A Rationale-Based Classification of MISRA C Guidelines

Roberto Bagnara*
University of Parma
Parma, Italy
Email: roberto.bagnara@unipr.it

Abramo Bagnara
BUGSENG
Parma, Italy
Email: abramo.bagnara@bugseng.com

Patricia M. Hill*
BUGSENG
Parma, Italy
Email: patricia.hill@bugseng.com

***Abstract*—MISRA C is the most authoritative language subset for the C programming language that is a de facto standard in several industry sectors where safety and security are of paramount importance. While MISRA C is currently encoded in 175 guidelines (coding rules and directives), it does not coincide with them: proper adoption of MISRA C requires embracing its preventive approach (as opposed to the "bug finding" approach) and a documented development process where justifiable non-compliances are authorized and recorded as *deviations*. MISRA C guidelines are classified along several axes in the official MISRA documents. In this paper, we add to these an orthogonal classification that associates guidelines with their main rationale. The advantages of this new classification are illustrated for different kinds of projects, including those not (yet) having MISRA compliance among their objectives.**

I. Introduction

The C programming language is the most used language for the development of embedded systems, including those implementing critical functionality of various kinds. There are strong economic reasons for this success: the possibility of writing concise code, high efficiency, easy access to hardware features, ISO standardization, the availability of tools such as C compilers, a long history of usage.

Unfortunately, the same reasons that are behind C's strong points are also the source of serious weaknesses: hundreds of behaviors are not fully defined by the language, which is also easy to misunderstand and open to all sorts of abuses [1], [2].

The solution adopted by industry to mitigate this problem is very pragmatic: *language subsetting*. Namely, the most important functional safety standards[1] either mandate or strongly recommend that critical applications programmed in C only use a restricted subset of the language such that the potential of committing possibly dangerous mistakes is reduced.

MISRA C is the most authoritative and most widespread subset for the C programming language. It is defined by a set of *guidelines* that enable the use of C for the development of safety- and/or security-related software as well as the development of applications with high integrity or high reliability requirements [8].

In this paper, we propose a classification of MISRA C guidelines that is complementary to the orthogonal classifications provided in the official MISRA C documents. This new classification encodes the *rationale* of the guideline, that is, the reason why the guideline is in MISRA C. The ability of quickly conveying the nature of the guideline rationale has several advantages:

- it reduces rule misunderstanding for those that do not take the time to read the full guideline specification;
- it helps better decision-making in the choice between compliance and deviation;
- it facilitates guideline prioritization for projects where MISRA compliance requirements arrived late in the development process;
- it facilitates guideline selection for projects that do no (yet) have MISRA compliance requirements.

The plan of the paper is as follows: Section II introduces MISRA C and the notion of MISRA compliance; Section III defines the classification; Section IV discusses the uses of the new classification in projects seeking MISRA compliance; Section V touches the subject of subsetting MISRA C guidelines; Section VI concludes.

II. MISRA C and MISRA Compliance

The first edition of MISRA C was published in 1998 [9] targeting the need —emerged in the automotive industry and reflected in 1994's MISRA *Development guidelines for vehicle based software* [10]— for "a restricted subset of a standardized structured language."

The first edition of MISRA C was very successful and it was adopted also outside the automotive sector. The wider industry use and the reported user experiences prompted a major reworking of MISRA C that resulted, in 2004, in the publication of its second edition.

In 2013, after 4 years of renewed effort, the third edition of MISRA C, MISRA C:2012, was published. MISRA C:2012 brought numerous improvements over the previous edition: coverage of language issues was extended; support for C99 [11] was added in addition to C90 [12]; mitigation for the lack of strong typing and corresponding type checking in

* Roberto Bagnara is a member of the *MISRA C Working Group* and of ISO/IEC JTC1/SC22/WG14, a.k.a. the *C Standardization Working Group*. Patricia M. Hill is a member of the *MISRA C++ Working Group*. Nonetheless, the views expressed in this paper are the authors' and should not be taken to represent the views of the mentioned working groups and organizations.

[1] IEC 61508 [3] (industrial, generic), ISO 26262 [4] (automotive), CENELEC EN 50128 [5] (railways), RTCA DO-178C [6] (aerospace) and FDA's *General Principles of Software Validation* [7] (medical devices).

