if`.
				As & in contrast to && is not subject of short circuit code, arr is also accessed with index arrSize, and an out-of-range condition occurs.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`    arr[ii]=-arr[ii];`
			`  return ii;`
			`}`	

74			`int myfunc(int arr[],int arrSize,int id)`
			`{`
			`  int ii=0;`
			`  while (arr{ii]!=id && ii<arrSize)`
			`    ii++;`
			`  return ii;`
			`}`
				If id is not found in `arr` an out-of-range condition occurs in the `while` -expression because the check on `arrSize` is performed after the access of `arr`.

75	Run-Time Fault	Division by zero	`double myDivZeroFuncDblErr(double x) {`
			`  double ret;`
			`  ret =1.0/(x*x-9.0);`
			`  ret+=1.0/(x*x);`
			`  return ret;`
			`}`
			`double myDivZeroFuncDblBound(double x)`
			`{`
			`  double ret;`
			`  ret=1.0/(x*x-9.0);`
			`  ret+=1.0/(x*x);`
			`  return ret;`
			`}`
			`int myDivZeroFuncIntErr(int x) {`
			`  int ret;`
				This set of functions shall check whether a division by zero will be detected, for floating-point and integer.
				The postfix *Err* indicates that the function is called with the critical values (0 and 3), while the functions with postfix *Bound* are only called with safe values (1,10).
				
				Polyspace should detect that for the *Bound*-functions no error can occur.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  ret =10000/(x*x-9);`
			`  ret+=10000/(x*x);`
			`  return ret;`
			`}`
			`int myDivZeroFuncIntBound(int x) {`
			`  int ret;`
			`  ret =10000/(x*x-9);`
			`  ret+=10000/(x*x);`
			`  return ret;`
			`}`	

76	Run-Time Fault	Sqrt on negative number	`double mySqrtNegDblErr(double x) {`
			`  double ret;`
			`  ret=sqrt(x*x-20*x+1);`
			`  return ret;`
			`}`
			`double mySqrtNegIntBound(double x) {`
			`  double ret;`
			`  ret=sqrt(x*x-20*x+1);`
			`  return ret;`
			`}`
				This set of functions shall check whether square root of a negative number will be detected.
				The postfix *Err* indicates that the function is called with critical values (1 and 3), while the functions with postfix *Bound* are only called with safe values (100,200).

77	Dereferencing invalid pointers (contd.)	NULL-pointer-access in seemingly dead code	`static int counter = 0;`
			
			`void func() {`
			`  if ((counter & 1) == 0) {`
			`    counter++;`
				The else-part of the conditional statement is clearly reachable, and contains an access to a NULL-pointer.
				However, a tool considering only the initial value of the global variable may assume that the else-branch is unreachable, and thus, that the invalid null-pointer access is irrelevant.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`  } else {`
			`    char* ptr = NULL;`
			`    strcpy(ptr,"something");`
			`  }`
			`  ...`
			`}`	

78		Unchecked return value (localtime)	`#include <time.h>`
			
			`int getdayofmonth() {`
			`  time_t t = time(NULL);`
			`  struct tm* loc = localtime(&t);`
			`  return loc->tm_mday;`
			`}`
				`localtime()` may return `NULL` in case of an error[1]. In that case, the indirect member access would constitute a runtime error.

79		Unchecked return value (malloc)	`#include <stdlib.h>`
			
			`char* getdigitsstr() {`
			`  char* s = malloc(sizeof(char[11]));`
			`  strcpy(s,"0123456789");`
			`  return s;`
			`}`
				`malloc()` may return `NULL` in case of an error[1]. In that case, the pointer access would constitute a runtime error.

80	File Handling	Invalid argument	`retFclose=fclose(fd);`
			`if (feof(retFclose)==0)`
				Invalid argument passed to feof and access of file descriptor after fclose
				
				1. The return value of fclose is passed to feof instead of the file descriptor
				2. The file descriptor is passed to feof after fclose.