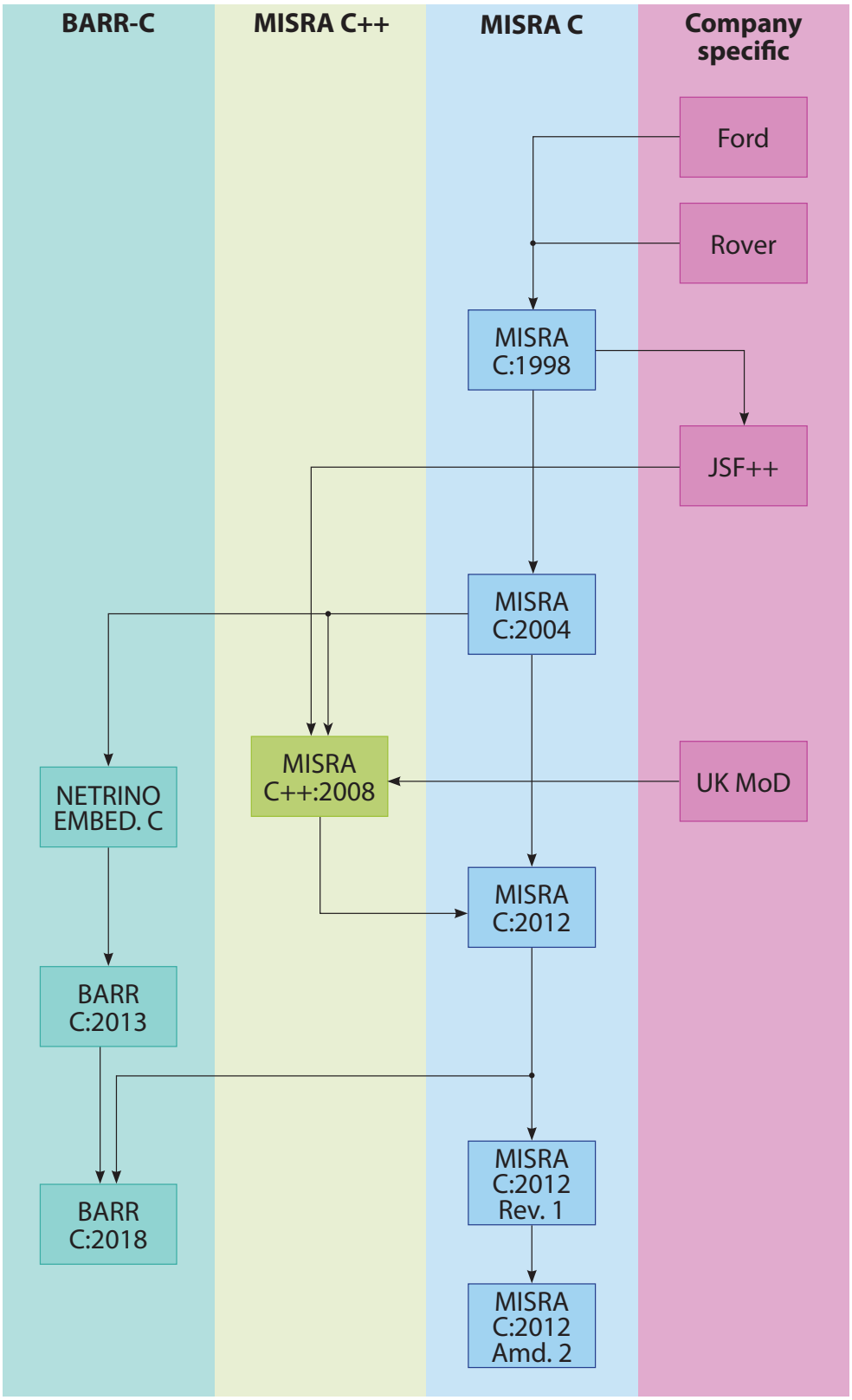

Fig. 1. Origin and history of MISRA C and BARR-C

C was considerably improved. In addition, the support for compliance was enhanced, the guidelines were specified more clearly and precisely, and, consequently, the likelihood that different static analysis tools would give the same results was significantly increased. Finally, MISRA C:2012 provides expanded rationales and the guidelines are more precisely targeted at real issues.

After the initial release of MISRA C:2012, the coding standard was further amended with the addition of 14 new guidelines [13] aimed at completing coverage against ISO/IEC TS 17961:2013 "C Secure" [14], [15] and increasing coverage against CERT C 2016 Edition [16], [17]. These amendments, along with *MISRA C:2012 Technical Corrigendum 1* [18] have been consolidated into the first revision of MISRA C:2012 [8]. A further amendment [19] extended the applicability of MISRA C to programs written according to the 2011 and 2018 standardized versions of the C language: in the sequel they are simply denoted by *C11* [20] and *C18* [21].

MISRA C, in its various versions, influenced all publicly-available coding standards for C and C++ that were developed after MISRA C:1998. Figure 1 shows part of the relationship and influence between the MISRA C/C++ guidelines and other sets of guidelines. The figure highlights the dependencies among MISRA C editions, revisions and amendments, and:

- Ford's and Land Rover's proprietary guidelines for the development of vehicle-based C software: these were merged into MISRA C:1998;
- Lockheed Martin's *JSF Air Vehicle C++ Coding Standards for the System Development and Demonstration Program* [22];
- BARR-C:2018 [23] and its ancestors: BARR-C:2013 [24] and the *Netrino's Embedded C Coding Standard* [25];
- MISRA C++:2008, the C++ counterpart to MISRA C, currently under revision [26];
- the list of *C++ vulnerabilities* that the UK Ministry of Defence contributed to MISRA C++:2008 [26, Appendix B].[2]

Many other items and relations are missing from the figure: MISRA C deeply influenced NASA's "JPL Institutional Coding Standard for the C Programming Language" [27] and several other coding standards (see, e.g., [16], [28]).

In the sequel *MISRA C* will denote *MISRA C:2012 Revision 1* [8] with *Amendment 2* [19].

Each of the 175 guidelines of MISRA C is classified as being either a *directive* or a *rule*:

Rule: a guideline such that information concerning compliance is fully contained in the source code *and* in the language implementation.

Directive: a guideline such that information concerning compliance is not fully contained in the source code and language implementation: requirements, specifications, designs and other considerations may need to be taken into account.

One of the things that is often misunderstood is that MISRA C is much more than just the set of its guidelines. The guidelines are meant to be used in the framework of a documented software development process. The official document detailing what must be covered when making a claim of MISRA compliance is MISRA Compliance:2020 [29], which also places constraints on such a development process.

The *deviation process* is an essential part of the adoption of the MISRA Guidelines, each one of which is assigned a single category: *mandatory*, *required* or *advisory*, defined as follows.

Mandatory: C code that complies with MISRA C must comply with every mandatory guideline: deviation is not permitted.

Required: C code that complies with MISRA C shall comply with every required guideline: a formal deviation is required where this is not the case.

Advisory: these are recommendations that should be followed as far as it is reasonably practical: formal deviation is not required, but non-compliances should be documented.[3]

Whenever complying with a guideline goes against code quality or does not allow access to the hardware or does not allow integrating or use suitably qualified adopted code, the guideline has to be *deviated*. That is, instead of modifying the code to bring it into compliance, for *required* guidelines, a written argument has to be provided to justify the violation whereas, for *advisory* guidelines, this is not necessary.

III. A New, Rationale-Based Classification of MISRA C Guidelines

MISRA C guidelines are classified along different axes [8]:

1) The series to which the guidelines belong, that is, the first number in the numerical part of the guideline identifier. For instance, Rule 8.1 belongs to rule series 8. Series are in one-to-one correspondence with the *main topic* of the guideline, and each one of them has a corresponding section in MISRA C, where the section title is remindful of the topic. For instance, rule series 7, 8 and 9 are contained in the following sections of [8]:
   8.7 Literals and constants
   8.8 Declarations and definitions
   8.9 Initialization
2) The guideline being a *rule* or a *directive*. This is indicated by the guideline name, which always begins with 'Rule' or 'Dir', respectively.
3) The guideline category: *mandatory*, *required* or *advisory*. In this paper, identifiers for mandatory MISRA C:2012 guidelines are set in boldface (e.g., **Rule 9.1**) whereas

[2]This is the equivalent of Annex J listing the various behaviors in ISO C, which is missing in ISO C++, making it hard work to identify them and to ensure they are covered by the guidelines.

[3]Advisory guidelines can also be *disapplied* upon certain conditions, but we prefer not to dwell on these distinctions here. The interested reader is referred to MISRA Compliance:2020 [29] for all the details.

identifiers for advisory guidelines are set in italics (e.g., *Dir 4.6*).

4) The guideline being *decidable* or *undecidable* according to whether answering the question about compliance can be done algorithmically.
5) The guideline scope being *single translation unit* or *system* according to the amount of code that needs to be analyzed in order to check compliance.

All these classifications are defined in MISRA C. While the last four axes have profound consequences on the notion of *MISRA compliance* and on the activities that tools and humans may or may not have to perform, the first axis is just for presentation convenience. Moreover, there are guidelines that would legitimately belong to more than one topic, and only the main one is captured in the series.[4]

Here we propose a further classification of MISRA guidelines. This classification encodes the main *rationale* for each guideline, that is, the nature of the addressed language and programming issues.[5] According to the new classification each guideline belongs to one or two of 16 named sets. Only three guidelines belong to two sets, that is, Rule 6.1, Rule 8.6 and **Rule 22.5**.

A synopsis of the new classification is presented in Figure 2. The following sections illustrate each named set.

A. LTLM: Language/Toolchain/Library Misuse

This set contains 41 guidelines: Dir 4.11, *Dir 4.13*, Rule 1.1, Rule 1.3, Rule 5.1, Rule 5.2, Rule 5.4, Rule 6.1, Rule 8.6, Rule 8.10, **Rule 9.1**, Rule 9.4, Rule 13.1, Rule 13.2, **Rule 17.4**, *Rule 17.5*, Rule 18.1, Rule 18.2, Rule 18.3, Rule 18.6, **Rule 19.1**, Rule 20.2, Rule 20.3, Rule 20.4, Rule 20.6, Rule 20.11, Rule 20.13, Rule 20.14, Rule 21.1, Rule 21.2, **Rule 21.13**, Rule 21.14, **Rule 21.17**, **Rule 21.18**, **Rule 21.19**, **Rule 21.20**, **Rule 22.2**, **Rule 22.4**, **Rule 22.6**, Rule 22.8, Rule 22.10.

These guidelines are extremely important: they have to do with the "contract" between those programming and reasoning about programs in C[6] and the artifacts that make this possible. They are:

- The language specification: if the program violates the C syntax and constraints, or if it has undefined[7] or critical unspecified behavior,[8] then the program does not have a defined meaning.
- The toolchain specification: if the program exceeds one or more of the translation limits[9] of the language implementation (which, usually, is the combination of compiler, assembler, linker and librarian) then, again, the program does not have a defined meaning.
- The library specification: if the program violates the preconditions of library functions and macros, or it fiddles with reserved identifiers, then, once more, the program does not have a defined meaning.

A very important guideline in this set is

> Rule 1.1: The program shall contain no violations of the standard C syntax and *constraints*, and shall not exceed the implementation's translation limits.

While this rule is not mandatory, it is one of the cornerstones of MISRA C: if the program does not satisfy the C language specification (modulo allowed extensions), nothing can really be said about the program, let alone about its MISRA compliance. Even when MISRA compliance is not among the objectives, violations point at important circumstances that otherwise would completely escape the developers, such as:

- The mismatch between the language standard and the features used in the program. For instance, the use of a C11-specific feature like `_Static_assert` in code compiled in C99 mode: perhaps the code should have been compiled in C11 or C18 mode, or perhaps `_Static_assert` was not meant to be used.
- The use of documented language extensions, such as GCC's statement expressions.[10] Once they are accepted as extensions, they will fall under the scope of *Rule 1.2*, which is covered later in this paper.
- The use of constructs that violate the C standard syntax or constraints, for which the toolchain does not issue a diagnostic message and does not document as a supported extension. This is the case, e.g., for the return of void expressions, undocumented but accepted without warning by GCC up to and including version 11.2. Of course, the lack of documentation makes it hard to legitimately consider such constructs as language extensions.

Another important rule in the LTLM set is:

> **Rule 9.1**: The value of an object with automatic storage duration shall not be read before it has been set.

Reading an uninitialized object residing on the stack is undefined behavior, so anything can happen.
In the most typical case, memory or register values that, due to the previous computation history, happen to be there,

[4]Note that with our choice of fonts for the category, every guideline name presented here will indicate its classifications with respect to the first three axes.

[5]In MISRA C:2012 this is explained in a section titled "Rationale" for each guideline [8], [19].

[6]As opposed to programming and reasoning about programs in assembly language.

[7]This is "behavior, upon use of a nonportable or erroneous program construct or of erroneous data, for which [the C standard] imposes no requirements" [21].

[8]Unspecified behavior is "behavior, that results from the use of an unspecified value, or other behavior upon which [the C standard] provides two or more possibilities and imposes no further requirements on which is chosen in any instance" [21]. In MISRA C an unspecified behavior is called *critical* "according to whether reliance on this behaviour is likely to lead to unexpected program operation" [8, Appendix H.2].

[9]These are minimal quantities that conforming implementations have to meet or exceed. For instance, C90 mandates a minimum of at least 8 nesting levels of `#include`d files, 6 significant initial characters in an external identifier, 257 `case` labels for a `switch` statement [12, Subclause 5.2.4.1]. While more recent versions of the ISO C standard have more generous limits and many C implementations go beyond those limits, it must be noted that an implementation needs not issue a diagnostic message when a translation limit is exceeded.

[10]See https://gcc.gnu.org/onlinedocs/gcc/Statement-Exprs.html.

CTIS		
Dir 2.1	Dir 4.10	

RTIS	
Dir 4.1	Rule 22.1

EAPI
Rule 1.4

BAPI	
Rule 21.7	Rule 21.11

DOCU		
Dir 1.1	Dir 3.1	*Dir 4.2*

TNTI
Dir 4.14

ENMO		
Dir 4.3	Dir 4.8	*Rule 8.7*
Rule 8.9		

PORT		
Dir 4.6	*Rule 1.2*	Rule 6.1
Rule 22.5		

RTIL		
Dir 4.7	Rule 15.7	Rule 16.4
Rule 17.7	Rule 22.7	Rule 22.9

IRRC		
Rule 2.1	Rule 2.2	*Rule 2.3*
Rule 2.4	*Rule 2.5*	*Rule 2.6*
Rule 2.7	Rule 8.6	

CSTR		
Rule 13.5	Rule 14.2	*Rule 15.1*
Rule 15.2	Rule 15.3	*Rule 15.4*
Rule 15.5	Rule 16.1	Rule 16.2
Rule 16.5	Rule 16.6	*Rule 18.4*

TYPM		
Rule 10.1	Rule 10.2	Rule 10.3
Rule 10.4	*Rule 10.5*	Rule 10.6
Rule 10.7	Rule 10.8	Rule 12.2
Rule 14.4	Rule 16.7	Rule 21.15

CTIL		
Rule 7.4	Rule 8.2	Rule 8.4
Rule 8.13	Rule 11.1	Rule 11.2
Rule 11.3	*Rule 11.4*	*Rule 11.5*
Rule 11.6	Rule 11.7	Rule 11.8
Rule 11.9	**Rule 17.3**	

HTDR		
Dir 4.12	Rule 8.14	*Rule 12.3*
Rule 14.1	Rule 16.3	Rule 17.1
Rule 17.2	**Rule 17.6**	*Rule 18.5*
Rule 18.7	Rule 18.8	*Rule 19.2*
Rule 20.1	*Rule 20.5*	*Rule 20.10*
Rule 20.12	Rule 21.3	Rule 21.4
Rule 21.5	Rule 21.6	Rule 21.8
Rule 21.9	Rule 21.10	*Rule 21.12*
Rule 21.16	Rule 21.21	Rule 22.3
Rule 22.5		

DEVM		
Dir 4.4	*Dir 4.5*	Dir 4.9
Rule 3.1	Rule 3.2	Rule 4.1
Rule 4.2	Rule 5.3	Rule 5.5
Rule 5.6	Rule 5.7	Rule 5.8
Rule 5.9	Rule 6.2	Rule 7.1
Rule 7.2	Rule 7.3	Rule 8.1
Rule 8.3	Rule 8.5	Rule 8.8
Rule 8.11	Rule 8.12	Rule 9.2
Rule 9.3	Rule 9.5	*Rule 12.1*
Rule 12.4	**Rule 12.5**	*Rule 13.3*
Rule 13.4	**Rule 13.6**	Rule 14.3
Rule 15.6	*Rule 17.8*	Rule 20.7
Rule 20.8	Rule 20.9	