Id	Main Fault Type	Fault Subtype	Example	Explanation
				In this case the file descriptor is considered undefined.
81	File Handling	Read invalid data from file	`fgets(buf,BUFLEN,fd);` `divByZero=atoi(buf);` `quot     =100 / divByZero;` `fgets(buf,BUFLEN,fd);` `negSqrt  =(double)atof(buf);` `result   =sqrt(negSqrt);`	Data read from file are invalid, e.g. division by zero and sqrt on negative number
82	Array access	Out-of-range	`int ESVW_func82_2(int *para){` ` for (ii=0;ii<20;ii++)` `}` `int ESVW_func82_1(int *para){` `  ret=ESVW_func82_2(para);` `}` `int arr[10];` `ESVW_func82_1(arr82);`	An array is passed to a function expecting a pointer, which in turn calls another function expecting a pointer. In latter function the pointer is accessed with an invalid index w.r.t. to the boundaries of the arry passed.
83	Array access	Out-of-range	`typedef struct TyMyStruct {` `  int items;` `  int item[ITEM_NO]; } TyMyStruct;` `int myFunc(uint8 *buf, uint8 bufLen,` `TyMyStruct *myStruct){` `  if (buf==NULL || myStruct==NULL)` `  ret=-1;` `  else {` `    if (bufLen<1)` `      ret=-2;` `    else {`	A buffer, a corresponding length and a structure are passed to a function which shall copy the logical elements of the structure from the buffer. In C the allocated memory and the passed length cannot be correlated. In the example the contents of the buffer is inconsistent with the passed length. The number of elements provided in the first byte of the buffer is larger than the what the buffer actually contains. Further, the check on the passed length of the buffer is faulty so that a buffer overflow will occur even if the check is successful. In this example no assumption is made on the context.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`      start=0;`
			`      items=buf[indBuf];`
			`      indBuf+=1;`
			`      if (items> ITEM_NO) {`
			`        items= ITEM_NO; ret=-3; }`
			`      myStruct->items=items;`
			`      ind=0;`
			`      while (items>ind) {`
			`        memcpy(&myStruct->item[ind],`
			`               &buf[indBuf],`
			`               sizeof(int));`
			`        ind++;`
			`        indBuf +=sizeof(int);`
			`      }`
			`    }`
			`  }`
			
			`  return ret; }`	

84	Array access	Out-of-range	`extern int myFunc(uint8 *buf, uint8`
			`bufLen, TyMyStruct *myStruct);`
			`void myFuncContext() {`
			`  uint8 buf[5]={10,0,0,0,0};`
			`  uint8 len=sizeof(buf);`
			`  TyMyStruct myStruct;`
			`  myFunc(buf,len,&myStruct);`
			`}`
				In this example we embed the function of example 83 into a context provided by function myFuncContext.

Id	Main Fault Type	Fault Subtype	Example	Explanation

85	src too small	Out-of-range	`#define NAME_LENGTH_85 30`
			`char dest85[NAME_LENGTH_85];`
			`char name85[] = "xxxx_xx_xxxx";`
			`memcpy(&dest85[1],name,NAME_LENGTH_85 - 1);`
			`      //   29      13           29`
				A standard length for names is defined and used for copying. However, an array is passed which does not have the standard length as it is implicitly defined by an initializer.
				The array is passed as pointer across function boundaries. Therefore the smaller length is not visible when being copied.

86	buffer too small	Out-of-range	`int myFunc86_buf_unconstrainedTopLevel(char *buf, const unsigned int bufSize, myStruct86 *ptr){`
			`  unsigned int ind=0,upLim,idx;`
			`  if (buf == NULL || ptr == NULL)`
			`    ret=-1;`
			`  else{`
			`    if (bufSize < 1)`
			`      ret=-2;`
			`    else {`
			`      ptr->elemNo = buf[ind];`
			`      ind += 1;`
			`      upLim= ptr->elemNo;`
			`      if (upLim>LENGTH_86){`
			`        ret=-3;`
			`        upLim=LENGTH_86;`
			`      }`
			`      idx=0;`
			`      while (upLim > idx){`
				A buffer is passed as pointer, in addition the size is passed as parameter. But both cannot be correlated at level on C language. A struct is passed as pointer into which the contents of buffer shall be inserted. It shall be checked whether the potential that the index of buffer may be out-of-range is recognised.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation
			```
			        read4Byte_86(&buf[ind],
			                     &ptr->elem[idx]);
			      ind += 4;
			      idx++;
			    }
			  }
			  return ret;
			}
			```
87	buffer too small	Out-of-range	*The example is too complex too provide pseudo code here. Only the description is given here.* It shall be checked whether the potential that the index of buffer may be out-of-range is recognised as for example 86. In addition, and this is the essential point here, it shall be checked whether a tool can recognise whether the reset of uplim1 to LENGTH_87 in case uplim1 is greatert than LENGTH_87 causes another defect: in the following loop only LENGTH_87 are read, while more elements may be stored there. Therefore the next step of idx2 starts at the wrong position, a the next element of myStruct87_1, while it expects elemNo_2.	
88	invalid pointer	invalid index	*The example is too complex too provide pseudo code here. Only the description is given here.* The capability to handle (nested) tables with different lenght of rows shall be checked. Indices into the tables are passed as parameters. Their values are derived from other tables.	