LTLM		
Dir 4.11	*Dir 4.13*	Rule 1.1
Rule 1.3	Rule 5.1	Rule 5.2
Rule 5.4	Rule 6.1	Rule 8.6
Rule 8.10	**Rule 9.1**	Rule 9.4
Rule 13.1	Rule 13.2	**Rule 17.4**
Rule 17.5	Rule 18.1	Rule 18.2
Rule 18.3	Rule 18.6	**Rule 19.1**
Rule 20.2	Rule 20.3	Rule 20.4
Rule 20.6	Rule 20.11	Rule 20.13
Rule 20.14	Rule 21.1	Rule 21.2
Rule 21.13	Rule 21.14	**Rule 21.17**
Rule 21.18	**Rule 21.19**	**Rule 21.20**
Rule 22.2	**Rule 22.4**	**Rule 22.6**
Rule 22.8	Rule 22.10	

Fig. 2. Synopsis of the rationale-based classification of MISRA C guidelines

will be read.[11] The production of unexpected program results is of course unacceptable in critical applications, but there is something worse: code violating this rule is vulnerable to exploits. Exploits can be of two kinds: (1) confidential information is leaked to unauthorized parties; (2) attackers are given ways to control the system. For the first kind, consider the following example, which is a simplified version of a real bug [30]:

```
int get_hw_address(struct device *dev,                    1
                   struct user *usr) {                    2
  unsigned char addr[MAX_ADDR_LEN];                       3
  if (!dev->has_address)                                  4
    return -EOPNOTSUPP;                                   5
  dev->get_hw_address(addr);                              6
  return copy_out(usr, addr, sizeof(addr));               7
}                                                         8
```

Automatic array variable `addr` declared in line 3 may only be partly initialized by the indirect function call at line 6, but in any case the entire contents of the array is copied from kernel space to user space and that may constitute an important brick to construct a more complex, reliable exploit. For the second kind of exploit, consider the following example [30]:

```
int queue_manage() {                                      1
  struct async_request *backlog;                          2
                                                          3
  if (engine->state == IDLE)                              4
    backlog = get_backlog(&engine->queue);                5
                                                          6
  if (backlog)                                            7
    backlog->complete(backlog, -EINPROGRESS);             8
                                                          9
  return 0;                                               10
}                                                         11
```

Here, the automatic pointer variable `backlog` is uninitialized at the point of declaration in line 2. The guard of the `if` statement at line 4 is usually true, so `backlog` is usually initialized at line 5 and testing is unlikely to find the issue. However, when the statement at line 5 is not executed, an attacker who can control the value of `backlog` can cause the application to call any function that the attacker stores at the address in `backlog->complete`.

A common form of language/toolchain/library misuse is captured by the following guidelines:

> Rule 21.1: `#define` and `#undef` shall not be used on a reserved identifier or reserved macro name.
>
> Rule 21.2: A reserved identifier or reserved macro name shall not be declared.

It is as if someone, in the distant past, thought that it was a good idea to define macros and other identifiers beginning with a leading underscore in user code. Maybe that person looked at standard library header files and concluded that doing so was good style, and this false belief was somehow propagated and perpetuated. Whatever the reason is, it is a matter of fact that projects that have not previously undergone MISRA compliance checking tend to have hundreds or thousands of violations of these guidelines. This is a potential source of undefined behavior and, as such, something to definitely avoid.

[11] Beware, this is not the only case. E.g., compilers can exploit undefined behavior in ways that may completely baffle programmers.

B. DEVM: Developer Misreading/Mistyping/Misunderstanding

This set contains 38 guidelines: *Dir 4.4*, *Dir 4.5*, Dir 4.9, Rule 3.1, Rule 3.2, Rule 4.1, *Rule 4.2*, Rule 5.3, Rule 5.5, Rule 5.6, Rule 5.7, Rule 5.8, *Rule 5.9*, Rule 6.2, Rule 7.1, Rule 7.2, Rule 7.3, Rule 8.1, Rule 8.3, Rule 8.5, Rule 8.8, *Rule 8.11*, Rule 8.12, Rule 9.2, Rule 9.3, Rule 9.5, *Rule 12.1*, *Rule 12.4*, **Rule 12.5**, *Rule 13.3*, *Rule 13.4*, **Rule 13.6**, Rule 14.3, Rule 15.6, *Rule 17.8*, Rule 20.7, Rule 20.8, Rule 20.9.

Guidelines in this set have to do with the prevention of mistakes arising from three phenomena:

1) developers not being aware of all the C language intricacies;
2) developers misreading or misinterpreting program elements;
3) developers typing something different from what was intended.

The first two phenomena are important (but not the only) sources of what MISRA C refers to as *developer confusion*.

An example guideline in this set is

> **Rule 12.5**: The `sizeof` operator shall not have an operand which is a function parameter declared as "array of type".

It is likely that the developer is not aware that the actual parameter type is a pointer instead of the written array. This guideline prevents potentially catastrophic consequences of this possibility.

C. CTIS: Compile-Time Issues

This set contains 2 guidelines: Dir 2.1, Dir 4.10.

These directives guard against issues that may occur at compile time whereby executable code does not match the source code that was meant to produce it.

The first directive is actually a constraint on the interface between the toolchain and the build procedure:

> Dir 2.1: All source files shall compile without any compilation errors.

A build procedure that is unable to reliably detect compilation errors may link old or invalid object code into the application executable.

The second directive prevents including a header file multiple times:

> Dir 4.10: Precautions shall be taken in order to prevent the contents of a *header file* being included more than once.

The systematic use of guards against multiple inclusion is required for three reasons:

1) To avoid circular inclusion chains, which may give rise to compilation errors that are difficult to diagnose and

understand, as well as undefined behavior (compilers are not required to detect circular inclusion chains).
2) To avoid mistakes in the ordering of header file inclusions. Without guards, programmers have to use a strict inclusion discipline in order to avoid compilation errors. Such a discipline that has been popular in the past is "no header file can include another header file." However, this results in non-header source files beginning with long lists of inclusions *that have to be specified in the right order*; getting the order wrong might result in a compilation error or, what is more important, unexpected behavior.
3) To avoid unnecessary long compilation time and static analysis time, the latter with the danger that a complexity-throttling measure in the static analyzer might lead to analysis imprecision.

D. RTIS: Run-Time Issues

This set contains 2 guidelines: Dir 4.1, Rule 22.1.

These guidelines cover run-time issues that are not covered by the rules in other sets.

> Dir 4.1: Run-time failures shall be minimized.

This directive requires covering run-time failure by any set of measures: prevention, testing, run-time detection, static analysis and more. The adopted measures should of course be documented to be amenable to peer review.

> Rule 22.1: All resources obtained dynamically by means of Standard Library functions shall be explicitly released.

Program resources (e.g., files, memory, synchronization devices, etc.) are all in finite supply: failure to release them, a.k.a. *resource leakage*, will, sooner or later, lead to their exhaustion and the consequent potential program misbehavior.

E. DOCU: Documentation

This set contains 3 guidelines: Dir 1.1, Dir 3.1, *Dir 4.2*.

The documentation to be provided concerns:

- implementation-defined behaviors affecting the program semantics (Dir 1.1);
- traceability information to requirements (Dir 3.1);
- assembly code (*Dir 4.2*).

Given the uncertainty about the requirements to be implemented, about the implementation-defined behaviors of the selected toolchain, and about the meaning of assembly code fragments, procrastinating compliance with these guidelines is not advisable.

F. ENMO: Encapsulation/Modularization

This set contains 4 guidelines: Dir 4.3, Dir 4.8, *Rule 8.7*, *Rule 8.9*.

Encapsulation of programming procedures by functions and macros makes a program easier to read and modify in the future. Modularization divides a function into an interface (a header file) and an implementation (the source file). Users of the function can only have access to the header file, and the header file contents should be minimized.

Consider the following directive:

> Dir 4.3: Assembly language shall be encapsulated and isolated.

By isolating the assembly language in functions or macros, the readability and maintainability of such code is improved. Moreover, this encapsulation ensures that the code can more easily be later replaced by alternative code with an equivalent implementation.

> Dir 4.8: If a pointer to a structure or union is never dereferenced within a translation unit, then the implementation of the object should be hidden.

When header files contain too much detail, chances are that such detail is depended upon in unwanted ways. This directive prescribes the use of *opaque pointers* to hide the implementation details of data structures from user code that does not need to interfere with them.

G. HTDR: Hard To Do Right

This set contains 28 guidelines: Dir 4.12, Rule 8.14, *Rule 12.3*, Rule 14.1, Rule 16.3, Rule 17.1, Rule 17.2, **Rule 17.6**, *Rule 18.5*, Rule 18.7, Rule 18.8, *Rule 19.2*, *Rule 20.1*, *Rule 20.5*, *Rule 20.10*, Rule 20.12, Rule 21.3, Rule 21.4, Rule 21.5, Rule 21.6, Rule 21.8, Rule 21.9, Rule 21.10, *Rule 21.12*, Rule 21.16, Rule 21.21, Rule 22.3, **Rule 22.5**.

These guidelines warn against the use of harmful, but possibly very useful language features that require considerable expertise if they have to be used in a critical system.

As an example, let us consider the required directive

> Dir 4.12: Dynamic memory allocation shall not be used.

There is an increasing demand to allow dynamic memory allocation, but insufficient understanding about the consequences of using this technique. Since Dir 4.12 is a required directive, it can be deviated with a proper motivation. However, formulating such a motivation in a defensible way, requires dealing with a number of issues, summarized below, that are not easy to solve. Note that such issues have to be dealt with anyway, even when not following MISRA C.

1) Out-of-storage Run-Time Failures: It is often not easy to ensure that the available heap memory will always be sufficient for the chosen allocation strategy. There are various possibilities:

a. Available memory is demonstrably sufficient even if allocated memory is never deallocated. This is a very favorable situation: we can avoid releasing memory and, doing so, we need not be concerned with memory allocation errors.
b. Available memory is not sufficient if we do not deallocate. Then we will have to release memory when it is no longer used. There are two options:

b1. Explicit release: then we are confronted with dangling pointers (when we release too early), memory leaks (when we release too late or fail to release), double-free errors (when we release more than once). The use

of smart pointers (preferably assisted by the use of automatic checkers to ensure they are used properly) can prevent all such errors; of course, using smart pointers comes with some space and time overhead that may or may not be appropriate depending on the application.

b2. Automatic release, i.e., via the use of a garbage collector: we avoid dangling pointers, memory leaks and double-free errors, but the execution time of our code becomes more difficult to predict (the so-called real-time, incremental garbage collectors do not completely solve the "stop-the-world effect").

In both cases, proving that the space is sufficient is very often difficult. One of the reasons is fragmentation, which is a particularly important problem for long-running systems (systems that will run continuously for many hours, days or even weeks): depending on the distribution of memory allocation sizes, the free memory can be filled by very small fragments that will have to be compacted (this is not always possible and may cause "stop-the-world effect") or they might cause an out-of-storage condition.

2) Timing Considerations: Memory allocation and deallocation functions are often subject to variable (and possibly long) latency: free lists or similar data-structures have to be searched and maintained, free blocks have to be searched, merged, splitted and possibly moved. This is unacceptable for systems that have to meet hard real-time constraints.

In order to better understand what is the best strategy for a project, one needs to answer the following questions (in no particular order):

- Do we have hard real-time constraints?
- Do we have tight limits on the available computing power or can we trade some computing power in exchange for correctness guarantees?
- Is there a simple pattern in the memory sizes we need to dynamically allocate? (There are relatively easy, partial solutions for the fragmentation problem if we only need blocks of, say, a dozen fixed dimensions.)
- Is our system long-running or can we assume it will be rebooted at least once in a given number of hours or days?
- Is there a demonstrable upper bound to the amount of memory that has to be dynamically allocated? Is this upper bound known at system-startup?

The rationale of the last question is the following: if dynamic memory allocation could take place at system-startup only, possibly depending on some configuration parameters, then the check for memory sufficiency is done during startup together with other startup self-test activities. This is the recommendation given by the *JSF Air Vehicle C++ Coding Standards for the System Development and Demonstration Program* [22], with the advantage that no dynamic memory allocation errors can occur once the system is operating in a safety-critical mode.

While Dir 4.12 is concerned with programming technique in general, however it is implemented, another rule in the HTDR set targets the typical implementations provided as part of the C standard library:

> Rule 21.3: The memory allocation and deallocation functions of `<stdlib.h>` shall not be used.

In addition to the issues mentioned above, such implementations, being general-purpose and not accompanied by solid specifications, have unknown fragmentation and timing properties.

A final example of a rule cautioning against a program feature that may be both exceedingly useful (for some applications) and exceedingly complex to get right is the following:

> Rule 17.2: Functions shall not call themselves, either directly or indirectly.

The possibility of writing (mutually) recursive functions greatly simplifies the task of writing code operating on non-trivial data structures defined by means of recursive rules (such as languages defined by context-free grammars). This, however, is not a frequent requirement for critical systems. In contrast, the presence of (mutually) recursive functions considerably complicates the task of bounding the maximum stack size needed to run the code. As is well known, stack overflow is a serious and potentially catastrophic issue (see, e.g., [31]).

H. EAPI: Emergent APIs

This set contains just 1 guideline: Rule 1.4.

> Rule 1.4: Emergent language features shall not be used.

This is the counterpart to Rule 1.1 that allows using language implementations conforming to C11 [20] and C18 [21], except for the features that are not yet covered by MISRA C, namely:

- type generic expressions: `_Generic`;
- support for no-return functions: `_Noreturn`, `<stdnoreturn.h>`;
- support for multiple threads of execution: `_Atomic`, `<stdatomic.h>`, `_Thread_local`, `<threads.h>`;
- support for the alignment of objects: `_Alignas`, `_Alignof`, `<stdalign.h>`;
- bounds-checking interfaces of C11/C18 Annex K.