89	invalid pointer	invalid index	
			```
			#define NAME_LENGTH_89 30
			int myFunc89(char *passedName){
			   char name89[] = "/---------1-------"
			                   "--2---------3";
			   int ind,ret=0;
			   ind = 0;
			   while((passedName[ind] != 0) &&
			         (ind <= sizeof(name89)))
			```
				The length of a passed string (source buffer) shall be determinned in order to check whether it does fit into the destination buffer.
				A wrong termiantion condition for the loop over the source buffer causes an out-of-range condition if the length of the sourcer really exceeds the length of the destination.

.

Id	Main Fault Type	Fault Subtype	Example	Explanation

			`     ind++;`
			`    if(ind == sizeof(name89))`
			`      ret = -1;`
			`    else`
			`      strcpy(buf,passedName);`
			`    return ret;`
			`}`	

Tab. 3-1: Fault Types

3.2 Counter Examples

Tab. 3-2 provides correct examples for some of the faults shown in Tab. 3-1, aiming to have code to proof that a tool does not report a fault.

Id	Fault Category	Fault Type	Example	Explanation

1	Invisible Correlations	Invisible Correlation between array and array length	`#define MAXLEN 128`
			`typedef struct TyMsg {`
			`  unsigned char msgLen;`
			`  char msgData[MAXLEN];`
			`} TyMsg;`
			`char buf[MAXLEN];`
			`void copyMsg(TyMsg* msg) {`
			`  memcpy(buf,`
			`         &msg->msgData[0],`
			`         msg->msgLen);`
			`}`
			`void processMessage() {`
			`  TyMsg msg;`
			`  msg.msgLen = rand() % (MAXLEN+1);`
			`  copyMsg(&msg);`
			`}`
				In this context, `processMessage` honors the contract with `copyMsg`, according to which `msgLen` may only be in the range 0..MAXLEN.

2	Ring Buffer handling		`typedef struct{`
			`   size_t source;`
			`   size_t length;`
			`} TyQueueEntry;`
			
			`TyQueueEntry queue[MAX_QUEUE_SIZE];`
			`char buffer[MAX_BUF_SZ];`
			`unsigned int head, tail;`
			`unsigned int full;`	

Id	Fault Category	Fault Type	Example	Explanation

```
TyResult write_data(void* data,
                    size_t data_len)
{
  unsigned int start;
  unsigned int space_avl;
  TyResult     result;

  if((head != tail) || full!=0) {
     unsigned int last_index =
       (head-1u) % MAX_QUEUE_SIZE;
     start =
       queue[last_index].source +
       queue[last_index].length);
  } else
     start = 0u;

  if((MAX_BUF_SZ-data_len)<start)
     start = 0u;

  if(head==tail && full==0)
     space_avl = MAX_BUF_SZ;
  else
     space_avl =
       (queue[tail].source – start) %
       MAX_BUF_SZ;

  if(space_avl >= data_len)
  {
```

Id	Fault Category	Fault Type	Example	Explanation

```
  memcpy(&(buffer[start]),
     data,data_len);
  if(full==0)
  {
   queue[head].source = start;
   queue[head].length = data_len;
   head++;
   if (head==MAX_QUEUE_SIZE)
    head = 0;
   if(head==tail)
    full = 1;
   result = ok_data_queued;
  }
  else
   result = nok_queue_full;
 }
 else
  result = nok_buffer_full;
 return result;
}
```

Id	Fault Category	Fault Type	Example	Explanation
3	Constant Array Compatibility		`typedef int TyArray[10];` `void func(const TyArray* a);` `TyArray b;` `func(&b);`	According to the C-Standard, the type of `&b` in the call to `func` is incompatible with the type declared for the single parameter of `func` (which is emitted as a warning, e.g., by gcc 4.7.2). The reason is that the declarator "`const TyArray* a`" does not declare a constant pointer to an array of `int`, but rather a pointer to an array of constant

Id	Fault Category	Fault Type	Example	Explanation
				integers (Clause 6.7.3, Paragraphs 5 and 9[1]). However, it is not clear why it should not be possible to use a pointer to a modifiable array in place of a pointer to a non-modifiable array. It should be noted that this is different from the case where, e.g., `char*` is passed for a `const char*` parameter, because the C-Standard only requires the unqualified versions of the types pointed to to be compatible (Clause 6.5.16.1, Paragraph 1[1]). Passing `char**` for a `const char**` parameter would thus again violate the constraints imposed by the language standard.