I. BAPI: Badly-designed/Obsolete APIs

This set contains 2 guidelines: Rule 21.7, Rule 21.11.

Standard library functions that have restricted input values have a variety of ways to inform the caller that an argument's value is wrong. These include returning a value such as `-1` or setting `errno` to an error value. However, this does not apply to all library functions. Consider the rule

> Rule 21.7: The Standard Library functions `atof`, `atoi`, `atol` and `atoll` of `<stdlib.h>` shall not be used.

These functions convert the initial portion of the string pointed to by its parameter to an arithmetic type. However, when the result cannot be represented, the behavior is undefined. As these functions have been superseded by `strtod`, `strtof`,

`strtold` `strtol` and `strtoll`, they can be considered both badly designed and obsolete.

Obsolescence is not the reason behind rule

> Rule 21.11: The standard *header file* `<tgmath.h>` shall not be used.

Here the point is that the use of the type-generic features provided by `tgmath.h` can have undefined behavior.

J. PORT: Portability

This set contains 4 guidelines: *Dir 4.6*, *Rule 1.2*, Rule 6.1, **Rule 22.5**.

Guidelines in this set have portability as their main objective. Consequently, for a project that is not meant to be portable and where its developers have complete knowledge of all the implementation-defined behaviors, this set may not seem to be very relevant apart from one notable exception Consider the rule

> *Rule 1.2*: Language extensions should not be used.

A program that relies on extensions will be difficult to port to a different language implementation, and hence adherence to this rule will help in ensuring its portability. But this is not the only reason why we have this rule. In order to code and reason about programs in C we need to build confidence on the correctness of the used C compiler. Functional safety standards such as ISO 26262 [4] and RTCA DO-178C [6] prescribe *compiler qualification*. This is usually achieved by running *validation suites* comprising tens of thousands of C test programs through the compiler and checking the resulting outcome. However, available C validation suites test the standardized language, not the compiler extensions: this is another important reason why the use of extensions should be limited as much as possible.

Another rule in this set is

> Rule 6.1: Bit-fields shall only be declared with an appropriate type.

This covers three issues:

1) A bit-field type specified as unqualified `int` may be signed or unsigned depending on the implementation, hence, a portability issue.
2) Using bit-field types different from `unsigned int` or `signed int` is undefined behavior in C90: this is the reason why Rule 6.1 also occurs in the LTLM set.
3) For later versions of ISO C, a conforming compiler supports also `_Bool` as well as an implementation-defined set of integer types for bit-fields. The used type may influence the implementation-defined aspects of bit-field layout.

K. CTIL: Compile-Time Information Loss

This set contains 14 guidelines: Rule 7.4, Rule 8.2, Rule 8.4, *Rule 8.13*, Rule 11.1, Rule 11.2, Rule 11.3, *Rule 11.4*, *Rule 11.5*, Rule 11.6, Rule 11.7, Rule 11.8, Rule 11.9, **Rule 17.3**.

The objectives of the rules in this set include making sure the compiler is given all the information it needs to diagnose common and dangerous programming mistakes. A good example is

> Rule 7.4: A string literal shall not be *assigned* to an object unless the object's type is "pointer to `const`-qualified `char`".

This rule demands that the read-only nature of string literals is propagated using the `const` type qualifier, allowing the compiler to flag possible misuses.

L. RTIL: Run-Time Information Loss

This set contains 6 guidelines: Dir 4.7, Rule 15.7, Rule 16.4, Rule 17.7, Rule 22.7, Rule 22.9.

Inadvertently corrupting, ignoring or erasing a run-time result is very dangerous.

> Rule 16.4: Every `switch` statement shall have a `default` label.

Compliance with this rule ensures that a `switch` handles *all* cases, including, e.g., the values of an enumerated type not corresponding to any enumerator.

M. TNTI: Tainted Input

This set contains just 1 guideline: Dir 4.14.

Any input from external sources may be invalid due to accidental error or malicious intent. For instance:

- A value used to determine an array index could cause an array bounds error,
- A value used to control a loop could cause (almost) infinite iterations,
- A value used to compute a divisor may make it zero,
- A value used to determine an amount of dynamic memory may lead to excessive memory allocation,
- A string used as a query to an SQL database may contain a '`;`' character.

The rule

> Dir 4.14: The validity of values received from external sources shall be checked

requires that all such input is checked appropriately.

N. IRRC: Irrelevant Code

This set contains 6 guidelines: Rule 2.1, Rule 2.2, *Rule 2.3*, *Rule 2.4*, *Rule 2.5*, *Rule 2.6*, *Rule 2.7*, Rule 8.6.

The coincidence of this set with series 2 of MISRA C (*Unused code*) is almost complete. The exception is for rule

> Rule 8.6: An identifier with external linkage shall have exactly one external definition.

for the cases where it does not give rise to undefined behavior. It is for the other cases that this rule also occurs in the LTLM set of guidelines.

O. TYPM: Types Misuse

This set contains 12 guidelines: Rule 10.1, Rule 10.2, Rule 10.3, Rule 10.4, *Rule 10.5*, Rule 10.6, Rule 10.7, Rule 10.8, Rule 12.2, Rule 14.4, Rule 16.7, Rule 21.15.

The C language definition relies strongly on implicit type conversion. Rather frequently, such conversions do not correspond to the will of the programmer. Sometimes this mismatch is due to a language standard misinterpretation, sometimes it may be due to a typo. Consider the following rule:

> Rule 14.4: The controlling expression of an `if` statement and the controlling expression of an *iteration-statement* shall have *essentially Boolean* type.

This asks that the guards are effectively Boolean to ensure the programmer intention is unambiguously expressed and matches the true-or-false nature of controlling expressions.

P. CSTR: Code Structuring

This set contains 12 guidelines: Rule 13.5, Rule 14.2, *Rule 15.1*, Rule 15.2, Rule 15.3, *Rule 15.4*, *Rule 15.5*, Rule 16.1, Rule 16.2, Rule 16.5, Rule 16.6, *Rule 18.4*.

These guidelines define a subset of C that is well-structured so that it is easier to review, maintain and analyze. There are several rules that define a restricted format for the selection and iteration statements. As an example, consider the following rule:

> Rule 16.1: All `switch` statements shall be well-formed.

Depending on the value of its controlling expression, a `switch` statement jumps to the code following a matching `case` label or, if there is no match, to a `default` label if present, or, if not, to the next statement. Associated `break` statements will cause control to jump out of the `switch` statement to the next statement. However, the C syntax for the body of a `switch` is completely general and these labels can be mixed with the code in the body in an arbitrary way.

Rule 16.1 restricts the `switch` statement to be a simple coherent structure where the body is a compound statement and all associated `case` and `default` labels are direct children of this block. With consistent indentation of dependent code, it should be straightforward to see which `switch` the `case` and `default` labels are associated to, and hence make it easier to examine how the different cases are handled.

IV. Using the New Classification for MISRA Compliance

The new classification of MISRA C guidelines proposed in this paper, being rationale-based, has the advantage of reminding programmers of the guidelines' rationale. This, in turn:

1) allows them to quickly and more faithfully reconstruct a guideline's objectives and, consequently,
2) improve their ability in formulating a defensible choice in the *comply vs deviate* decision.

While formal training is one of the requirements for MISRA compliance [29, Section 7.1], programmers cannot generally be expected to have the details of MISRA guidelines fresh in their minds. Of course, for such programmers, the right thing to be done is to refresh knowledge of the guideline and, in particular, of its rationale. Unfortunately, not all programmers when in doubt will open the relevant MISRA documents.

Independently of this, the decision, whether to comply with a MISRA guideline or to deviate from it, is important. Therefore a quick reminder of the nature of the guideline rationale via our proposed new classification is helpful to both programmers and project managers to determine the subsequent steps.

Here is a non-exclusive set of questions for each rationale-based set. When confronted with a violation of a guideline, the programmer can check its rationale-based classification and, using the following list (possibly enhanced with further pertinent queries), consider their response to the associated questions:

LTLM: Do I really want to do that? (Hint: probably not.)
DEVM: Am I sure about what the C standard prescribes here? Did I make a typing mistake? Can I clarify my intentions?
CTIS: Do I trust my build procedure? Do I have multiple inclusion guards in place?
RTIS: Did I exclude or mitigate possible run-time errors?
DOCU: Is this documentation available? (Hint: if the answer is "no", then there is a problem.)
ENMO: Will we need to maintain this program? (Hint: probably yes; we are not prescient.)
HTDR: Did I consider the intricacies of this language feature?
EAPI: Do I really want to use this API? (Hint: probably not.)
BAPI: Do I really want to use this API? (Hint: definitely not.)
PORT: Is portability among the project objectives?
CTIL: Can I get this right without compiler assistance?
RTIL: Am I losing valuable information here?
TNTI: Is this input tainted? If so, can it cause harm?
IRRC: Why is this code here?
TYPM: Can this result into a run-time type error? Am I willing to run the risk?
CSTR: Can I make this code easier to understand?

V. Subsetting MISRA C Guidelines

Whereas the notion of MISRA compliance has degrees of flexibility, to go beyond what is prescribed by MISRA Compliance:2020 is non-negotiable. In particular, in MISRA C, there is no grading for the importance of guidelines other than the one implied by the *mandatory*/*required*/*advisory* categorization. That is, *mandatory* guidelines are the most important ones, then come *required*, then come *advisory* guidelines; all required guidelines, whether rules or directives are of equal importance, as are all mandatory guidelines, as are all advisory ones. A project cannot be MISRA compliant if some guidelines are completely neglected: all have to be given appropriate consideration.

So it is clear that subsetting the MISRA C Guidelines beyond what is allowed by MISRA Compliance:2020 does not allow MISRA compliance of the project. However, two things, expounded in the following sections, have to be taken into account.

A. No MISRA Compliance Requirements for Most Projects

The vast majority of projects and organizations do not (yet) have a MISRA C requirement. In addition, far too many projects in somewhat critical embedded systems, are at the stage "no coding standard, no static analysis". For these projects, in [32], we proposed considering compliance with (subsets of) the BARR-C:2018 coding standard [23].

Here we propose an alternative, namely, the (incremental) adoption of subsets of MISRA C. The rational-based classification allows the simplified identification of the guidelines that are more relevant to the project objectives. For instance, if portability is among the objectives, the guidelines dealing with portability are better included in the project coding standard.

One advantage of using subsets of MISRA C is the availability of good tools supporting it. For projects leaning on preventive measures, this is much better than resorting to bug finders, which are notoriously plagued with false negatives and will thus only discover parts of the issues.

A good strategy for a C project without MISRA compliance requirements could be to enforce LTLM guidelines first: they concern harmful situations that should always be avoided one way or the other. If time allows, the DEVM guidelines should be considered next: even though violations of these guidelines are possibly harmless, they are generally easily avoidable and solving them reduces the risk of misleading others. Then other rationale-based classification sets can be considered depending on the context: use PORT for portability, HTDR if inexperienced programmers are in the team, ENMO if maintainability and the possibility of reusing code for other projects is contemplated.

B. MISRA Compliance Is Often Undertaken Late

Even for projects that do have MISRA C requirements, it is a matter of fact that the work on such requirements often begins far too late in the development cycle. It is well known that the highest payoff from the adoption of MISRA C is achieved when the MISRA guidelines are systematically enforced from the start of the project or, at the very least, before the code development reaches the review and unit testing phases (since otherwise a lot of rework and retesting has to be expected). Nonetheless, many projects start late and find themselves confronted with a backlog of violations that has to be carefully managed to minimize project delays. In these cases, prioritization is an important tool to avoid submerging developers in unmanageable workloads, and our orthogonal classification may be of help in this regard. Most importantly, we believe the classification will help projects with a MISRA compliance objective not to procrastinate the MISRA effort entirely: it might be acceptable to postpone compliance with the guidelines dealing with unused/unreachable/dead code, but other guideline subsets need to be considered earlier.

A good prioritization strategy for such projects is to first make sure the documentation required to comply with the DOCU guidelines is available. Then enforce the LTLM guidelines: among other things, this will fix the used C dialect and will allow the compiler qualification efforts to proceed in parallel with further code development once the compilation options have been decided. The TYPM guidelines should be checked next, as delaying compliance with them typically increases the costs due to code reworking and refactoring. The DEVM and HTDR guidelines should not be delayed too much if the amount of code still to be written is substantial. These are just rules of thumb that the project manager can adapt to the specifics of the project and the team.

VI. Conclusion

In this paper, we provide a new, rationale-based classification of the MISRA C guidelines that is orthogonal to the classifications provided in the MISRA documents. By reminding programmers of the main rationales for violated guidelines, we believe they can more easily identify the right course of action in each case. In particular, the decision whether to comply with or deviate from a guideline is simplified by the recognition of the guideline rationale, as this influences both the nature of the correct remediation measures and the information to be provided in the deviation record.

A further advantage of the new classification is that it opens the door to projects without MISRA compliance requirements. This is the majority of C projects and too many of them are at the stage where no coding standard is adopted and no static analysis is performed. Some of them use bug finders to find some bugs but, as is well known, bug finders favor false negatives over false positives and are thus inadequate to implement preventive measures as in the spirit of the MISRA coding standards. To projects without MISRA compliance requirements but with a strong emphasis on code quality and reliability, we propose adoption of subsets of the MISRA C guidelines that are compatible with the project objectives and time frame. This is not in conflict with the possibility of adopting BARR-C:2018 proposed in [32]: the two approaches are complementary and both can be applied at the same time or incrementally.