4	Correct conditional return		`int func(int a) {`
			`  if (a!=0) {`
			`    return 1;`
			`  } else {`
			`    return 0;`
			`  }`
			`}`
				All branches contain a return statement, and there is none at the end of the function. However, as the end of the function cannot be reached, there is no condition under which the result can be undefined.

5	Correct conditional initialisation		`int func(int a) {`
			`  int ret;`
			`  if (a!=0) {`
			`    ret = 1;`
			`  } else {`
			`    ret = 0;`
			`  }`
				All branches contain proper initialisation statements for `ret`, so that in any case, `ret` is defined before it is used.

Id	Fault Category	Fault Type	Example	Explanation

			`  return ret;`
			`}`	

6	Fulfilled contract		**`File ctrlInit.c`**
			`int ctrlArr[4]={1,1,1,0};`
			
			`File ctrlVal.c`
			`extern int ctrlArr[];`
			`int funcCreatePtr`
			`(char **ptr, int ind) {`
			`  if (ctrlArr [ind])`
			`  {`
			`    *ptr=getStrPtr();`
			`    return SUCCESS;`
			`  } else`
			`    return ERR;`
			`}`
			
			`File ctrlExec.c`
			`int funcCreatePtr(char** ptr, int ind);`
			`char *ptr=NULL;`
			`if (ERR==funcCreatePtr(&ptr,3))`
			`  exit(99);`
			`else`
			`  strcpy(ptr,”xxx”);`
				This is the correct implementation of Example 12. `funcCreatePtr` correctly returns `SUCCESS` only if `*ptr` is properly initialized.

7	Correct length for memcpy		`#define MAXLEN 128`
			`typedef struct TyMsg {`
			`  unsigned char msgLen;`
				This is the correct implementation of Example 23. Here msgLen is of an unsigned integer type, and the length is checked to be in the proper range.

Id	Fault Category	Fault Type	Example	Explanation

			`  char msgData[MAXLEN];`
			`} TyMsg;`
			`TyMsg theMsg;`
			`char buf[MAXLEN];`
			`void getMsg(TyMsg* msg);`
			
			`getMsg(&theMsg);`
			`memcpy(buf,`
			`       &theMsg.msgData[0],`
			`       theMsg.msgLen);`	

8	Correct table termination		`typedef struct {`
			`  unsigned char id1;`
			`  unsigned char id2;`
			`  ...`
			`} TyEntry;`
			`#define INVALID_ID 0xFF`
			`TyEntry pkttab[TABLEN]={`
			`  {0,0,...},`
			`  {0,0,...},`
			`  {0,1,...},`
			`  {1,0,...},`
			`  {1,2,...},`
			`  {1,2,...},`
			`  {INVALID_ID,INVALID_ID,...}`
			`};`
			`#define TABLEN 6`
			`unsigned int idx1 = 0;`
				This is the correct implementation of Example 29. In this case, the table is correctly terminated.

Id	Fault Category	Fault Type	Example	Explanation
			`unsigned int idx2 = 0;` `unsigned int idx3 = 0;` `unsigned char id1, id2;` `while(idx1<TABLEN) {` `  id1 = pkttab[idx1].id1;` `  id2 = INVALID_ID;` `  while((pkttab[idx2].id1== id1) &&` `        (id2 != pkttab[idx2].id2)) {` `    id2 = pkttab[idx2].id2;` `    while((pkttab[idx3].id1 == id1) &&` `          (id2 == pkttab[idx3].id2)) {` `      ...` `      idx3++;` `    }` `    idx2 = idx3;` `  }` `  idx1 = idx2;` `}`	
9	File Handling	Read valid data from file	`fgets(buf,BUFLEN,fd);` `div =atoi(buf);` `quot=100 / div;` `fgets(buf,BUFLEN,fd);` `ops  =(double)atof(buf);` `result   =sqrt(ops);`	Data read from file with valid values, counter example to 81
10	Access out-of-range array	Invalid array length	`#define SIZE 500` `int i,j,arr[2*SIZE],result;`	Counter example to 21, the array size is doubled. The example is iterated in the source code for matters

Id	Fault Category	Fault Type	Example	Explanation
			`for (i=0;i<SIZE;i++)` `  for (j=0;j<SIZE;j++)` `    result=arr[i+j];`	of tool evaluation.

Tab. 3-2: Suggestions for Non-Defects

4. Aspects of Fault Type Classification

The classification is guided by two aspects:

- The mistake made by the programmer, e.g. the wrong thought, the typographical mistake or the violation of interface constraints, and
- the method of detection.

This means that faults are categorised separately if they differ by the kind of programmer's mistake. A fault type may be further split up, if different fault detection methods can be conceived for otherwise similar faults represented by the type.
In Ch. 4.1 aspects of classification in context of standards are discussed. Ch.4.2 provides a list of fault types aiming to harmonize messages from a number of tools.

4.1 Relationship to Standards

Theoretically, standards – such as the language standards – should be a natural source for a systematic classification of faults. The standards describe semantic constraints which define the admissibility of language and programming constructs, and any violation of these constraints could be considered a fault.
However, neither do all faults result from a violation of the semantic constraints of the language, nor do all violations of semantic language constraints actually imply faults.
For the first part of the proposition, it should be clear that not every valid program also conforms with functional and further non-functional requirements, so a valid program may very well contain faults. For an example, refer to Entry 68 in Tab. 3-1. Here, no semantic constraints of the language are violated, but still the result is wrong for all but one case.
Tab. 3-2 contains examples for the second part of the proposition, most notably Entry 3. The language rules here are more strict than required for the situation, and thus any tool basing its decision on these language rules would report a fault here.
On inspection, however, a software engineer would probably conclude that the situation is not critical and insert an explicit cast of the pointer passed as argument to silence the compiler and/or verification tool, without changing the observable behaviour of the code.
Further, while in theory language standards should provide well-defined semantics for a language, in practice, this is seldomly the case, if at all.
For example, the C Language Standard (ISO/IEC 9899[1]) distinguishes constraints from semantics, whereas constraints are statically verifiable properties, e.g. correspondence of types. Semantics mostly define the behaviour of the relevant language elements, such as expressions or statements, in terms of the state of the program during execution.
Very often, the actual result, however, explicitly remains undefined. In some cases, this is because no useful definition can be given, such as in an array subscript expression, where the index given points beyond the last element of the array.

In other cases, the lack of a definition is not as well-founded, such as in the case of converting an integer value to a type in which the value cannot be represented:

- When converting to a signed integer type, the result is undefined.
- When converting to an unsigned integer type, a definition is given in terms of modulo arithmetic, by repeatedly adding or subtracting the maximum representable value plus one until the result fits into the target type.

In principle, a very similar definition could have been used for conversion to a signed integer type, with the added benefit of a more consistent language concept holding less surprises for a user of the language.
Of course, compilers generate code for both operations which also provide a well-defined result, albeit that definition is now implementation-dependent instead of being standardised. Still, this implementation-based definition seems to be so widespread that programmers – not having read and understood the standard in detail – may even expect this to be the standard definition. Therefore this de-facto definition may very well be consistent with the software requirements, even though formally, the language standard declares the result to be undefined.
These examples show that even on the basic level of language semantics, it is difficult to systematically derive fault classifications. The issue is even intensified by the fact that not all language standards are structured in a systematic manner. Definitions applying to a single operation may be widely dispersed over the standards document, so that the process of deriving the conditions for a fault as seen from the point of view of the standard may be very error-prone by itself and subject to omissions.

4.2 Defect Types and Criticality Levels

The contents of the following tables Tab. 4-1, Tab. 4-2 and Tab. 4-3 is based on assessments of tool reports and their harmonisation, introducing common terms for the reports considered as relevant.
The text characterizing a defect issued by the tools differed considerably for the same defect type. Therefore a common set of defect types is defined. All relevant tool reports can be mapped uniquely onto this set. In future these defect types may replace the scheme (fault main type / fault subtype) as applied in Ch. 3.
For the fault analysis the focus was put on fault types which are considered as relevant for the type of software analyzed: embedded space software. 20 such defect types were identified.
To each such defect type a criticality level is assigned as shown in Tab. 4-1.