We would like to stress that the notion of "MISRA compliance" is formalized in *MISRA Compliance:2020* [29] and is not something open to interpretation. In particular, there is no notion of "partial MISRA compliance" (the MISRA C guidelines constitute a whole that is much superior to any of its parts) and this paper does not mean to introduce it. Nonetheless, the availability of good static analyzers[12] supporting the MISRA C guidelines makes this opportunity valuable for many projects, especially those that cannot exclude the emergence of MISRA compliance requirements.

Acknowledgments

The MISRA C Guideline headlines are reproduced with permission of *The MISRA Consortium Limited*. We are grateful to the following BUGSENG collaborators: Simone Ballarin, for help with the experimental part of this work; Anna Camerini for the composition of Figure 1. We are also grateful to Frank

[12] I.e., roughly: confined false negatives, low rate of false positives, small configuration overhead, powerful reporting and filtering facilities, easy integration within CI/CD systems.

Büchner (Hitex GmbH) for a careful reading of draft versions of this paper, which resulted in several improvements.

References

[1] R. Bagnara, A. Bagnara, and P. M. Hill, "The MISRA C coding standard and its role in the development and analysis of safety- and security-critical embedded software," in *Static Analysis: Proceedings of the 25th International Symposium (SAS 2018)*, ser. Lecture Notes in Computer Science, A. Podelski, Ed., vol. 11002. Freiburg, Germany: Springer International Publishing, 2018, pp. 5–23.

[2] ——, "The MISRA C coding standard: A key enabler for the development of safety- and security-critical embedded software," in *embedded world Conference 2019 — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2019, pp. 543–553.

[3] IEC, *IEC 61508-1:2010: Functional Safety of Electrical/Electronic/Programmable Electronic Safety-Related Systems*. Geneva, Switzerland: IEC, Apr. 2010.

[4] ISO, *ISO 26262:2018: Road Vehicles — Functional Safety*. Geneva, Switzerland: ISO, Dec. 2018.

[5] CENELEC, *EN 50128:2011: Railway applications — Communication, signalling and processing systems - Software for railway control and protection systems*. CENELEC, Jun. 2011.

[6] RTCA, SC-205, *DO-178C: Software Considerations in Airborne Systems and Equipment Certification*. RTCA, Dec. 2011.

[7] *General Principles of Software Validation; Final Guidance for Industry and FDA Staff*, U.S. Department Of Health and Human Services; Food and Drug Administration; Center for Devices and Radiological Health; Center for Biologics Evaluation and Research, Jan. 2002, version 2.0, available at http://www.fda.gov/MedicalDevices/DeviceRegulationandGuidance/GuidanceDocuments/ucm085281.htm.

[8] MISRA, *MISRA-C:2012 — Guidelines for the use of the C language critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2019, third edition, first revision.

[9] Motor Industry Software Reliability Association, *MISRA-C:1998 — Guidelines for the use of the C language in vehicle based sofware*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jul. 1998.

[10] The Motor Industry Research Association, *Development Guidelines For Vehicle Based Software*. Nuneaton, Warwickshire CV10 0TU, UK: The Motor Industry Research Association, Nov. 1994.

[11] ISO/IEC, *ISO/IEC 9899:1999/Cor 3:2007: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2007, Technical Corrigendum 3.

[12] ——, *ISO/IEC 9899:1990/AMD 1:1995: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 1995.

[13] MISRA, *MISRA C:2012 Amendment 1 — Additional security guidelines for MISRA C:2012*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Apr. 2016.

[14] ISO/IEC, *ISO/IEC TS 17961:2013, Information technology — Programming languages, their environments & system software interfaces — C Secure Coding Rules*. Geneva, Switzerland: ISO/IEC, Nov. 2013.

[22] VV. AA., "JSF Air vehicle C++ coding standards for the system development and demonstration program," Lockheed Martin Corporation, Document 2RDU00001, Rev C, Dec. 2005.

[15] MISRA, *MISRA C:2012 Addendum 2 — Coverage of MISRA C:2012 (including Amendment 1) against ISO/IEC TS 17961:2013 "C Secure"*, 2nd ed. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jan. 2018.

[16] CERT, *SEI CERT C Coding Standard: Rules for Developing Safe, Reliable, and Secure Systems*. Software Engineering, Carnegie Mellon University, 2016, 2016 edition.

[17] MISRA, *MISRA C:2012 Addendum 3 — Coverage of MISRA C:2012 (including Amendment 1) against CERT C 2016 Edition*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jan. 2018.

[18] ——, *MISRA C:2012 Technical Corrigendum 1 — Technical clarification of MISRA C:2012*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Ltd, Jun. 2017.

[19] ——, *MISRA C:2012 Amendment 2 — Updates for ISO/IEC 9899:2011 Core functionality*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[20] ISO/IEC, *ISO/IEC 9899:2011: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2011.

[21] ——, *ISO/IEC 9899:2018: Programming Languages — C*. Geneva, Switzerland: ISO/IEC, 2018.

[23] M. Barr, *BARR-C:2018 — Embedded C Coding Standard*. www.barrgroup.com: Barr Group, 2018.

[24] ——, *Embedded C Coding Standard*. Germantown, MD, USA: Barr Group, 2013.

[25] ——, *Embedded C Coding Standard*. 6030 Marshalee Dr, #355 Elkridge, Maryland 21075, USA: Netrino, LLC, 2009.

[26] Motor Industry Software Reliability Association, *MISRA C++:2008 — Guidelines for the use of the C++ language in critical systems*. Nuneaton, Warwickshire CV10 0TU, UK: MIRA Ltd, Jun. 2008.

[27] VV. AA., "JPL institutional coding standard for the C programming language," Jet Propulsion Laboratory, California Institute of Technology, Tech. Rep. JPL DOCID D-60411, Mar. 2009.

[28] Software Engineering Center, *Embedded System Development Coding Reference: C Language Edition*. Information-Technology Promotion Agency, Japan, Jul. 2014, version 2.0.

[29] MISRA, *MISRA Compliance:2020 — Achieving compliance with MISRA Coding Guidelines*. Nuneaton, Warwickshire CV10 0TU, UK: HORIBA MIRA Limited, Feb. 2020.

[30] JF Bastien. (2019, Oct.) Making UB hurt less: security mitigations through automatic variable initialization. LLVM Foundation. Presentation Slides, 2019 LLVM Developers' Meeting — Bay Area. [Online]. Available: https://www.youtube.com/watch?v=I-XUHPimq3o

[31] P. Koopman. (2014, Sep.) A case study of Toyota unintended acceleration and software safety. Department of Electrical and Computer Engineering, Carnegie Mellon University. ECE Seminar Slides. [Online]. Available: https://users.ece.cmu.edu/~koopman/toyota/

[32] R. Bagnara, M. Barr, and P. M. Hill, "BARR-C:2018 and MISRA C:2012 (with Amendment 2): Synergy between the two most widely used C coding standards," in *embedded world Conference 2021 DIGITAL — Proceedings*, DESIGN&ELEKTRONIK, Ed. Nuremberg, Germany: WEKA FACHMEDIEN, Richard-Reitzner-Allee 2, 85540 Haar, Germany, 2021, pp. 378–391.