Criticality Level	**Comment**	**Examples**
Critical	The defect type does impact the correctness of system operations if being activated, i.e. it manifests to an error or a failure.	Division by zero
		NULL-ptr dererefence
		index out-of-bounds

Warning	The defect type highlights a possibly unintended operation in the source code which may, but not necessarily does manifest as a critical defect.	*Invariant condition* #define var 1 if (var==0)

		Invariant expression
		int ii=0;
		… // no modification
		// of ii
		ii+=1;

		Unused result / return value
		ret=myFunc();
		if (var<0) printf("ERROR in"
		"myFunc\n");

Uncritical	The defect type is neither critical nor can it be considered a warning.	violation of a lexical or layout rule or naming convention

Tab. 4-1: Critcality Levels

Level "critical" means that the defect compromises the system state in any case. Similarly, level "uncritical" means that such a defect does not compromise the system state in any case.
Level "warning" indicates that the defect of this type may hint at a critical defect. However, the effect may just as well be intended or a simple oversight without leading to an error.
The example on "return value not used" in Tab. 4-1 manifests as a defect which is of level "critical" because error handling cannot be activated due to the unintended mix of *ret* and *var*, while in other cases the return value may be dropped without any impact on the system state, then being classified as "uncritical".
Instead, the example on "redundant operation" is of type "uncritical" because the system state is not affected at all in this case. Here the report just indicates that the code could be optimized. In other cases code may be – unintentionally –missing which takes and modifies the value of `ii` before it is set again. In this case the level would be elevated to "critical".
The defect types are not necessarily orthogonal. The "non-terminating loop" as a result of static analysis is also covered by "timeout" which is a defect type from dynamic analysis, but covering more than defects of type "infinite loops".

.

Tab. 4-2 shows the common set of defect types together with the assigned criticality level.

Defect Type	Criticality Level
Array Index Out-of-Bounds	Critical
Dereference of Invalid Pointer	Critical
Dereference of NULL-Pointer	Critical
File Access Error	Critical
Invalid function pointer	Critical
Non-terminating Loop	Critical
Passing invalid argument to standard library routine	Critical
(Possible) Recursion	Critical
Resource Leak	Critical
Undefined Result of Arithmetic Operation	Critical
Uninitialized Variable	Critical
Arithmetic Operation on NULL Pointer	Warning
Invariant Condition	Warning
Invariant Expression	Warning
Parameter Type Mismatch in Function Call	Warning
Timeout during execution	Warning
Unnecessary loop construct	Warning
Unreachable Code	Warning
Unused Result	Warning
Multiple return paths	Uncritical

Tab. 4-2:Common Set of Defect Types

.

In Tab. 4-3 a description is given for defect types for which their meaning cannot be directly understood from its name.

Defect Type	Description	Example

Undefined Result of Arithmetic Operation	An arithmetic operation is performed the result of which is undefined as per the language standard. For example, the result of a conversion from a signed integer to an unsigned integer type is undefined if the signed value does not fit into the unsigned type.	`unsigned int func(int a, int b) {`
		`  return (unsigned int)(a+b);`
		`}`

Arithmetic Operation on NULL Pointer	A pointer is used for address calculation and its value is NULL.	`ptr=NULL; ind=5;`
		`myfunc(&ptr[ind]);`

Invariant Condition	A condition in a logical expression is invariant.	`if ( 1== 0)`
		`if (a == 0 || a != 0)`

Invariant Expression	An expression always yields the same result.	`ii=0;`
		`i+=1; // could be expressed by`
		`ii=1;`

Unnecessary loop construct	A loop can be executed once only.	`#define LIMIT 1`
		`...`
		`unsigned int i;`
		`for (i=0;i<LIMIT;i++) {`
		`  ...`
		`}`

Multiple return paths	There is more than one return-statement within a function. This is considered bad practice in structured programming because it complicates semantic analysis of the function itself and any code using it.	`if (a>=0) return a;`
		`else return –a;`

Tab. 4-3: Description of Some Defect Types

5. Abbreviations and Acronyms

Acronym	Description
BSSE	Dr. Rainer Gerlich BSSE System and Software Engineering Immenstaad
DLR	Deutsches Zentrum für Luft- und Raumfahrt
ECSS	European Cooperation for Space Standardization
OBSW	On-Board Software
SW	Software
tbc	to be confirmed
tbd	to be defined